



# 历年 CSP 题目解析

仅为参考练习所用

作者: Ionlyn

组织: Shanxi University Algorithm Group

时间: December 30, 2021

版本: 1.0

## 特别声明

该书仅供内部学习使用，如有侵权请联系作者。

信息学竞赛的发展，吸引越来越多的人加入了“卷”的行列。CCF CSP 的历年题解在网上也是随处可见，但题解质量参差不齐。很多题解只有标准答案，缺少题目分析；更有甚者无法通过答案，充满了分号大小写问题等错误。

本书的目的是为了实现以下几点：

- 提供规范的代码程序。这里的规范，既要具有程序的可读性，也要具备考场的简易性。
- 提供多样的解题思路。有些时候，网上的大佬往往一语道破问题求解的思路，但怎么想到的却往往不提。这里力求从部分分开始，逐渐深入，汇集众人智慧，逐步解决难题。
- 提供筛选的额外补充。做一道题的目的不是只做一道题，而是可以做到举一反三，但我们常常忽略这一点。考虑到 CSP 考试的形式也在不断变化，本书章节编排从新到旧。

感谢 [Elegant \$\text{\LaTeX}\$](#)  提供如此精美的模板，希望这本书能够给大家带来帮助。

lonlyn

December 30, 2021

# 目录

1	CCF CSP 认证总览	1
2	第 24 次认证 (2021 年 12 月)	2
2.1	题目及涉及知识点	2
2.2	202112-1 序列查询	3
2.2.1	50% 数据——模拟	4
2.2.1.1	思路	4
2.2.1.2	C++ 实现	4
2.2.2	100% 数据——利用 $f(x)$ 单调性	5
2.2.2.1	思路	5
2.2.2.2	C++ 实现	5
2.2.3	100% 数据——阶段求和	6
2.2.3.1	思路	6
2.2.3.2	C++ 实现	6
2.3	202112-2 序列查询新解	7
2.3.1	与上一题的比较	9
2.3.2	70% 数据——计算出每个 $f(x), g(x)$ 的值	9
2.3.2.1	思路	9
2.3.2.2	C++ 实现	9
2.3.3	100% 数据——对 $f(x), g(x)$ 都相同的区间进行求和处理	9
2.3.3.1	思路	9
2.3.3.2	C++ 实现	10
2.3.4	100% 思路——以 $f(x)$ 为单位, 讨论内部 $g(x)$ 求和	11
2.3.4.1	思路	11
2.3.4.2	C++ 实现	11
2.4	202112-3 登机牌号码	15
2.4.1	40% 数据——直接模拟	19
2.4.1.1	思路	19
2.4.1.2	C++ 实现	19
2.4.2	100% 数据——模拟 + 多项式除法	21
2.4.2.1	思路	21
2.4.2.2	C++ 实现	22
2.5	202112-4 磁盘文件操作	26
2.5.1	25% 数据——直接模拟	27
2.5.1.1	思路	27
2.5.2	100% 数据——离散化 + 线段树	28
2.5.2.1	思路	28
2.5.2.2	C++ 实现	29
2.5.3	100% 数据——动态开点线段树	35
2.5.3.1	思路	35
2.5.3.2	C++ 实现	36

---

2.6	202112-5 极差路径 . . . . .	42
<b>3</b>	<b>第 23 次认证 (2021 年 09 月)</b>	<b>45</b>
3.1	题目及设计知识点 . . . . .	45
3.2	202109-1 数组推导 . . . . .	46
3.3	202109-2 非零段划分 . . . . .	48
3.4	202109-3 脉冲神经网络 . . . . .	50
3.5	202109-4 收集卡牌 . . . . .	54
3.6	202109-5 箱根山岳险天下 . . . . .	56
<b>4</b>	<b>第 22 次认证 (2021 年 04 月)</b>	<b>59</b>
4.1	题目及设计知识点 . . . . .	59
4.2	202104-1 灰度直方图 . . . . .	60
4.3	202104-2 邻域均值 . . . . .	62
4.4	202104-3 DHCP 服务器 . . . . .	64
4.5	202104-4 校门外的树 . . . . .	67
4.6	202104-5 疫苗运输 . . . . .	70

# 第 1 章 CCF CSP 认证总览

待补充。

## 第 2 章 第 24 次认证（2021 年 12 月）

### 2.1 题目及涉及知识点

题目编号	题目名称	知识点
202112-1	序列查询	数学
202112-2	序列查询新解	数学
202112-3	登机牌条码	模拟，多项式除法
202112-4	磁盘文件操作	线段树
202112-5	极差路径	树分治

## 2.2 202112-1 序列查询

### 题目背景

西西艾弗岛的购物中心里店铺林立，商品琳琅满目。为了帮助游客根据自己的预算快速选择心仪的商品，IT 部门决定研发一套商品检索系统，支持对任意给定的预算  $x$ ，查询在该预算范围内 ( $\leq x$ ) 价格最高的商品。如果没有商品符合该预算要求，便向游客推荐可以免费领取的西西艾弗岛定制纪念品。

假设购物中心里有  $n$  件商品，价格从低到高依次为  $A_1, A_2, \dots, A_n$ ，则根据预算  $x$  检索商品的过程可以抽象为如下序列查询问题。

### 题目描述

$A = [A_0, A_1, A_2, \dots, A_n]$  是一个由  $n+1$  个  $[0, N)$  范围内整数组成的序列，满足  $0 = A_0 < A_1 < A_2 < \dots < A_n < N$ 。（这个定义中蕴含了  $n$  一定小于  $N$ 。）

基于序列  $A$ ，对于  $[0, N)$  范围内任意的整数  $x$ ，查询  $f(x)$  定义为：序列  $A$  中小于等于  $x$  的整数里最大的数的下标。具体来说有以下两种情况：

1. 存在下标  $0 \leq i < n$  满足  $A_i \leq x < A_{i+1}$ ，此时序列  $A$  中从  $A_0$  到  $A_i$  均小于等于  $x$ ，其中最大的数为  $A_i$ ，其下标为  $i$ ，故  $f(x) = i$ 。
2.  $A_n \leq x$ ，此时序列  $A$  中左右的数都小于等于  $x$ ，其中最大的数是  $A_n$ ，故  $f(x) = n$ 。

令  $sum(A)$  表示  $f(0)$  到  $f(N-1)$  的总和，即：

$$sum(A) = \sum_{i=0}^{N-1} f(i) = f(0) + f(1) + f(2) + \dots + f(N-1)$$

对于给定的序列  $A$ ，试计算  $sum(A)$ 。

### 输入格式

从标准输入读入数据。

输入的第一行包含空格分隔的两个正整数  $n$  和  $N$ 。

输入的第二行包含  $n$  个用空格分隔的整数  $A_1, A_2, \dots, A_n$ 。

注意  $A_0$  固定为 0，因此输入数据中不包括  $A_0$ 。

### 输出格式

输出到标准输出。

仅输出一个整数，表示  $sum(A)$  的值。

### 样例 #1

输入 #1:

```
3 10
2 5 8
```

输出 #1:

```
15
```

解释 #1:

$A = [0, 2, 5, 8]$

$i$	0	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	1	1	1	2	2	2	3	3

如上表所示,  $sum(A) = f(0) + f(1) + \dots + f(9) = 15$ 。

考虑到  $f(0) = f(1)$ 、 $f(2) = f(3) = f(4)$ 、 $f(5) = f(6) = f(7)$  以及  $f(8) = f(9)$ , 亦可通过如下算式计算  $sum(A)$ :

$$sum(A) = f(0) \times 2 + f(2) \times 3 + f(5) \times 3 + f(8) \times 2$$

## 样例 #2

输入 #2:

输出 #2:

9 10  
1 2 3 4 5 6 7 8 9

45

## 子任务

50% 的测试数据满足  $1 \leq n \leq 200$  且  $n \leq N \leq 1000$ ;

全部的测试数据满足  $1 \leq n \leq 200$  且  $n \leq N \leq 10^7$ 。

## 提示

若存在区间  $[i, j)$  满足  $f(i) = f(i+1) = \dots = f(j-1)$ , 使用乘法运算  $f(i) \times (j-i)$  代替将  $f(i)$  到  $f(j-1)$  逐个相加, 或可大幅提高算法效率。

### 2.2.1 50% 数据——模拟

#### 2.2.1.1 思路

模拟一下这个过程, 计算出每一个  $f(i)$  后加起来即可。

考虑针对确定的  $x$ , 如何求解  $f(x)$ 。我们可以从小到大枚举  $A$  中的数, 枚举到第一个大于等于  $x$  的数即可。注意末尾的判断。

枚举  $x$  时间复杂度  $O(N)$ , 计算  $f(x)$  时间复杂度  $O(n)$ , 整体时间复杂度  $O(nN)$ 。

#### 2.2.1.2 C++ 实现

待补充。



## 2.2.2 100% 数据——利用 $f(x)$ 单调性

### 2.2.2.1 思路

为了方便，设  $f(n+1) = \infty$ 。

通过模拟，可以得到一个显然的结论：

#### 定理 2.1 ( $f(x)$ 的单调性)

对于  $x, y \in [0, N)$ ，若  $x \leq y$ ，则  $f(x) \leq f(y)$ 。



那么，我们可以从小到大枚举  $x$ ，同时记录目前  $f(x)$  的值，设为  $y$ ，那么  $A_{y+1}$  是第一个大于  $x$  的数。当需要计算  $f(x+1)$  的时候，我们从小到大依次判断  $A_{y+1}, A_{y+2}, \dots$  是否满足条件，直到遇到第一个大于  $f(x+1)$  的数  $A_z$ ，那么  $f(x+1) = z - 1$ 。之后，在  $f(x+1)$  的基础上以同样的步骤求  $f(x+2)$ ，直到求完所有的值。

考虑该算法的时间复杂度，枚举  $x$  的复杂度是  $O(N)$ ，而  $A$  数组中每个数对多被枚举一次，枚举所有  $x$  的整体复杂度  $O(n)$ ，可以得到整体复杂度  $O(N+n)$ 。

### 2.2.2.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define ll long long
#define il inline
const int maxn = 210;
int n, N;
int a[maxn];
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    int cur = 0;
    for (int i = 0; i < N; ++i) {
        while (cur < n && a[cur + 1] <= i)
            ++cur;
        ans += cur;
    }
    printf("%lld\n", ans);
    return 0;
}
```

## 2.2.3 100% 数据——阶段求和

### 2.2.3.1 思路

在提示中，指出了可以将  $f(x)$  相同的值一起计算。现在需要解决的问题是如何快速确定  $f(x)$  值相等的区间。

通过观察和模拟可以发现，随着  $x$  增大， $f(x)$  只会在等于某个  $A$  数组的值时发生变化。更具体的说，对于某个属于  $A$  数组的值  $A_i$  来说， $[A_i, A_{i+1} - 1]$  间的  $f(x)$  值是相同的，这样的数共有  $A_{i+1} - A_i$  个。

也可以以另一种方式理解：对于一个值  $y$ ，考虑有多少  $x$  满足  $f(x) = y$ 。当  $x < A_y$  时， $f(x) < y$ ，当  $x \geq A_{y+1}$  时， $f(x) > y$ 。只有  $x \in [A_y, A_{y+1}]$  时才能得到  $f(x) = y$ 。

得到范围后，我们就可以根据  $A$  数组来进行求和计算。

考虑  $f(x) = n$  的处理：我们可以得知满足  $f(x) = n$  的  $x$  共有  $N - A_n$  个，根据上文推算，我们可以将  $A_{n+1}$  设置为  $A_n + (N - A_n) = N$  即可等效替代。

时间复杂度  $O(n)$ 。

### 2.2.3.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define ll long long
#define il inline
const int maxn = 210;
int n, N;
int a[maxn];
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    a[n + 1] = N;
    for (int i = 1; i <= n + 1; ++i) {
        // 处理区间 [A(i-1), A(i)] 的 f(x) 值的和
        ans += 1ll * (a[i] - a[i - 1]) * (i - 1);
    }
    printf("%lld\n", ans);
    return 0;
}
```

## 2.3 202112-2 序列查询新解

### 题目背景

上一题“序列查询”中说道： $A = [A_0, A_1, A_2, \dots, A_n]$  是一个由  $n+1$  个  $[0, N)$  范围内整数组成的序列，满足  $0 = A_0 < A_1 < A_2 < \dots < A_n < N$ 。基于序列  $A$ ，对于  $[0, N)$  范围内任意的整数  $x$ ，查询  $f(x)$  定义为：序列  $A$  中小于等于  $x$  的整数里最大的数的下标。

对于给定的序列  $A$  和整数  $x$ ，查询  $f(x)$  是一个很经典的问题，可以使用二分搜索在  $O(\log n)$  的时间复杂度内轻松解决。但在 IT 部门讨论如何实现这一功能时，小 P 同学提出了些新的想法。

### 题目描述

小 P 同学认为，如果事先知道了序列  $A$  中整数的分布情况，就能直接估计出其中小于等于  $x$  的最大整数的大致位置。接着从这一估计位置开始线性查找，锁定  $f(x)$ 。如果估计得足够准确，线性查找的时间开销可能比二分查找算法更小。

比如说，如果  $A_1, A_2, \dots, A_n$  均匀分布在  $(0, N)$  的区间，那么就可以估算出：

$$f(x) \approx \frac{(n+1) \cdot x}{N}$$

为了方便计算，小 P 首先定义了比例系数  $r = \lfloor \frac{N}{n+1} \rfloor$

，其中  $\lfloor \cdot \rfloor$  表示下取整，即  $r$  等于  $N$  除以  $n+1$  的商。进一步地，小 P 用  $g(x) = \lfloor \frac{x}{r} \rfloor$

表示自己估算出的  $f(x)$  的大小，这里同样使用了下取整来保证  $g(x)$  是一个整数。

显然，对于任意的询问  $x \in [0, N)$ ， $g(x)$  和  $f(x)$  越接近则说明小 P 的估计越准确，后续进行线性查找的时间开销也越小。因此，小 P 用两者差的绝对值  $|g(x) - f(x)|$  来表示处理询问  $x$  时的误差。

为了整体评估小 P 同学提出的方法在序列  $A$  上的表现，试计算：

$$error(A) = \sum_{i=0}^{N-1} |g(i) - f(i)| = |g(0) - f(0)| + \dots + |g(N-1) - f(N-1)|$$

### 输入格式

从标准输入读入数据。

输入的第一行包含空格分隔的两个正整数  $n$  和  $N$ 。

输入的第二行包含  $n$  个用空格分隔的整数  $A_1, A_2, \dots, A_n$ 。

注意  $A_0$  固定为 0，因此输入数据中不包括  $A_0$ 。

### 输出格式

输出到标准输出。

仅输出一个整数，表示  $error(A)$  的值。

### 样例 #1

输入 #1:

输出 #1:

3 10  
2 5 8

5

解释 #1:

$A = [0, 2, 5, 8]$   
 $r = \lfloor \frac{N}{n+1} \rfloor = \lfloor \frac{10}{3+1} \rfloor = 2$

<i>i</i>	0	1	2	3	4	5	6	7	8	9
<i>f(i)</i>	0	0	1	1	1	2	2	2	3	3
<i>g(i)</i>	0	0	1	1	2	2	3	3	4	4
$ g(i) - f(i) $	0	0	0	0	1	0	1	1	1	1

样例 #2

输入 #2:

输出 #2:

9 10  
1 2 3 4 5 6 7 8 9

0

样例 #3

输入 #3:

输出 #3:

2 10  
1 3

6

解释 #3:

$A = [0, 1, 3]$   
 $r = \lfloor \frac{N}{n+1} \rfloor = \lfloor \frac{10}{2+1} \rfloor = 3$

<i>i</i>	0	1	2	3	4	5	6	7	8	9
<i>f(i)</i>	0	1	1	2	2	2	2	2	2	2
<i>g(i)</i>	0	0	0	1	1	1	2	2	2	3
$ g(i) - f(i) $	0	1	1	1	1	1	0	0	0	1

## 子任务

70 % 的测试数据满足  $1 \leq n \leq 200$  且  $n \leq N \leq 1000$ ;

全部的测试数据满足  $1 \leq n \leq 10^5$  且  $n \leq N \leq 10^9$ 。

## 提示

需要注意, 输入数据  $[A_1 \cdots A_n]$  并不一定均匀分布在  $(0, N)$  区间, 因此总误差  $error(A)$  可能很大。

### 2.3.1 与上一题的比较

1. 上一题是求和, 而本题要求求绝对值的和, 无法转化为两者求差的形式。
2.  $f(x), g(x)$  的变化是各自独立的, 当  $f(x)$  改变时,  $g(x)$  可能不变, 也可能改变;  $g(x)$  对  $f(x)$  也是如此。
3. 对于所有数据点,  $n$  和  $N$  都增大了许多。如果复杂度涉及到  $n$ , 则最多预计为  $O(n \log n)$  级别; 如果涉及到  $N$ , 则必须是亚线性级别。

### 2.3.2 70% 数据——计算出每个 $f(x), g(x)$ 的值

#### 2.3.2.1 思路

由于 1,2 条限制, 我们无法直接对  $f(x), g(x)$  分别进行处理。但我们可以求出每个  $f(x), g(x)$  的值, 再计算求和即可。

$f(x)$  的计算同第一问, 任意方法皆可。单个  $g(x)$  的值可以直接  $O(1)$  求得。

#### 2.3.2.2 C++ 实现

待补充

### 2.3.3 100% 数据——对 $f(x), g(x)$ 都相同的区间进行求和处理

#### 2.3.3.1 思路

注: 为了防止混淆, 将题目中的  $r$  改为  $ratio$ 。

假设  $f(x)$  一共有  $x$  种取值,  $g(x)$  一共有  $y$  种取值。直接来看  $f(x), g(x)$  的组合一共有  $xy$  种, 但注意到  $f(x), g(x)$  都是单调不递减函数, 所以真正的组合只有  $x + y$  种。

在第一题中已经说明  $f(x)$  的取值范围为  $[0, n]$ , 在  $O(n)$  级别。考虑  $g(x)$  的取值情况, 将  $ratio$  的公式带入可以得到  $g(x) = \lfloor \frac{x}{ratio} \rfloor = \lfloor \frac{\frac{x}{N}}{\frac{1}{n+1}} \rfloor$ 。由于  $x$  取值有  $N$  种, 所以  $g(x)$  的取值是  $O(\frac{N}{\frac{1}{n+1}}) = O(n)$  级别的。所以, 整体复杂度为  $O(n + n) = O(n)$ 。



**提示** 有些时候, 题目给出的某些量的值会比较特殊 (如本题  $ratio = \lfloor \frac{N}{n+1} \rfloor$ ), 代表着出题人可能想要隐藏某些做法, 但不得不为了让时间复杂度正确而妥协。在没有思路的时候, 可以作为突破口。

考虑范围问题: 假设当前左端点为  $l$ , 如何找到右端点  $r$ , 满足  $f(l) = f(l+1) = \cdots = f(r), g(l) = g(l+1) = \cdots = g(r)$  且  $f(l) \neq f(r+1)$  or  $g(l) \neq g(r+1)$ 。我们可以对  $f(x), g(x)$  分别考虑:

1. 对于  $f(x)$  而言, 第一个满足  $f(x) = f(l) + 1$  的  $x$  值为  $A_{f_l+1}$ 。
2. 对于  $g(x)$  而言, 因为分母  $ratio$  是固定的, 所以值相同的区间长度也是固定为  $ratio$ 。我们不妨将  $g(x)$  值相同的数字为一组, 则可以得到  $[0, ratio - 1], [ratio, 2 \cdot ratio - 1], \cdots, [n \cdot ratio, (n+1) \cdot ratio - 1], \cdots$  这样的分组序列, 每组的  $g(x)$  取值为  $0, 1, \cdots, n, \cdots$ 。可以发现, 对于一个数  $l$ , 其所属的分组是  $\lfloor \frac{l}{ratio} \rfloor$ , 也即  $g(l)$ ; 而下一组开始的第一个数为  $ratio \cdot (g(l) + 1)$ , 从而可以得到右端点  $r = ratio \cdot (g(l) + 1) - 1$ 。
3. 在  $f(x), g(x)$  计算得到的右端点中, 选择较小的一个作为计算的右端点。

计算完一段后, 设  $l = r + 1$  继续计算下一段, 直到结束。时间复杂度  $O(n)$ 。

## 2.3.3.2 C++ 实现

```

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define ll long long
const int maxn = 100010;
int n, N;
int a[maxn];
int rat, f, g;
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    rat = N / (n + 1); // 为了防止冲突，题目中 r 改为 rat
    int cur = 0; // 用来计算 f(x)
    bool flag = false; // 如果需要更新 f(x) 值，则 flag = true
    for (int l = 0, r; l < N; l = r + 1) {
        flag = false;
        // 利用 f(x) 的值确定 r 的范围
        if (cur < n)
            r = a[cur + 1] - 1;
        else
            r = N - 1;
        // 判断 f(x), g(x) 谁先变化，选择较小的区间
        if ((l / rat + 1) * rat - 1 < r) {
            // 如果 g(x) 先变化，则改为选择 g(x)
            r = (l / rat + 1) * rat - 1;
        } else {
            // 如果 f(x) 先变化，则确定选择 f(x)，计算后更新 f(x)
            flag = true;
        }
        // [l, r] 区间内的值是相等的，可以求和
        ans += 1ll * (r - l + 1) * abs(l / rat - cur);
        // 更新 f(x) 的值
        if (flag)
            ++cur;
    }
    printf("%lld\n", ans);
    return 0;
}

```

### 2.3.4 100% 思路——以 $f(x)$ 为单位，讨论内部 $g(x)$ 求和

感谢 DoctorLazy 提供的宝贵思路，原文可以查看 [第二题 100 分题解 by DoctorLazy.md](#)。

#### 2.3.4.1 思路

和上文同样的思路，我们需要进行区间求和来降低复杂度。一种思路是，整体上对  $f(x)$  进行求和，而在内部对  $g(x)$  的情况进行分类讨论。

我们单独考虑每一个  $f(x)$  的区间，每个区间上  $f(x)$  的值相同。可以观察到，对于一个区间上的下标  $i$ ，可能存在  $g(i) \geq f(i)$ ，也可能存在  $g(i) < f(i)$ 。求绝对值时，前者用  $g(x) - f(x)$ ，后者用  $f(x) - g(x)$ 。

观察到，由于  $g(x)$  单调不减的性质，我们可以得到：对于该区间，一定存在一个下标  $p$ ，如同一个分界线，当  $i \geq p$  时，有  $g(i) \geq f(i)$ ，当  $i < p$ ，有  $g(i) < f(i)$ 。这样，就把该区间分成了两个“小区间”。我们就可以用“乘法思想”来加速两个“小区间”的求解了。

更规范些，用  $contribution(i)$  代表区间  $[A_i, A_{i+1})$  对答案的贡献，用  $len(l, r) = r - l + 1$  代表区间长度，用公式可以表达为：

$$\begin{aligned} contribution(i) = & len(A_i, p-1) \times f(x) - \sum_{x=A[i]}^{p-1} g(x) \\ & + \sum_{x=p}^{A_{i+1}-1} g(x) - len(p, A_{i+1}-1) \times f(x) \end{aligned}$$

上式中， $f(x)$  是一个常数，所以乘以“小区间”的长度即可； $g(x)$  的求和，大家可以发挥数学思维：因为  $g(x)$  其实非常规律，它的每一块是定长的，我们可以通过除法和取余来确定相同值的数量，再利用乘法思想求和，灵活实现，在  $O(n)$  时间内求出即可。 $p$  的具体值可以通过在  $g(x)$  中二分查找， $O(\log n)$  时间内求出， $n$  为区间的长度。

一个例子：

$x$	...	4	5	6	7	...
$f(x)$	...	2	2	2	2	...
$g(x)$	...	1	1	2	2	...

上面的表格截取了一个小区间， $f(x)$  的值固定 2， $p = 6$ ，那么  $p$  的左边用  $f(x) - g(x)$ ， $p$  的右边用  $g(x) - f(x)$ 。

当然，有一个特殊的边界情况，那就是该区间上有可能所有的  $g(x)$  都绝对大于或小于  $f(x)$ ，这时候  $p$  可能会在区间外。该情况大家可以对  $p$  设置初值，然后在写完二分后加以判断即可。

#### 2.3.4.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <map>
#include <queue>
#include <vector>
using namespace std;
```

```

typedef long long LL;
typedef unsigned long long uLL;
typedef pair<int, int> pii;
const int mod = 1e9 + 7;
const int maxn = 1e5 + 5;

LL N, n;
LL arr[maxn]; // 题中 A 数组
vector<LL> f; // 存储每个区间上f的值
vector<pii> pos; // 存储每个区间的边界，是左闭右闭
LL r, ans; // 题中的 r, ans为计算的答案

// 下面的函数用于计算g(x)在区间上的和
// 这一步比较细，具体可以灵活实现
// 下面的思路还是比较冗杂的
LL totG(LL be, LL ed) {
    // 右边界小于左边界，返回0
    if (ed < be) {
        return 0;
    }
    // 两边界重合，返回一个g值
    if (be == ed) {
        return be / r;
    }
    // 如果两边界g值相同，返回该值乘以区间长度
    if (be / r == ed / r) {
        return (be / r) * (ed - be + 1);
    }
    // 将区间分为三部分，分别累计
    LL tot = 0;
    // 对于左边界，其值为be/r,数目为 r - be % r
    tot += (r - (be % r)) * (be / r);
    // 对于右边界，其值为ed/r,数目为 ed % r + 1
    tot += (ed % r + 1) * (ed / r);
    // 对于不在边界上的g值，我们用等差数列求和公式
    if (ed / r - be / r > 1) {
        be = be / r + 1;
        ed = ed / r - 1;
        tot += r * ((be + ed) * (ed - be + 1) / 2);
    }
    return tot;
}

void solve() {
    // 输入
    scanf("%lld%lld", &n, &N);
    r = N / (n + 1);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
    }
}

```



```

}
// 根据数组，生成f(x)的每个区间，值存入f，区间边界存入pos
LL last = 0ll; // 记录上一个边界
// 这里的逻辑参考第一题
for (int i = 1; i <= n; i++) {
    if (arr[i] > arr[last]) {
        f.push_back(last);
        pos.push_back({arr[last], arr[i] - 1});
        last = i;
    }
}
// 单独处理下最后一个区间，即[A[n],N-1]
f.push_back(n);
pos.push_back({arr[last], N - 1});
// 对于每个f区间，将g分成两个小区间
int si = f.size();
for (int i = 0; i < si; i++) {
    LL num = f[i];
    LL be = pos[i].first;
    LL ed = pos[i].second;
    LL length = ed - be + 1;
    // 因为be和ed在二分过程其值发生变化，所以下面再存一份
    LL bbe = be, eed = ed;
    // 下面使用二分，在g中寻找分界p
    LL pin = -1;
    while (be <= ed) {
        LL mid = (be + ed) / 2;
        LL cur = mid / r;
        if (cur >= num) {
            pin = mid;
            ed = mid - 1;
        } else {
            be = mid + 1;
        }
    }
    // 如果f的值一直大于g，p值不会被二分的过程赋值，所以还是初值
    if (pin == -1) {
        ans += num * length - totG(bbe, eed);
    } else {
        // 左边的用f-g，右边用g-f。就算g的值一直大于f，即左边的部分长度为0
        ans += num * (pin - bbe) - totG(bbe, pin - 1);
        ans += totG(pin, eed) - num * (eed - pin + 1);
    }
}
printf("%lld", ans);
}

int main() {
    int t;

```

```
t = 1;  
while (t--) {  
    solve();  
}  
return 0;  
}
```

2.4 202112-3 登机牌号码

题目背景

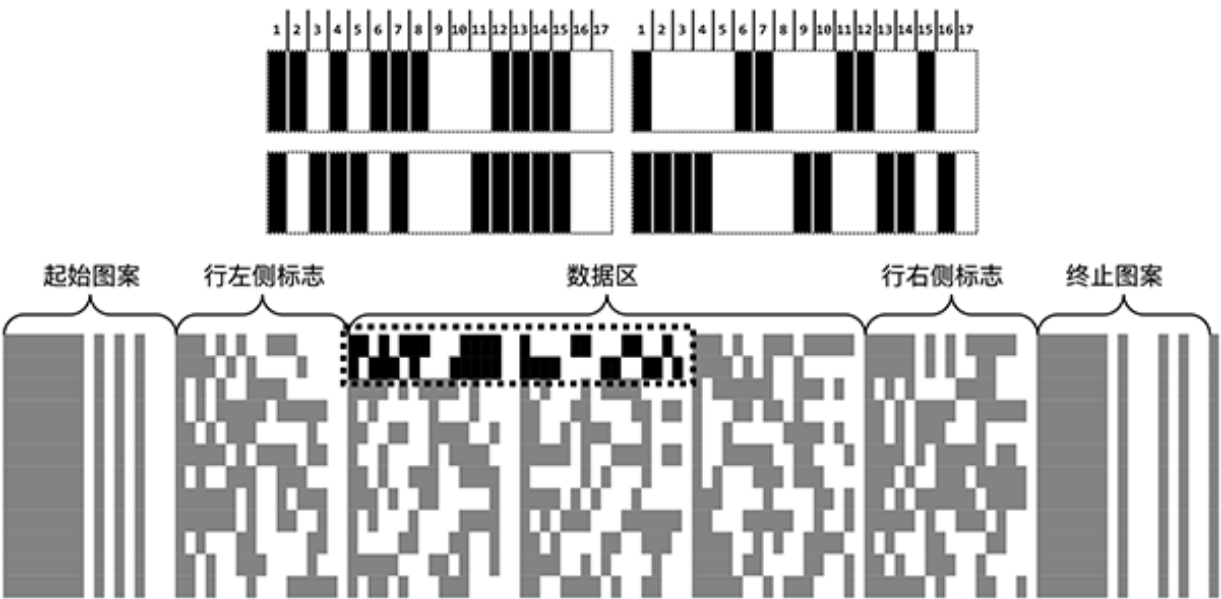
西西艾弗岛景色优美，游人如织。但是，由于和外界的交通只能靠渡船，交通的不便严重制约了岛上旅游业的发展。西西艾弗岛管委会经过努力，争取到了一笔投资，建设了一个通用航空机场。在三年紧锣密鼓的主体建设后，西西艾弗岛通用航空机场终于开始进行航站楼内部软硬件系统的安装和调试工程了。小 C 是机场运营公司信息部的研发工程师，最近，信息部门的一项重要任务是，研发登机牌自助打印系统。如图所示的是设计部门根据国际民航组织的行业标准设计的登机牌样张。



登机牌上最重要的部分就是最下方的机读条形码了。小 C 承担了生成机读条形码算法的开发工作。从被编码的数据到条形码，中间有好多步骤要走。小 C 请你来帮忙，让你帮忙处理一下数据编码的问题。

题目描述

登机牌上的条形码，是 PDF417 码。PDF417 码的结构如下图所示。



PDF417 码组成的基本元素是码元 (Module)，所有的码元都是等大的矩形，填充有黑色或白色。码元先组成行，若干行堆叠组成整个 PDF417 码。每一行中，每 17 个码元表示一个码字 (Code word)。码字是 PDF417 编码中的最小数据单位。每个码字图案中，有交替排列的四个黑色矩形和四个白色矩形，这便是“417”的由来。每行开始和结尾有固定的起始和中止图案。与他们相邻的是行左侧和右侧标志，表示行号、行内码字个数等信息。中间的是有效数据区。编码的步骤是：先按照编码规则，将被编码的数据转换为码字；接着根据选定 PDF417 码的宽度（即每行码字的数目）以及冗余程度计算校验码字；最后将码字按规则转换为对应的图案，并按照从左至右，从上至下的顺序填入有效数据区，并与起始终止图案和行左右标志拼合，形成完整的 PDF417 码。

每个码字是一个 0 至 928 之间的数字，每个码字可以编码两个输入字符。对于输入的被编码的数据，按照下表进行编码。编码器共有三种模式：大写字母模式、小写字母模式和数字模式。在编码开始时，编码器处于大写字母模式。编码器处于某种模式时，仅能编码对应类型的字符，如果需要编码其它类型的字符，需要通过特殊值切换到对应模式下。要进行模式切换，可以有多种切换方法。例如，要从大写模式切入小写模式，可以直接用 27 切入，也可以先用 28 切入数字模式后立刻再用 27 切入小写模式。你需要选择最短的方式进行切换，因此只有前一种方法是正确的。需要注意的是，从小写模式不能直接切入大写模式，必须要经过数字模式过渡。

值	大写模式	小写模式	数字模式
0	A	a	0
1	B	b	1
2	C	c	2
3	D	d	3
4	E	e	4
5	F	f	5
6	G	g	6
7	H	h	7
8	I	i	8
9	J	j	9
10	K	k	
11	L	l	
12	M	m	
13	N	n	
14	O	o	
15	P	p	
16	Q	q	
17	R	r	
18	S	s	
19	T	t	
20	U	u	
21	V	v	
22	W	w	
23	X	x	
24	Y	y	
25	Z	z	
27	小写		小写
28	数字	数字	大写
29	填充	填充	填充

按照这个方法可以得到一系列的不超过 30 的数字。如果有奇数个这样的数字，则在最后补充一个 29，使之成为偶数个。将它们两两成组，假设  $H$  和  $L$  是一组中连续出现的两个数字，那么可以得到一个码字是：

$$30 \times H + L$$

例如，要编码 “HEllo”，首先根据字母表，产生数字序列：

H E 1 1 o  
7 4 28 1 27 11 14

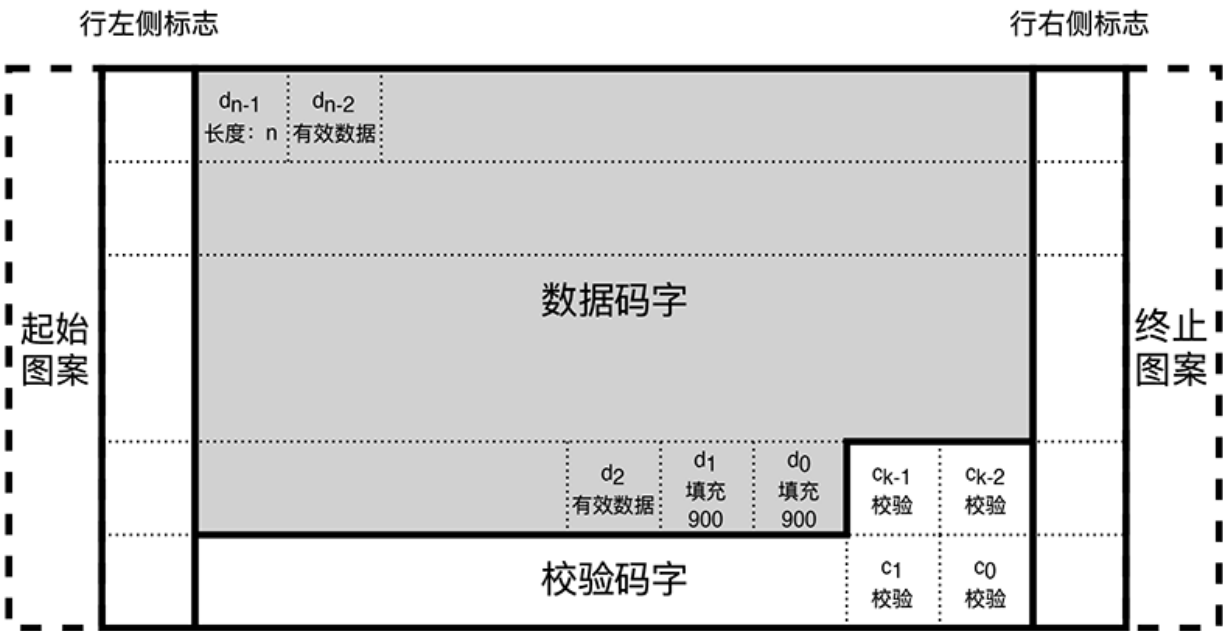
由于只有奇数个数字，需要在末尾补充 29，然后将它们两两成组：

(7, 4), (28, 1), (27, 11), (14, 29)

最后计算码字，例如：  $30 \times 7 + 4 = 214$ ，以此类推，可以得到码字为：

214, 841, 821, 449

接下来要计算校验码。校验码字的数目，由校验级别确定。假设校验级别为  $s(0 \leq s \leq 8)$ ，则校验码字的数目为  $k = 2^{s+1}$ 。特别地，如果指定了  $s = -1$ ，则表示不需要计算校验码字。要计算校验码字，首先要确定数据码字。数据码字由以下数据按顺序拼接而成（如图所示）：



- 一个长度码字，表示全部数据码字的个数  $n$ ，包括该长度码字、有效数据码字、填充码字；
- 若干有效数据码字，是此前计算的码字序列；
- 零个或多个由重复的 900 组成的填充码字，使得包括校验码字在内的码字总数恰能被有效数据区的行宽度整除。

设全部数据码字依次为  $d_{n-1}, d_{n-2}, \dots, d_0$ ；校验码字依次为  $c_{k-1}, c_{k-2}, \dots, c_0$ 。那么校验码字按照如下方式计算：

取  $k$  次多项式  $g(x) = (x - 3)(x - 3^2) \cdots (x - 3^k)$ ， $(n - 1)$  次多项式  $d(x) = d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + \cdots + d_1x + d_0$ ，找到多项式  $q(x)$  和不超过  $k - 1$  次的多项式  $r(x)$ ，使得

$$x^k d(x) \equiv q(x)g(x) - r(x)$$

那么多项式  $r(x)$  中  $x$  的  $i$  次项系数对 929 取模后（取正值）的数字即为校验码字  $c_i$ 。

例如，如果要将 `HEllo` 编码为 PDF417 条码，且有效数据区的行宽是 4 码字（即 68 码元），校验级别为 0。此时校验码字有两个。根据此前的编码结果，有效数据码字有 4 个。再加上一个长度码字，共有 7 个码字。因此需要补充一个填充码字，使包括校验码字在内的总码字数能够被 4 整除。这样，用于计算校验码字的数据码字有 6 个，分别是：

6, 214, 841, 821, 449, 900

因此有  $g(x) = x^2 - 12x + 27$ ， $d(x) = 6x^5 + 214x^4 + 841x^3 + 821x^2 + 449x + 900$ ，不难得到  $r(x) = -32902164x + 98246277$ ，因此相应可以计算出  $c_1 = 299 \equiv -32902164 \pmod{929}$ ， $c_0 = 811 \equiv 98246277 \pmod{929}$ 。这样，全部码字序列即为：

6, 214, 841, 821, 449, 900, 229, 811

在本题中，你需要帮助小 C 完成的任务是，给定被编码的数据，计算出需要填入有效数据区的码字序列。被处理的数据中只含有大写字母、小写字母和数字。

## 输入格式

从标准输入读入数据。

输入的第一行包含两个用空格分隔的整数  $w$ 、 $s$ ，分别表示有效数据区每行能容纳的码字数和校验级别。保证  $0 < w < 929$ ， $-1 \leq s \leq 8$ 。特别地，当  $s = -1$  时，表示不需要计算校验码字。

输入的第二行是一个非空字符串，仅包含大小写字母和数字，长度保证编码后全部数据码字的个数少于 929。

## 输出格式

输出到标准输出。

输出若干行，每行一个数字，表示编码后的全部码字序列。

## 样例 #1

输入 #1:

5 -1  
HELLO

输出 #1:

5  
214  
341  
449  
900

解释 #1:

要求编码数据是 `HELLO`，首先查表将其对应成数字。注意，由于编码器在开始时就处于大写字母模式，因此不需要额外的模式切换。因此对应成的数字为：7, 4, 11, 11, 14。由于只有奇数个数字，因此补充 29，形成序列 7, 4, 11, 11, 14, 29。然后两两成组计算码字： $7 \times 30 + 4 = 214$ ，以此类推，得到 214, 341, 449。本输入不要求产生校验码，且有效数据区的宽度是 5 码字。目前有效数据的码字是 3 个，加上开头要添加的长度码字，共有 4 个码字。因此，需要补充一个填充码字，使得总码字数达到 5 个，充满一行。注意，长度码字中的长度数据包括所有数据码字，因此长度码字是 5 而不是 4。最终可以得到码字序列 5, 214, 341, 449, 900。

## 样例 #2

输入 #2:

```
4 0
HE11o
```

输出 #2:

```
6
214
841
821
449
900
229
811
```

解释 #2:

本组数据即为此前用于说明编码过程的示例。

## 子任务

对于 20% 的数据，有  $s = -1$ ，且输入字符串中仅含有大写字母或小写字母；

对于 40% 的数据，有  $s = -1$ ；

对于 80% 的数据，有  $s \leq 2$ ；

对于 100% 的数据，满足全部对于输入的要求。

### 2.4.1 40% 数据——直接模拟

#### 2.4.1.1 思路

这一部分数据满足  $s = -1$ ，即校验码为空。我们按照题目要求进行对应操作即可，大体分为以下几个步骤：

1. 得到数字序列，注意不同模式的切换以及最后的补全。
2. 将得到的数字转换为码字。
3. 根据有效数据区每行能容纳的码字数  $w$  及目前码字个数，在末尾补充码字。注意不要忽略长度码字。
4. 输出结果。

#### 2.4.1.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
const int maxn = 210;
const int mod = 929;
```

```

int w, lev;
char s[100010];
int n;
vector<int> numberList; // 数字序列
vector<int> codeWord; // 码字序列
int currentMode; // 目前编码器模式。0:大写模式 1:小写模式 2:数字模式
void checkmode(char c) {
    /*
        检查将要输出的下个字符与目前模式是否匹配，
        若不匹配，则输出对应更改模式步骤。
    */
    if (c >= '0' && c <= '9') {
        if (currentMode != 2) {
            numberList.push_back(28);
            currentMode = 2;
        }
    } else if (c >= 'a' && c <= 'z') {
        if (currentMode != 1) {
            numberList.push_back(27);
            currentMode = 1;
        }
    } else if (c >= 'A' && c <= 'Z') {
        if (currentMode == 1) {
            numberList.push_back(28);
            numberList.push_back(28);
            currentMode = 0;
        }
        if (currentMode == 2) {
            numberList.push_back(28);
            currentMode = 0;
        }
    }
}

int main() {
    scanf("%d%d", &w, &lev); // lev 表示校验级别
    scanf("%s", s);
    n = strlen(s);
    // 步骤一：得到数字序列
    currentMode = 0; // 初始为大写模式
    for (int i = 0; i < n; ++i) {
        checkmode(s[i]);
        if (s[i] >= '0' && s[i] <= '9') {
            numberList.push_back(s[i] - '0');
        } else if (s[i] >= 'a' && s[i] <= 'z') {
            numberList.push_back(s[i] - 'a');
        } else if (s[i] >= 'A' && s[i] <= 'Z') {
            numberList.push_back(s[i] - 'A');
        }
    }
}

```



```

if (numberList.size() % 2)
    numberList.push_back(29);
// 步骤二：转换为码字
for (int i = 0; i < numberList.size(); i += 2) {
    codeWord.push_back(30 * numberList[i] + numberList[i + 1]);
}
// 步骤三：补充码字
while ((1 + codeWord.size()) % w != 0) {
    codeWord.push_back(900);
}
// 步骤四：输出结果
codeWord.insert(codeWord.begin(), codeWord.size() + 1);
for (int i = 0; i < codeWord.size(); ++i) {
    printf("%d\\n", codeWord[i]);
}
return 0;
}

```

## 2.4.2 100% 数据——模拟 + 多项式除法

### 2.4.2.1 思路

这部分数据要求我们对校验码进行处理，所以步骤变为：

1. 得到数字序列，注意不同模式的切换以及最后的补全。
2. 将得到的数字转换为码字。
3. 根据有效数据区每行能容纳的码字数  $w$ 、目前码字个数以及校验码的位数，在末尾补充码字。注意不要忽略长度码字。
4. 输出数据码部分结果。
5. 计算得出校验码，并输出。

校验码的位数能比较方便得出，关键在于校验码的计算。考虑关键公式：

$$x^k d(x) \equiv q(x)g(x) - r(x)$$

其中  $d(x)$  是  $n-1$  次多项式（已知）， $g(x)$  是  $k$  次多项式（已知），未知项有  $q(x), r(x)$ ，其中  $r(x)$  为所求。考虑消去  $q(x)$  的影响：可以在两端同时对  $g(x)$  取余，则  $q(x)g(x)$  项会被直接消去，可以化所求式为：

$$x^k d(x) \equiv -r(x) \pmod{g(x)}$$

所以目前问题转化为求解  $x^k d(x) \pmod{g(x)}$ 。

#### 定义 2.1 (多项式带余除法)

若  $f(x)$  和  $g(x)$  是两个多项式，且  $g(x)$  不等于 0，则存在唯一的多项式  $q(x)$  和  $r(x)$ ，满足：

$$f(x) = q(x)g(x) + r(x)$$

其中  $r(x)$  的次数小于  $g(x)$  的次数。此时  $q(x)$  称为  $g(x)$  除  $f(x)$  的商式， $r(x)$  称为余式。

**定义 2.2 (多项式长除法)**

求解多项式带余除法的一种方法，步骤如下：

1. 把被除式、除式按某个字母作降幂排列，并把所缺的项用零补齐；
2. 用被除式的第一项除以除式第一项，得到商式的第一项；
3. 用商式的第一项去乘除式，把积写在被除式下面（同类项对齐），消去相等项，把不相等的项结合起来；
4. 把减得的差当作新的被除式，再按照上面的方法继续演算，直到余式为零或余式的次数低于除式的次数时为止。

下面展示的是一个多项式长除法的例子：

$$\begin{array}{r}
 \phantom{x^2 - 12x + 27) } 6x^5 \phantom{+ 214x^6} + 286x^4 \phantom{+ 841x^5} + 4111x^3 \phantom{+ 821x^4} + 42431x^2 \phantom{+ 449x^3} + 398624x \phantom{+ 900x^2} + 3638751 \\
 \underline{x^2 - 12x + 27) \phantom{+ 214x^6} 6x^7 + 214x^6 + 841x^5 + 821x^4 + 449x^3 + 900x^2} \\
 \phantom{x^2 - 12x + 27) } - 6x^7 + 72x^6 - 162x^5 \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} 286x^6 + 679x^5 + 821x^4 \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \underline{- 286x^6 + 3432x^5 - 7722x^4} \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} 4111x^5 - 6901x^4 + 449x^3 \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \underline{- 4111x^5 + 49332x^4 - 110997x^3} \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} 42431x^4 - 110548x^3 + 900x^2 \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \underline{- 42431x^4 + 509172x^3 - 1145637x^2} \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \phantom{+ 509172x^3} 398624x^3 - 1144737x^2 \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \phantom{+ 509172x^3} \underline{- 398624x^3 + 4783488x^2 - 10762848x} \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \phantom{+ 509172x^3} \phantom{+ 4783488x^2} 3638751x^2 - 10762848x \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \phantom{+ 509172x^3} \phantom{+ 4783488x^2} \underline{- 3638751x^2 + 43665012x - 98246277} \\
 \phantom{x^2 - 12x + 27) } \phantom{+ 214x^6} \phantom{+ 679x^5} \phantom{+ 49332x^4} \phantom{+ 509172x^3} \phantom{+ 4783488x^2} \phantom{+ 43665012x} 32902164x - 98246277
 \end{array}$$

得到求解多项式带余除法的步骤后，考虑求解  $r(x)$  的步骤：

1. 计算  $g(x) = (x - 3)(x - 3^2) \cdots (x - 3^k)$ ；
2. 计算  $x^k d(x)$ ；
3. 计算  $x^k d(x) \bmod g(x)$ ，得到  $-r(x)$ ；
4. 对得到的每一项取反即可得到  $r(x)$ 。

计算  $g(x)$ ：考虑到每一次多项式乘以的因子都是  $(x - a)$  的格式，所以可以把  $A(x - a)$  的多项式相乘转化为  $xA - aA$  的格式。 $xA$  可以通过整体移项实现；在移项后，原本在  $x^i$  的系数成为  $x^{i+1}$  的系数，所以可以在一个数组上，从低位到高位依次计算，得到结果。

计算  $x^k d(x)$ ：这部分比较简单，将低  $k$  位的系数赋 0，再将已计算出的数据位放入对应位置即可。

计算  $x^k d(x) \bmod g(x)$ ：利用上文提到的多项式长除法即可。本题  $g(x)$  的最高位系数恒为 1，简化了计算。

**2.4.2.2 C++ 实现**

```

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

```

```

const int maxn = 210;
const int mod = 929;
int w, lev;
char s[100010];
int n;
vector<int> numberList;
vector<int> codeWord;
int verifyCodeLen; // 校验码长度
int currentMode;
void checkmode(char c) {
    /*
        检查将要输出的下个字符与目前模式是否匹配，
        若不匹配，则输出对应更改模式步骤。
    */
    if (c >= '0' && c <= '9') {
        if (currentMode != 2) {
            numberList.push_back(28);
            currentMode = 2;
        }
    } else if (c >= 'a' && c <= 'z') {
        if (currentMode != 1) {
            numberList.push_back(27);
            currentMode = 1;
        }
    } else if (c >= 'A' && c <= 'Z') {
        if (currentMode == 1) {
            numberList.push_back(28);
            numberList.push_back(28);
            currentMode = 0;
        }
        if (currentMode == 2) {
            numberList.push_back(28);
            currentMode = 0;
        }
    }
}
vector<int> get_gx(int k) {
    // 根据 k 计算 g(x)
    vector<int> res;
    res.push_back(mod - 3);
    res.push_back(1);
    int a0 = 3;
    for (int i = 2; i <= k; ++i) {
        a0 = (a0 * 3) % mod;
        res.insert(res.begin(), 0); // 在最低位插入 1，即整体次数 +1
        for (int j = 0; j < i; ++j) {
            res[j] = (res[j] - (a0 * res[j + 1]) % mod + mod) % mod;
        }
    }
}

```

```

    return res;
}

void get_verify_code() {
    // 计算校验码并输出
    vector<int> tmp;
    vector<int> g = get_gx(verifyCodeLen);
    // 初始化  $x^{-kd}(x)$ 
    for (int i = 1; i <= verifyCodeLen; ++i) {
        tmp.push_back(0);
    }
    for (int i = codeWord.size() - 1; i >= 0; --i) {
        tmp.push_back(codeWord[i]);
    }
    // 多项式长除法计算结果
    for (int i = tmp.size() - 1; i >= verifyCodeLen; --i) {
        int val = tmp[i];
        for (int j = 0; j < g.size(); ++j) {
            tmp[i - j] =
                (tmp[i - j] - (val * g[g.size() - 1 - j]) % mod + mod) % mod;
        }
    }
    // 将  $-r(x)$  转化为  $r(x)$ 
    for (int i = 0; i < verifyCodeLen; ++i) {
        // 注意: 不能直接  $\text{mod} - \text{tmp}[i]$ , 因为  $\text{tmp}[i]$  可能为 0
        tmp[i] = (mod - tmp[i]) % mod;
    }
    // 输出结果
    for (int i = verifyCodeLen - 1; i >= 0; --i) {
        printf("%d\n", tmp[i]);
    }
}

int main() {
    scanf("%d%d", &w, &lev);
    scanf("%s", s);
    n = strlen(s);
    // 步骤一: 得到数字序列
    currentMode = 0;
    for (int i = 0; i < n; ++i) {
        checkmode(s[i]);
        if (s[i] >= '0' && s[i] <= '9') {
            numberList.push_back(s[i] - '0');
        } else if (s[i] >= 'a' && s[i] <= 'z') {
            numberList.push_back(s[i] - 'a');
        } else if (s[i] >= 'A' && s[i] <= 'Z') {
            numberList.push_back(s[i] - 'A');
        }
    }
    if (numberList.size() % 2)

```

```
    numberList.push_back(29);
// 步骤二：转换为码字
for (int i = 0; i < numberList.size(); i += 2) {
    codeWord.push_back(30 * numberList[i] + numberList[i + 1]);
}
if (lev == -1)
    verifyCodeLen = 0;
else {
    verifyCodeLen = 1;
    for (int i = 0; i <= lev; ++i) {
        verifyCodeLen *= 2;
    }
}
// 步骤三：补充码字
while ((1 + verifyCodeLen + codeWord.size()) % w != 0) {
    codeWord.push_back(900);
}
codeWord.insert(codeWord.begin(), codeWord.size() + 1);
// 步骤四：输出数据码结果
for (int i = 0; i < codeWord.size(); ++i) {
    printf("%d\n", codeWord[i]);
}
// 步骤五：如果有校验码，则计算并输出
if (verifyCodeLen != 0) {
    get_verify_code();
}
return 0;
}
```

## 2.5 202112-4 磁盘文件操作

### 题目背景

小 C 对计算机运行的原理很感兴趣，经常进行一些研究和实验。

有一天，他在尝试删除一个好几 GB 大小的文件时，惊奇地发现删除操作几乎在一瞬间就完成了！这让他很是纳闷：如果计算机在每次删除文件时都直接在磁盘上把对应的数据抹掉，不是应该要花挺长时间吗？

于是他找来了小 S 和小 P 一起讨论。小 S 说，或许计算机是一个很“懒”的体系，在删除时不会真的去抹除数据吧？而小 P 则更见多识广一些，他当即找来了一个号称能“恢复磁盘数据”的软件，当场把小 C 刚刚删除的文件恢复了！

这让小 C 有了更强的好奇心，于是他们决定设计一个模型来模拟一个磁盘文件的写入、删除及恢复过程。但是在他们生活的西西艾弗岛上没有合适的条件来运行他们的模型，于是他们联系了带着一台算力超强的电脑来西西艾弗岛旅游的你来帮助他们。

### 题目描述

在小 C、小 S 和小 P 设计的模型中，计算机中有  $n$  段程序（编号为  $1 \sim n$ ），它们共享一块大小为  $m$  的磁盘空间（编号为  $1 \sim m$ ），磁盘上的每个位置可以写入一个整数。

最初，磁盘上每个位置上的数都是 0，并不被任何程序占用。

现在，这  $n$  段程序同时执行，在某一时刻，某段程序可能对磁盘数据进行读写等操作。

操作共  $k$  个，按时间先后顺序给出，具体操作如下：

- $0\ id\ l\ r\ x$ ：编号为  $id$  的程序尝试向磁盘空间中  $[l, r]$  位置上每个位置都写入一个整数  $x$ 。
  - 操作执行过程中，程序  $id$  会尝试从最左端  $l$  开始向右顺次写入数据。
  - 对于每个位置，若目前不被任何程序占用，则成功写入整数  $x$ ，并将其视为被程序  $id$  占用；
  - 若该位置目前正被程序  $id$  自己占用，则这次写入的  $x$  可以覆盖之前写入的结果，此后该位置仍被程序  $id$  占用；
  - 直到成功向  $r$  位置写入数据，或遇到第一个正在被其他程序占用的位置为止，此时该操作立刻中断。
- $1\ id\ l\ r$ ：程序  $id$  尝试删除磁盘中  $[l, r]$  位置上的所有数据。
  - 这一操作当且仅当  $[l, r]$  区间内所有位置都正在被程序  $id$  占用时才能成功执行。
  - 执行效果为将其中所有位置都解除占用，即恢复到可以被任意程序写入的状态。但为了便于恢复数据，不会立即将全部位置重新覆盖成 0。
  - 否则，认为此操作执行失败，不进行任何修改。
- $2\ id\ l\ r$ ：程序  $id$  尝试恢复磁盘中  $[l, r]$  位置上的所有数据。
  - 这一操作当且仅当  $[l, r]$  区间内所有位置都未被占用，且上一次被占用是被程序  $id$  占用时才能成功执行。
  - 执行效果为将其中所有位置恢复为被程序  $id$  占用的状态，同时由于之前删除操作并未改变其存储的值，因此本次操作也不需要改变每个位置上的值。
  - 否则，认为此操作执行失败，不进行任何修改。
- $3\ p$ ：尝试读取磁盘中  $p$  位置的数据，返回结果为两个整数。
  - 如果该位置当前正被程序  $id$  占用且存储的值为  $p$ ，返回结果为  $id\ p$ 。
  - 如果该位置当前没有被任何程序占用，返回 0 0。

你需要实现一个程序，帮助小 C、小 S 和小 P 来模拟实现上述过程，并对于每个操作输出操作结果。

### 输入格式

从标准输入读入数据。

第一行：3 个正整数  $n, m, k$ 。

接下来  $k$  行，每行若干个整数描述一个操作，格式如上所述。

## 输出格式

输出到标准输出。

输出共  $k$  行，对于每个操作输出一行。

对于每个写入操作，输出一个整数表示此次操作写入成功的最右位置；特别地如果该操作一个位置也没有写入成功，输出  $-1$ 。

对于每个删除、恢复操作，若该操作成功，输出一个字符串 OK，否则输出一个字符串 FAIL。

对于每个读取操作，输出两个整数表示此次查询的结果。

## 样例 #1

输入 #1:

```
3 15 12
0 1 1 5 -1
0 2 10 13 2
0 1 4 14 6
1 1 2 8
3 1
3 3
3 14
2 1 3 5
0 3 7 8 -4
2 1 6 8
1 3 6 7
0 2 5 7 3
```

输出 #1:

```
5
13
9
OK
1 -1
0 0
0 0
OK
8
FAIL
FAIL
-1
```

## 子任务

对于 25% 的数据， $n, k \leq 2000, m \leq 10000$ ；

对于另外 15% 的数据，没有删除、恢复操作；

对于另外 20% 的数据，没有恢复操作；

对于另外 15% 的数据， $n = 1$ 。

对于 100% 的数据， $1 \leq n, k \leq 2 \times 10^5, 1 \leq m \leq 10^9, 1 \leq id \leq n, 1 \leq l \leq r \leq m, 1 \leq p \leq m, |x| \leq 10^9$ 。

### 2.5.1 25% 数据——直接模拟

#### 2.5.1.1 思路

我们按照题目要求进行对应操作即可，注意每一个要求执行的条件：


- 写入操作：从左往右依次执行，直到第一个不被自己占用的位置。除了第一个点就被其他程序占用以外，必然会写入。遇到自己占用，则覆盖。

- 删除操作：同时整体进行，要求所有位置都被目前程序占用。要么全删，要么不做任何更改。
- 恢复操作：同时整体进行，要求所有位置都不被占用，且上次占用程序为目前程序。要么全恢复，要么不做任何更改。遇到自己占用，则不做任何更改。
- 读入操作：读取占用程序和数值，若未被占用，则输出 0 0。

## 2.5.2 100% 数据——离散化 + 线段树

### 2.5.2.1 思路

通过这道题的操作要求等，我们可以大致推测出这道题可能需要使用线段树。

 **提示** 如果没什么思路，可以拿各种数据结构往上套。例如本题，因为涉及区间修改、单点查询，对于树状数组来说负担太重，我们可以尝试其他数据结构。如果使用平衡树，则一般是要求出第 k 大数，或者是序列翻转类问题，对于本题来说不太契合。其他数据结构不再一一列举。综合考虑下，线段树是比较符合要求的。

#### 2.5.2.1.1 考虑线段树的做法 先不考虑线段树的内存空间问题，我们分析一下如何用线段树解决这道题目。

考虑我们需要维护的量，目前已知的有磁盘位置的值、目前占用程序 id、上次占用程序 id。

在这里，我们假设一个位置未被占用和被 id 为 0 的程序占用是等价的。

- 写入操作：可以划分为找到写入右边界和直接写入两个操作。  
直接写入操作就是直接的线段树区间修改，而划分操作需要知道该区间被占用的位置是否属于将要写入的 id。我们不妨将这个量设为 id1。
- 删除操作：可以划分为判断是否可删和直接删除两个操作。  
直接删除操作就是直接的线段树区间修改，而判断是否可删需要知道该区间所有的位置是否属于将要写入的 id。我们不妨将这个量设为 id2，注意 id1 与 id2 的区别——是否允许包含未被占用的程序。
- 恢复操作：可以划分为判断是否可恢复和直接恢复两个操作。  
该操作与删除操作类似，不过需要注意的是判断时需要判断目前占用的 id 和上次被占用的 id。
- 读取操作：可以划分为查询占用程序 id 和查询值两个操作。  
该操作是相对比较质朴的单点查询，当然也可以处理为区间。

通过以上分析，我们得到了需要维护的量：值、有关目前占用程序 id 的两个量、上次被占用的程序 id。我们考虑每个量针对父子之间的维护。

- 值 val：每个节点代表取值的多少，若左右子节点不同则设为一个不存在的值。因为我们是单点查询，所以不用担心查询到不存在的值的问题。
- 被占用位置程序 id1：
  - 若左右子节点都未被占用，则该节点标记为未占用；
  - 若左右子节点中存在不唯一节点，则该节点标记为不唯一。
  - 若左右子节点中一个节点未占用，则该节点标记为另一个非空节点的标记；
  - 若左右子节点都非空且相等，则该节点标记为任意一个节点；
  - 若左右子节点都非空且不等，则该节点标记为不唯一；
- 被占用位置程序 id2：为了方便进行讨论，将未被程序占用节点视为被 id 为 0 的程序占用。
  - 若左右子节点中存在不唯一节点，则该节点标记为不唯一。
  - 若左右子节点相等，则该节点标记为任意一个节点；
  - 若左右子节点不等，则该节点标记为不唯一；
- 上一次被占用程序 lid：与 id2 相同。
  - 若左右子节点中存在不唯一节点，则该节点标记为不唯一。
  - 若左右子节点相等，则该节点标记为任意一个节点；
  - 若左右子节点不等，则该节点标记为不唯一；



**2.5.2.1.2 解决空间问题** 理解线段树的解法之后，就会出现另一个问题：空间达到了  $1e9$  级别，肯定会 MLE。

我们可以从另一个角度考虑：一共有  $2 \times 10^5$  次询问，每次最多操作涉及一个区间，可以用两个端点表示。考虑到临界处的影响，一次操作最多会涉及 4 个点（比如原来的区间是  $[1, 10]$ ，我们更改了区间  $[3, 5]$ ，那么得到的区间为  $[1, 2], [3, 5], [6, 10]$ ，多出了 2, 3, 5, 6 四个点）。那么总体来看，涉及到的点最多有  $2 \times 10^5 \times 4 = 8 \times 10^5$  个。

我们可以维持这些点的相对大小关系，而将其投影到一个值域较小的区域，就可以减少空间占用了。这种方法称为离散化。

### 定义 2.3 (离散化)

把无限空间中有限的个体映射到有限的空间中去，以此提高算法的时空效率。通俗的说，离散化是在不改变数据相对大小的条件下，对数据进行相应的缩小。离散化本质上可以看成是一种哈希，其保证数据在哈希以后仍然保持原来的全/偏序关系。

离散化的步骤：

1. 统计所有出现过的数字，在知道确切上界的时候可以用数组，不清楚情况下可以用 vector；
2. 对所有的数据排序 (sort)、去重 (unique)；
3. 对于每个数，其离散化后的对应值即为其在排序去重后数组中的位置，可以通过二分 (lower\_bound) 确定。

这道题允许我们提前将所有可能出现的数记录下来（当然不是所有的题目都允许这样），所以这道题就解决了。线段树节点的个数与询问个数成比例，时间复杂度  $O(k \log k)$ 。

## 2.5.2.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
const int maxn = 200010;
const int INF = 1e9 + 10;
int n, m, k;
#define ls o << 1
#define rs ls | 1
struct treenode {
    // 当前节点的值，若不唯一则为 INF; lazy 为 INF 表示无延迟更新
    int val, lazy_val;
    // 当前占用 id，若存在除 0 以外两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id1, lazy_id1;
    // 当前占用 id，若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id2, lazy_id2;
    // 上次占用 id，若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int lid, lazy_lid;
} tree[maxn << 5];

void pushup(int o) {
```

```

// 线段树上传操作，合并左右子树结果
// val 的合并
tree[o].val = (tree[ls].val == tree[rs].val) ? tree[ls].val : INF;
// id1 的合并
if (tree[ls].id1 == -1 || tree[rs].id1 == -1) {
    tree[o].id1 = -1;
} else if (tree[ls].id1 == tree[rs].id1) {
    tree[o].id1 = tree[ls].id1;
} else if (tree[ls].id1 == 0) {
    tree[o].id1 = tree[rs].id1;
} else if (tree[rs].id1 == 0) {
    tree[o].id1 = tree[ls].id1;
} else {
    tree[o].id1 = -1;
}
// id2 的合并
if (tree[ls].id2 == -1 || tree[rs].id2 == -1) {
    tree[o].id2 = -1;
} else if (tree[ls].id2 == tree[rs].id2) {
    tree[o].id2 = tree[ls].id2;
} else {
    tree[o].id2 = -1;
}
// lid 的合并
if (tree[ls].lid == -1 || tree[rs].lid == -1) {
    tree[o].lid = -1;
} else if (tree[ls].lid == tree[rs].lid) {
    tree[o].lid = tree[ls].lid;
} else {
    tree[o].lid = -1;
}
}

void pushdown(int o) {
    // 线段树标记下传操作
    if (tree[o].lazy_val != INF) {
        tree[ls].val = tree[rs].val = tree[o].lazy_val;
        tree[ls].lazy_val = tree[rs].lazy_val = tree[o].lazy_val;
        tree[o].lazy_val = INF;
    }
    if (tree[o].lazy_id1 != -1) {
        tree[ls].id1 = tree[rs].id1 = tree[o].lazy_id1;
        tree[ls].lazy_id1 = tree[rs].lazy_id1 = tree[o].lazy_id1;
        tree[o].lazy_id1 = -1;
    }
    if (tree[o].lazy_id2 != -1) {
        tree[ls].id2 = tree[rs].id2 = tree[o].lazy_id2;
        tree[ls].lazy_id2 = tree[rs].lazy_id2 = tree[o].lazy_id2;
        tree[o].lazy_id2 = -1;
    }
}

```

```

    }
    if (tree[o].lazy_lid != -1) {
        tree[ls].lid = tree[rs].lid = tree[o].lazy_lid;
        tree[ls].lazy_lid = tree[rs].lazy_lid = tree[o].lazy_lid;
        tree[o].lazy_lid = -1;
    }
}

void build(int o, int l, int r) {
    // 线段树初始化建树
    if (l == r) {
        tree[o].val = 0;
        tree[o].lazy_val = INF;
        tree[o].id1 = tree[o].id2 = tree[o].lid = 0;
        tree[o].lazy_id1 = tree[o].lazy_id2 = tree[o].lazy_lid = -1;
        return;
    }
    int mid = (l + r) >> 1;
    build(ls, l, mid);
    build(rs, mid + 1, r);
    tree[o].lazy_val = INF;
    pushup(o);
}

#define ALLOK -2
int find_right(int o, int l, int r, int ql, int qid) {
    // 操作一中，固定左端点，寻找右端点可能的最大值
    // 这里没有考虑和右端点的比较，直接寻找了最大的可能值
    pushdown(o);
    if (r < ql || tree[o].id1 == qid || tree[o].id1 == 0) {
        // 全部符合条件
        return ALLOK;
    } else if (tree[o].id2 != -1) {
        // 不符合条件，返回该区域左边第一个
        return l - 1;
    } else {
        // 需要寻找确切位置
        // 先查找左区间，如果左区间全满足则再寻找右区间
        int mid = (l + r) >> 1;
        int leftPart = (ql <= mid) ? find_right(ls, l, mid, ql, qid) : ALLOK;
        return (leftPart == ALLOK) ? find_right(rs, mid + 1, r, ql, qid)
            : leftPart;
    }
}

#undef ALLOK

void modify_val(int o, int l, int r, int ql, int qr, int val, int id,
                bool ignoreLid = false) {
    // 若 val = INF 代表不需要对 val 进行处理

```

```

// 若 ignoreLid = true 则不对 lid 进行更改
if (l >= ql && r <= qr) {
    if (val != INF)
        tree[o].val = tree[o].lazy_val = val;
    tree[o].id1 = tree[o].lazy_id1 = id;
    tree[o].id2 = tree[o].lazy_id2 = id;
    if (!ignoreLid)
        tree[o].lid = tree[o].lazy_lid = id;
    return;
}
pushdown(o);
int mid = (l + r) >> 1;
if (ql <= mid) {
    modify_val(ls, l, mid, ql, qr, val, id, ignoreLid);
}
if (qr > mid) {
    modify_val(rs, mid + 1, r, ql, qr, val, id, ignoreLid);
}
pushup(o);
}

bool is_same_id(int o, int l, int r, int ql, int qr, int id,
                bool isRecover = false) {
    // 判断该区域 id 和 lid 是否满足条件
    if (l >= ql && r <= qr) {
        if (isRecover) {
            // 检查 id = 0 且 lid = id
            return (tree[o].id2 == 0 && tree[o].lid == id);
        } else {
            // 检查 id = id
            return (tree[o].id2 == id);
        }
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    bool isSame = true;
    if (ql <= mid) {
        isSame = isSame && is_same_id(ls, l, mid, ql, qr, id, isRecover);
    }
    if (qr > mid && isSame) {
        isSame = isSame && is_same_id(rs, mid + 1, r, ql, qr, id, isRecover);
    }
    return isSame;
}

int query_val(int o, int l, int r, int p) {
    // 线段树单点求值: val
    if (p >= l && p <= r && tree[o].val != INF) {
        return tree[o].val;
    }
}

```

```

}
pushdown(o);
int mid = (l + r) >> 1;
if (p <= mid)
    return query_val(ls, l, mid, p);
else
    return query_val(rs, mid + 1, r, p);
}

int query_id(int o, int l, int r, int p) {
    // 线段树单点求值: id2
    if (p >= l && p <= r && tree[o].id2 != -1) {
        return tree[o].id2;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_id(ls, l, mid, p);
    else
        return query_id(rs, mid + 1, r, p);
}

#undef ls
#undef rs

struct instruction {
    int opt, id, l, r, x;
} inst[maxn];
// numList 存储所有可能出现的数, totnum 表示个数
int numList[maxn << 2], totnum;
void discretization() {
    // 离散化操作
    sort(numList + 1, numList + 1 + totnum);
    totnum = unique(numList + 1, numList + 1 + totnum) - numList - 1;
    m = totnum;
    for (int i = 1; i <= k; ++i) {
        if (inst[i].opt == 0 || inst[i].opt == 1 || inst[i].opt == 2) {
            inst[i].l =
                lower_bound(numList + 1, numList + 1 + totnum, inst[i].l) -
                numList;
            inst[i].r =
                lower_bound(numList + 1, numList + 1 + totnum, inst[i].r) -
                numList;
        } else {
            inst[i].x =
                lower_bound(numList + 1, numList + 1 + totnum, inst[i].x) -
                numList;
        }
    }
}

```

```

}

int main() {
    scanf("%d%d%d", &n, &m, &k);
    numList[++totnum] = 1;
    numList[++totnum] = m;
    for (int i = 1; i <= k; ++i) {
        scanf("%d", &inst[i].opt);
        if (inst[i].opt == 0) {
            scanf("%d%d%d", &inst[i].id, &inst[i].l, &inst[i].r, &inst[i].x);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            // 注意边界问题，为了方便这里把交界处两点分开了，下同
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else if (inst[i].opt == 1) {
            scanf("%d%d", &inst[i].id, &inst[i].l, &inst[i].r);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else if (inst[i].opt == 2) {
            scanf("%d%d", &inst[i].id, &inst[i].l, &inst[i].r);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else {
            scanf("%d", &inst[i].x);
            // 对于查询的数，不需要进行离散化，查找第一个比它大的数即可
        }
    }

    // 离散化处理
    discretization();

    // 线段树初始化建树
    build(1, 1, m);

    // 进行操作
    for (int i = 1; i <= k; ++i) {
        if (inst[i].opt == 0) {
            // 写入操作：先求得范围，再进行填充

```

```

    int r = find_right(1, 1, m, inst[i].l, inst[i].id);
    if (r == -2)
        // r = -2 代表全部满足
        r = inst[i].r;
    else
        r = min(r, inst[i].r);
    if (inst[i].l <= r) {
        printf("%d\n", numList[r]); // 注意返回离散化前的值
        modify_val(1, 1, m, inst[i].l, r, inst[i].x, inst[i].id);
    } else {
        printf("-1\n");
    }
} else if (inst[i].opt == 1) {
    // 删除操作：先判断是否可行，之后执行
    if (is_same_id(1, 1, m, inst[i].l, inst[i].r, inst[i].id)) {
        printf("OK\n");
        modify_val(1, 1, m, inst[i].l, inst[i].r, INF, 0, true);
    } else {
        printf("FAIL\n");
    }
} else if (inst[i].opt == 2) {
    // 恢复操作：先判断是否可行，之后执行
    if (is_same_id(1, 1, m, inst[i].l, inst[i].r, inst[i].id, true)) {
        printf("OK\n");
        modify_val(1, 1, m, inst[i].l, inst[i].r, INF, inst[i].id,
                    true);
    } else {
        printf("FAIL\n");
    }
} else if (inst[i].opt == 3) {
    // 读取操作：分别读取 id 和 val
    int id = query_id(1, 1, m, inst[i].x);
    int val = query_val(1, 1, m, inst[i].x);
    if (id == 0) {
        printf("0_0\n");
    } else {
        printf("%d_%d\n", id, val);
    }
}
}
return 0;
}

```

## 2.5.3 100% 数据——动态开点线段树

### 2.5.3.1 思路

在上一题的做法中，我们需要先读入所有的数据并进行离散化处理，之后再执行主要的算法过程。但不是所有的题目都可以在执行主要的算法过程前得到所有的输入数据。

**定义 2.4 (离线算法)**

要求在执行算法前输入数据已知的算法称为离线算法。一般而言，如果没有对输入输出做特殊处理，则可以用离线算法解决该问题。

**定义 2.5 (在线算法)**

不需要输入数据已知就可以执行的算法称为在线算法。一般而言，如果对输入输出做特殊处理（如本次的询问需要与上次执行的答案进行异或才能得到真正的询问），则只能用离线算法解决该问题。

对于一道能用离线和在线算法解决的题目，如果出题人对数据进行了加密处理，导致只能使用在线算法，则我们称这道题是**强制在线**的。

离散化需要事先知道所有可能出现的数，所以是**离线算法**。如果要强制在线，就需要另一种思路。

同样，从询问涉及的点有限出发，我们考虑最多能涉及线段树上点的个数。线段树的高度为  $O(\log m)$ ，假设每个涉及查询的点都到达了线段树的叶子结点，且不考虑根到任意两个结点之间重复的结点，则总共涉及的线段树节点数的个数为  $O(k \log m)$ 。所以我们只需要为用到的节点开辟空间即可。

针对一般的线段树，我们是预先建好了整棵线段树（build 函数），每个线段树节点的左右子节点编号与其本身编号都是对应的（通常一个子节点是父结点的二倍，而另一个子节点则相差 1）。而对于这种只为需要用到节点开辟空间的线段树，其左右子树只有在需要的时候才会被创建，所以编号间没有特定关系，需要记录。

考虑什么时候需要开辟新结点：在初始化的时候需要开创一个根结点；在进行修改及查询的时候，如果区间不是所要的区间，则需要开创新的节点。有一个技巧是，在修改和查询的时候往往要下传标记（pushdown），可以在此之前检查是否需要开创节点。

**2.5.3.2 C++ 实现**

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
const int maxn = 200010;
const int INF = 1e9 + 10;
int n, m, k;
struct treenode {
    // 左右子节点编号
    int lc, rc;
    // 当前节点的值，若不唯一则为 INF; lazy 为 INF 表示无延迟更新
    int val, lazy_val;
    // 当前占用 id，若存在除 0 以外两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id1, lazy_id1;
    // 当前占用 id，若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id2, lazy_id2;
    // 上次占用 id，若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int lid, lazy_lid;
} tree[maxn << 5];
int cnt; // 线段树节点个数
```



```

#define ls tree[o].lc
#define rs tree[o].rc
int insert_node() {
    // 向线段树中插入一个节点
    ++cnt;
    tree[cnt].lc = tree[cnt].rc = 0;
    tree[cnt].val = 0;
    tree[cnt].id1 = tree[cnt].id2 = 0;
    tree[cnt].lid = 0;
    tree[cnt].lazy_val = INF;
    tree[cnt].lazy_id1 = tree[cnt].lazy_id2 = -1;
    tree[cnt].lid = -1;
    return cnt;
}

void pushup(int o) {
    // 线段树上传操作，合并左右子树结果
    // val 的合并
    tree[o].val = (tree[ls].val == tree[rs].val) ? tree[ls].val : INF;
    // id1 的合并
    if (tree[ls].id1 == -1 || tree[rs].id1 == -1) {
        tree[o].id1 = -1;
    } else if (tree[ls].id1 == tree[rs].id1) {
        tree[o].id1 = tree[ls].id1;
    } else if (tree[ls].id1 == 0) {
        tree[o].id1 = tree[rs].id1;
    } else if (tree[rs].id1 == 0) {
        tree[o].id1 = tree[ls].id1;
    } else {
        tree[o].id1 = -1;
    }
    // id2 的合并
    if (tree[ls].id2 == -1 || tree[rs].id2 == -1) {
        tree[o].id2 = -1;
    } else if (tree[ls].id2 == tree[rs].id2) {
        tree[o].id2 = tree[ls].id2;
    } else {
        tree[o].id2 = -1;
    }
    // lid 的合并
    if (tree[ls].lid == -1 || tree[rs].lid == -1) {
        tree[o].lid = -1;
    } else if (tree[ls].lid == tree[rs].lid) {
        tree[o].lid = tree[ls].lid;
    } else {
        tree[o].lid = -1;
    }
}

```

```

void pushdown(int o) {
    // 线段树标记下传操作
    // 如果对应点未被创建, 则进行创建
    if (!ls)
        ls = insert_node();
    if (!rs)
        rs = insert_node();
    if (tree[o].lazy_val != INF) {
        tree[ls].val = tree[rs].val = tree[o].lazy_val;
        tree[ls].lazy_val = tree[rs].lazy_val = tree[o].lazy_val;
        tree[o].lazy_val = INF;
    }
    if (tree[o].lazy_id1 != -1) {
        tree[ls].id1 = tree[rs].id1 = tree[o].lazy_id1;
        tree[ls].lazy_id1 = tree[rs].lazy_id1 = tree[o].lazy_id1;
        tree[o].lazy_id1 = -1;
    }
    if (tree[o].lazy_id2 != -1) {
        tree[ls].id2 = tree[rs].id2 = tree[o].lazy_id2;
        tree[ls].lazy_id2 = tree[rs].lazy_id2 = tree[o].lazy_id2;
        tree[o].lazy_id2 = -1;
    }
    if (tree[o].lazy_lid != -1) {
        tree[ls].lid = tree[rs].lid = tree[o].lazy_lid;
        tree[ls].lazy_lid = tree[rs].lazy_lid = tree[o].lazy_lid;
        tree[o].lazy_lid = -1;
    }
}

#define ALLOK -2
int find_right(int o, int l, int r, int ql, int qid) {
    // 操作一中, 固定左端点, 寻找右端点可能的最大值
    // 这里没有考虑和右端点的比较, 直接寻找了最大的可能值
    pushdown(o);
    if (r < ql || tree[o].id1 == qid || tree[o].id1 == 0) {
        // 全部符合条件
        return ALLOK;
    } else if (tree[o].id2 != -1) {
        // 不符合条件, 返回该区域左边第一个
        return l - 1;
    } else {
        // 需要寻找确切位置
        // 先查找左区间, 如果左区间全满足则再寻找右区间
        int mid = (l + r) >> 1;
        int leftPart = (ql <= mid) ? find_right(ls, l, mid, ql, qid) : ALLOK;
        return (leftPart == ALLOK) ? find_right(rs, mid + 1, r, ql, qid)
            : leftPart;
    }
}

```

```

#undef ALLOC

void modify_val(int o, int l, int r, int ql, int qr, int val, int id,
               bool ignoreLid = false) {
    // 若 val = INF 代表不需要对 val 进行处理
    // 若 ignoreLid = true 则不对 lid 进行更改
    if (l >= ql && r <= qr) {
        if (val != INF)
            tree[o].val = tree[o].lazy_val = val;
        tree[o].id1 = tree[o].lazy_id1 = id;
        tree[o].id2 = tree[o].lazy_id2 = id;
        if (!ignoreLid)
            tree[o].lid = tree[o].lazy_lid = id;
        return;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (ql <= mid) {
        modify_val(ls, l, mid, ql, qr, val, id, ignoreLid);
    }
    if (qr > mid) {
        modify_val(rs, mid + 1, r, ql, qr, val, id, ignoreLid);
    }
    pushup(o);
}

bool is_same_id(int o, int l, int r, int ql, int qr, int id,
               bool isRecover = false) {
    // 判断该区域 id 和 lid 是否满足条件
    if (l >= ql && r <= qr) {
        if (isRecover) {
            // 检查 id = 0 且 lid = id
            return (tree[o].id2 == 0 && tree[o].lid == id);
        } else {
            // 检查 id = id
            return (tree[o].id2 == id);
        }
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    bool isSame = true;
    if (ql <= mid) {
        isSame = isSame && is_same_id(ls, l, mid, ql, qr, id, isRecover);
    }
    if (qr > mid && isSame) {
        isSame = isSame && is_same_id(rs, mid + 1, r, ql, qr, id, isRecover);
    }
    return isSame;
}

```

```

int query_val(int o, int l, int r, int p) {
    // 线段树单点求值: val
    if (p >= l && p <= r && tree[o].val != INF) {
        return tree[o].val;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_val(ls, l, mid, p);
    else
        return query_val(rs, mid + 1, r, p);
}

int query_id(int o, int l, int r, int p) {
    // 线段树单点求值: id2
    if (p >= l && p <= r && tree[o].id2 != -1) {
        return tree[o].id2;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_id(ls, l, mid, p);
    else
        return query_id(rs, mid + 1, r, p);
}

#undef ls
#undef rs

int main() {
    scanf("%d%d%d", &n, &m, &k);
    // 创建根节点
    insert_node();
    // 进行操作
    int r_opt, r_id, r_l, r_r, r_x, r_p;
    while (k--) {
        scanf("%d", &r_opt);
        if (r_opt == 0) {
            // 写入
            scanf("%d%d%d%d", &r_id, &r_l, &r_r, &r_x);
            int r = find_right(1, 1, m, r_l, r_id);
            if (r == -2)
                r = r_r;
            else
                r = min(r, r_r);
            if (r_l <= r) {
                printf("%d\n", r);
                modify_val(1, 1, m, r_l, r, r_x, r_id);
            }
        }
    }
}

```

```

    } else {
        printf("-1\n");
    }
} else if (r_opt == 1) {
    // 删除
    scanf("%d%d%d", &r_id, &r_l, &r_r);
    if (is_same_id(1, 1, m, r_l, r_r, r_id)) {
        printf("OK\n");
        modify_val(1, 1, m, r_l, r_r, INF, 0, true);
    } else {
        printf("FAIL\n");
    }
} else if (r_opt == 2) {
    // 恢复
    scanf("%d%d%d", &r_id, &r_l, &r_r);
    if (is_same_id(1, 1, m, r_l, r_r, r_id, true)) {
        printf("OK\n");
        modify_val(1, 1, m, r_l, r_r, INF, r_id, true);
    } else {
        printf("FAIL\n");
    }
} else {
    // 查询
    scanf("%d", &r_p);
    int id = query_id(1, 1, m, r_p);
    int val = query_val(1, 1, m, r_p);
    if (id == 0) {
        printf("0_0\n");
    } else {
        printf("%d_%d\n", id, val);
    }
}
}
return 0;
}

```

## 2.6 202112-5 极差路径

### 题目背景

众所周知，西西艾弗岛是一个旅游胜地，但是由于兴建机场，西西艾弗岛最近的财务状况有点紧张。

### 题目描述

为了从游客手中获取更多的经济利润，岛上仅有的三个小学生小 C、小 S 和小 P 建立了  $n$  个景点，编号依次从 1 到  $n$ 。编号为  $i$  的景点是第  $i$  个被修建的。由于越到后期经费越是不足，所以编号更大的景点通常更令人不满意——方便起见，假定编号为  $i$  的景点的令人不满意程度是  $i$ 。

有些景点之间修有双向可通行的道路，但是出于减少经费的考虑，他们只修了能使得所有景点连通的最少数量的道路，从而这些景点和其间的道路形成一棵树的结构。

对于每个游客而言，由于只修了  $n-1$  条道路，所以他只能沿着树上的边参观，并且由于他不可能重复参观一个景点，所以他的游览路径一定是树上的一条简单路径。

现在西西艾弗岛希望制定一些推荐游览路径，但并非所有树上的路径都是合意的，因为这条路径上的景点令人不满意程度的极差可能过大，使游客产生这些景点质量不稳定的错觉。由于最开始的景点和最后的景点令人印象比较深刻，所以游客通常会把游览路径上的景点和这两个景点作比较。因此，最令人不满意的景点不能比这两个景点差太多，最优秀的景点也不能比这两个景点优秀太多。

具体来说，一条从  $x$  到  $y$  的游览路径（记作  $(x, y)$ ）是推荐的，当且仅当下式成立：

$$\min\{x, y - k_1\} \leq \min\{P(x, y)\} \leq \max\{P(x, y)\} \leq \max\{x, y + k_2\}$$

其中  $P(x, y)$  表示一条从  $x$  出发到达  $y$  的简单路径上的点的令人不满意程度的数值的集合（包括  $x$  和  $y$ ，也就是  $x$  到  $y$  的简单路径上的点的编号的集合）， $\min S$  和  $\max S$  分别表示集合  $S$  中的最小和最大值， $k_1, k_2$  是西西艾弗岛经过数次尝试后选取的两个给定的参数，保证  $k_1, k_2 > 0$ 。

特别的，容易验证  $x = y$  时  $(x, y)$  总是推荐的。

现在西西艾弗岛想知道，一共有多少树上的简单路径作为游览路径是被推荐的？这里我们认为  $(x, y)$  和  $(y, x)$  是同一条路径。

### 输入格式

从标准输入读入数据。

第一行输入三个非负整数  $n, k_1, k_2$ 。

接下来  $n-1$  行，每行两个正整数  $x, y$  表示树上的一条边。

### 输出格式

输出到标准输出。

输出一行一个非负整数表示答案。

### 样例 #1

输入 #1:

输出 #1:

```

5 0 1
5 4
4 2
4 1
2 3

```

12

解释 #1:

容易验证  $(1, 1), (1, 4), (1, 5), (1, 3), (2, 2), (2, 4), (2, 5), (2, 3), (3, 3), (4, 4), (4, 5), (5, 5)$  都是推荐的游览路径, 因此答案是 12。

### 子任务

测试点	$n \leq$	$k_1$	$k_2$	树的形态	堆性质
1	5000	$\leq n$	$\leq n$	$A_3$	无
2	5000	$\leq n$	$\leq n$	$A_3$	无
3	5000	$\leq n$	$\leq n$	$A_3$	无
4	$5 \times 10^5$	$= 0$	$= 0$	$A_1$	有
5	$5 \times 10^5$	$= 0$	$= 0$	$A_1$	无
6	$5 \times 10^5$	$\leq n$	$= 0$	$A_1$	有
7	$5 \times 10^5$	$\leq n$	$= 0$	$A_1$	无
8	$5 \times 10^5$	$\leq n$	$\leq n$	$A_1$	有
9	$5 \times 10^5$	$\leq n$	$\leq n$	$A_1$	无
10	$5 \times 10^5$	$= 0$	$= 0$	$A_2$	无
11	$5 \times 10^5$	$\leq n$	$= 0$	$A_2$	无
12	$5 \times 10^5$	$\leq n$	$\leq n$	$A_2$	无
13	$5 \times 10^5$	$= 0$	$= 0$	$A_3$	有
14	$5 \times 10^5$	$= 0$	$= 0$	$A_3$	无
15	$5 \times 10^5$	$= 0$	$= 0$	$A_3$	无
16	$5 \times 10^5$	$= 0$	$= 0$	$A_3$	无
17	$5 \times 10^5$	$\leq n$	$= 0$	$A_3$	有
18	$5 \times 10^5$	$\leq n$	$= 0$	$A_3$	无
19	$5 \times 10^5$	$\leq n$	$= 0$	$A_3$	无
20	$5 \times 10^5$	$\leq n$	$= 0$	$A_3$	无
21	$5 \times 10^5$	$\leq n$	$\leq n$	$A_3$	有
22	$5 \times 10^5$	$\leq n$	$\leq n$	$A_3$	有
23	$5 \times 10^5$	$\leq n$	$\leq n$	$A_3$	无
24	$5 \times 10^5$	$\leq n$	$\leq n$	$A_3$	无
25	$5 \times 10^5$	$\leq n$	$\leq n$	$A_3$	无

对于 100% 的数据,  $1 \leq n \leq 5 \times 10^5, 0 \leq k_1, k_2 \leq n$ , 保证输入的  $n - 1$  条边一定构成一棵树。  
树的形态:

- $A_1$ : 树是一条链;
- $A_2$ : 存在一个度数为  $n-1$  的点;
- $A_3$ : 树的形态无特殊性质。

堆性质: 若取编号为 1 的点为根, 则除 1 号点外, 每个点的编号都比其父节点的编号大。

### 提示

请注意答案可能的取值范围。



## 第 3 章 第 23 次认证（2021 年 09 月）

### 3.1 题目及设计知识点

题目编号	题目名称	知识点
202109-1	数组推导	模拟
202109-2	非零段划分	模拟，数学
202109-3	脉冲神经网络	模拟
202109-4	收集卡牌	状压 dp
202109-5	箱根山岳险天下	树链剖分，动态树

## 3.2 202109-1 数组推导

### 题目描述

$A_1, A_2, \dots, A_n$  是一个由  $n$  个自然数（即非负整数）组成的数组。在此基础上，我们用数组  $B_1 \dots B_n$  表示  $A$  的前缀最大值。

$$B_i = \max \{A_1, A_2, \dots, A_i\}$$

如上所示， $B_i$  定义为数组  $A$  中前  $i$  个数的最大值。根据该定义易知  $A_1 = B_1$ ，且随着  $i$  的增大， $B_i$  单调不降。此外，我们用  $sum = A_1 + A_2 + \dots + A_n$  表示数组  $A$  中  $n$  个数的总和。

现已知数组  $B$ ，我们想要根据  $B$  的值来反推数组  $A$ 。显然，对于给定的  $B$ ， $A$  的取值可能并不唯一。试计算，在数组  $A$  所有可能的取值情况中， $sum$  的最大值和最小值分别是多少？

### 输入格式

从标准输入读入数据。

输入的第一行包含一个正整数  $n$ 。

输入的第二行包含  $n$  个用空格分隔的自然数  $B_1, B_2, \dots, B_n$ 。

### 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，表示  $sum$  的最大值。

第二行输出一个整数，表示  $sum$  的最小值。

### 样例 #1

输入 #1:

```
6
0 0 5 5 10 10
```

输出 #1:

```
30
15
```

解释 #1:

数组  $A$  的可能取值包括但不限于以下三种情况。

- 情况一:  $A = [0, 0, 5, 5, 10, 10]$
- 情况二:  $A = [0, 0, 5, 3, 10, 4]$
- 情况三:  $A = [0, 0, 5, 0, 10, 0]$

其中第一种情况  $sum = 30$  为最大值，第三种情况  $sum = 15$  为最小值。

## 样例 #2

输入 #2:

输出 #2:

```
7
10 20 30 40 50 60 75
```

```
285
285
```

解释 #2:

$A = [10, 20, 30, 40, 50, 60, 75]$  是唯一可能的取值, 所以  $sum$  的最大、最小值均为 285。

## 子任务

50% 的测试数据满足数组  $B$  单调递增, 即  $0 < B_1 < B_2 < \cdots < B_n < 10^5$ ;

全部的测试数据满足  $n \leq 100$  且数组  $B$  单调不降, 即  $0 \leq B_1 \leq B_2 \leq \cdots \leq B_n \leq 10^5$ 。

### 3.3 202109-2 非零段划分

#### 题目描述

$A_1, A_2, \dots, A_n$  是一个由  $n$  个自然数（非负整数）组成的数组。我们称其中  $A_i, \dots, A_j$  是一个非零段，当且仅当以下条件同时满足：

- $1 \leq i \leq j \leq n$ ;
- 对于任意的整数  $k$ ，若  $i \leq k \leq j$ ，则  $A_k > 0$ ;
- $i = 1$  或  $A_{i-1} = 0$ ;
- $j = n$  或  $A_{j+1} = 0$ 。

下面展示了几个简单的例子：

- $A = [3, 1, 2, 0, 0, 2, 0, 4, 5, 0, 2]$  中的 4 个非零段依次为  $[3, 1, 2]$ 、 $[2]$ 、 $[4, 5]$  和  $[2]$ ;
- $A = [2, 3, 1, 4, 5]$  仅有 1 个非零段;
- $A = [0, 0, 0]$  则不含非零段（即非零段个数为 0）。

现在我们可以对数组  $A$  进行如下操作：任选一个正整数  $p$ ，然后将  $A$  中所有小于  $p$  的数都变为 0。试选取一个合适的  $p$ ，使得数组  $A$  中的非零段个数达到最大。若输入的  $A$  所含非零段数已达最大值，可取  $p = 1$ ，即不对  $A$  做任何修改。

#### 输入格式

从标准输入读入数据。

输入的第一行包含一个正整数  $n$ 。

输入的第二行包含  $n$  个用空格分隔的自然数  $A_1, A_2, \dots, A_n$ 。

#### 输出格式

输出到标准输出。

仅输出一个整数，表示对数组  $A$  进行操作后，其非零段个数能达到的最大值。

#### 样例 #1

输入 #1:

输出 #1:

```
11
3 1 2 0 0 2 0 4 5 0 2
```

```
5
```

解释 #1:

$p = 2$  时， $A = [3, 0, 2, 0, 0, 2, 0, 4, 5, 0, 2]$ ，5 个非零段依次为  $[3]$ 、 $[2]$ 、 $[2]$ 、 $[4, 5]$  和  $[2]$ ；此时非零段个数达到最大。

## 样例 #2

输入 #2:

输出 #2:

```
14
5 1 20 10 10 10 10 15 10 20 1 5 10 15
```

4

解释 #2:

$p = 12$  时,  $A = [0, 0, 20, 0, 0, 0, 0, 15, 0, 20, 0, 0, 0, 15]$ , 4 个非零段依次为  $[20]$ 、 $[15]$ 、 $[20]$  和  $[15]$ ; 此时非零段个数达到最大。

## 样例 #3

输入 #3:

输出 #3:

```
3
1 0 0
```

1

解释 #3:

$p = 1$  时,  $A = [1, 0, 0]$ , 此时仅有 1 个非零段  $[1]$ , 非零段个数达到最大。

## 样例 #4

输入 #4:

输出 #4:

```
3
0 0 0
```

0

解释 #4:

无论  $p$  取何值,  $A$  都不含有非零段, 故非零段个数至多为 0。

## 子任务

70% 的测试数据满足  $n \leq 1000$ ;

全部的测试数据满足  $n \leq 5 \times 10^5$ , 且数组  $A$  中的每一个数均不超过  $10^4$ 。

### 3.4 202109-3 脉冲神经网络

#### 题目背景

在本题中，你需要实现一个 SNN（spiking neural network，脉冲神经网络）的模拟器。一个 SNN 由以下几部分组成：

1. 神经元：按照一定的公式更新内部状态，接受脉冲并可以发放脉冲
2. 脉冲源：在特定的时间发放脉冲
3. 突触：连接神经元-神经元或者脉冲源-神经元，负责传递脉冲

#### 题目描述

神经元会按照一定的规则更新自己的内部状态。本题中，我们对时间进行离散化处理，即设置一个时间间隔  $\Delta t$ ，仅考虑时间间隔整数倍的时刻  $t = k\Delta t (k \in \mathbb{Z}^+)$ ，按照下面的公式，从  $k-1$  时刻的取值计算  $k$  时刻的变量的取值：

$$v_k = v_{k-1} + \Delta t(0.04v_{k-1}^2 + 5v_{k-1} + 140 - u_{k-1}) + I_k$$

$$u_k = u_{k-1} + \Delta t a(bv_{k-1} - u_{k-1})$$

其中  $v$  和  $u$  是神经元内部的变量，会随时间而变化， $a$  和  $b$  是常量，不会随时间变化；其中  $I_k$  表示该神经元在  $k$  时刻接受到的所有脉冲输入的强度之和，如果没有接受到脉冲，那么  $I_k = 0$ 。当进行上面的计算后，如果满足  $v_k \geq 30$ ，神经元会发放一个脉冲，脉冲经过突触传播到其他神经元；同时， $v_k$  设为  $c$  并且  $u_k$  设为  $u_k + d$ ，其中  $c$  和  $d$  也是常量。图 1 展示了一个神经元  $v$  变量随时间变化的曲线。

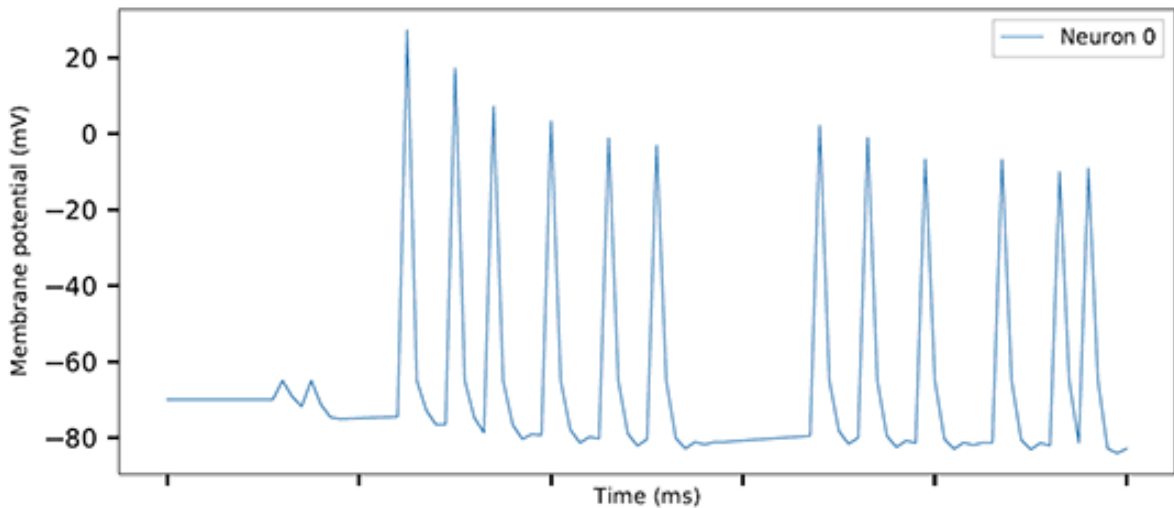


图 3.1: 神经元  $v$  变量随时间变化的曲线

突触表示的是神经元-神经元、脉冲源-神经元的连接关系，包含一个入结点和一个出结点（可能出现自环和重边）。当突触的入结点（神经元或者脉冲源）在  $k$  时刻发放一个脉冲，那么在传播延迟  $D (D > 0)$  个时刻以后，也就是在  $k + D$  时刻突触的出结点（神经元）会接受到一个强度为  $w$  的脉冲。

脉冲源在每个时刻以一定的概率发放一个脉冲，为了模拟这个过程，每个脉冲源有一个参数  $0 < r \leq 32,767$ ，并统一采用以下的伪随机函数：

C++ 版本：

```
static unsigned long next = 1;

/* RAND_MAX assumed to be 32767 */
int myrand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

Python 版本:

```
next = 1
def myrand():
    global next
    next = (next * 1103515245 + 12345) % (2 ** 64)
    return (next // 65536) % 32768
```

Java 版本:

```
long next = 1;
int myrand() {
    next = next * 1103515245 + 12345;
    return (int)((Long.divideUnsigned(next, 65536)) % 32768);
}
```

在每个时间刻,按照编号顺序从小到大,每个脉冲源调用一次上述的伪随机函数,当  $r > \text{myrand}()$  时,在当前时间刻发放一次脉冲,并通过突触传播到神经元。

进行仿真的时候,已知 0 时间刻各个神经元的状态,从 1 时间刻开始按照上述规则进行计算,直到完成  $T$  时刻的计算,再输出  $T$  时刻神经元的  $v$  值和发放的脉冲次数分别的最小值和最大值。

规定输入数据中结点按如下方式顺序编号:  $[0, N-1]$  为神经元的编号,  $[N, N+P-1]$  为脉冲源的编号。

代码中请使用双精度浮点类型。

## 输入格式

从标准输入读入数据。

输入的第一行包括四个以空格分隔的正整数  $N S P T$ , 表示一共有  $N$  个神经元,  $S$  个突触和  $P$  个脉冲源, 输出时间刻  $T$  时神经元的  $v$  值。

输入的第二行是一个正实数  $\Delta t$ , 表示时间间隔。

输入接下来的若干行, 每行有以空格分隔的一个正整数  $R_N$  和六个实数  $v u a b c d$ , 按顺序每一行对应  $R_N$  个具有相同初始状态和常量的神经元: 其中  $v u$  表示神经元在时刻 0 时的变量取值;  $a b c d$  为该神经元微分方程里的四个常量。保证所有的  $R_N$  加起来等于  $N$ 。它们从前向后按编号顺序描述神经元, 每行对应一段连续编号的神经元的信息。

输入接下来的  $P$  行, 每行是一个正整数  $r$ , 按顺序每一行对应一个脉冲源的  $r$  参数。

输入接下来的  $S$  行, 每行有以空格分隔的两个整数  $s(0 \leq s < N+P)$ 、 $t(0 \leq t < N)$ 、一个实数  $w(w \geq 0)$  和一个正整数  $D$ , 其中  $s$  和  $t$  分别是入结点和出结点的编号;  $w$  和  $D$  分别表示脉冲强度和传播延迟。

## 输出格式

输出到标准输出。

输出共有两行，第一行由两个近似保留 3 位小数的实数组成，分别是所有神经元在时刻  $T$  时变量  $v$  的取值的最小值和最大值。第二行由两个整数组成，分别是所有神经元在整个模拟过程中发放脉冲次数的最小值和最大值。

只要按照题目要求正确实现就能通过，不会因为计算精度的问题而得到错误答案。

### 样例 #1

输入 #1:

```
1 1 1 10
0.1
1 -70.0 -14.0 0.02 0.2 -65.0 2.0
30000
1 0 30.0 2
```

输出 #1:

```
-35.608 -35.608
2 2
```

解释 #1:

该样例有 1 个神经元、1 个突触和 1 个脉冲源，时间间隔  $\Delta t = 0.1$ 。唯一的脉冲源通过脉冲强度为 30.0、传播延迟为 2 的突触传播到唯一的神经元。

该样例一共进行 10 个时间步的模拟，随机数生成器生成 10 次随机数如下：

```
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

因此唯一的脉冲源在时刻 1-4 和 6-10 发放脉冲。在时间刻从 1 到 10 时，唯一的神经元的  $v$  取值分别为：

```
-70.000
-70.000
-40.000
-8.200
-65.000
-35.404
-32.895
0.181
-65.000
-35.608
```

该神经元在时刻 5 和时刻 9 发放，最终得到的  $v = -35.608$ 。



## 样例 #2

输入 #2:

```

1 1 1 10
0.1
1 -70.0 -14.0 0.02 0.2 -65.0 2.0
30000
1 0 30.0 2

```

输出 #2:

```

-35.608 -35.608
2 2

```

## 子任务

子任务	$T$	$N$	$S$	$P$	$D$	分值
1	$\leq 10^2$	$\leq 10^2$	$\leq 10^2$	$\leq 10^2$	$\leq 10^2$	30
2	$\leq 10^3$	$\leq 10^3$	$\leq 10^3$	$\leq 10^3$	$\leq 10^3$	40
3	$\leq 10^5$	$\leq 10^3$	$\leq 10^3$	$\leq 10^3$	$\leq 10$	30

## 3.5 202109-4 收集卡牌

### 题目描述

小林在玩一个抽卡游戏，其中有  $n$  种不同的卡牌，编号为 1 到  $n$ 。每一次抽卡，她获得第  $i$  种卡牌的概率为  $p_i$ 。如果这张卡牌之前已经获得过了，就会转化为一枚硬币。可以用  $k$  枚硬币交换一张没有获得过的卡。

小林会一直抽卡，直至集齐了所有种类的卡牌为止，求她的期望抽卡次数。如果你给出的答案与标准答案的绝对误差不超过  $10^{-4}$ ，则视为正确。

提示：聪明的小林会把硬币攒在手里，等到通过兑换就可以获得剩余所有卡牌时，一次性兑换并停止抽卡。

### 输入格式

从标准输入读入数据。

输入共两行。第一行包含两个用空格分隔的正整数  $n, k$ ，第二行给出  $p_1, p_2, \dots, p_n$ ，用空格分隔。

### 输出格式

输出到标准输出。

输出共一行，一个实数，即期望抽卡次数。

### 样例

#### 样例 #1

输入 #1:

输出 #1:

```
2 2
0.4 0.6
```

```
2.52
```

解释 #1:

共有 2 种卡牌，不妨记为 A 和 B，获得概率分别为 0.4 和 0.6，2 枚硬币可以换一张卡牌。下面给出各种可能出现的情况：

1. 第一次抽卡获得 A，第二次抽卡获得 B，抽卡结束，概率为  $0.4 \times 0.6 = 0.24$ ，抽卡次数为 2。
2. 第一次抽卡获得 A，第二次抽卡获得 A，第三次抽卡获得 B，抽卡结束，概率为  $0.4 \times 0.4 \times 0.6 = 0.096$ ，抽卡次数为 3。
3. 第一次抽卡获得 A，第二次抽卡获得 A，第三次抽卡获得 A，用硬币兑换 B，抽卡结束，概率为  $0.4 \times 0.4 \times 0.4 = 0.064$ ，抽卡次数为 3。
4. 第一次抽卡获得 B，第二次抽卡获得 A，抽卡结束，概率为  $0.6 \times 0.4 = 0.24$ ，抽卡次数为 2。
5. 第一次抽卡获得 B，第二次抽卡获得 B，第三次抽卡获得 A，抽卡结束，概率为  $0.6 \times 0.6 \times 0.4 = 0.144$ ，抽卡次数为 3。
6. 第一次抽卡获得 B，第二次抽卡获得 B，第三次抽卡获得 B，用硬币兑换 A，抽卡结束，概率为  $0.6 \times 0.6 \times 0.6 = 0.216$ ，抽卡次数为 3。

因此答案是  $0.24 \times 2 + 0.096 \times 3 + 0.064 \times 3 + 0.24 \times 2 + 0.144 \times 3 + 0.216 \times 3 = 2.52$ 。

## 样例 #2

输入 #2:

```
4 3
0.006 0.1 0.2 0.694
```

输出 #2:

```
7.3229920752
```

## 子任务

对于 20% 的数据，保证  $1 \leq n, k \leq 5$ 。

对于另外 20% 的数据，保证所有  $p_i$  是相等的。

对于 100% 的数据，保证  $1 \leq n \leq 16, 1 \leq k \leq 5$ ，所有的  $p_i$  满足  $p_i \geq \frac{1}{10000}$ ，且  $\sum_{i=1}^n p_i = 1$ 。

## 3.6 202109-5 箱根山岳险天下

### 题目背景

“你知道对长跑选手来说，最棒的赞美是什么吗？”

“是‘快’吗？”

“不，是‘强’，”清濑说，“光跑得快，是没办法在长跑中脱颖而出的。天候、场地、比赛的发展、体能，还有自己的精神状态——长跑选手必须冷静分析这许多要素，即使面对再大的困难，也要坚忍不拔地突破难关。长跑选手需要的，是真正的‘强’。所以我们必须把‘强’当作最高的荣誉，每天不断跑下去。”

不论阿走或其他房客，全都全神贯注地聆听清濑的话。

“看了你这三个月来的表现，我越来越相信自己没看错人，”清濑接着说，“你很有天分，也很有潜力。所以呢，阿走，你一定要更相信自己，不要急着想一飞冲天。变强需要时间，也可以说它永远没有终点。长跑是值得一生投入的竞赛，有些人即使老了，仍然没有放弃慢跑或马拉松运动。”

——三浦紫苑《强风吹拂》

箱根驿传（正式名称为东京箱根间往复大学驿传竞走）是日本一项在每年1月2-3日举行的驿站接力赛，由关东学生田径联盟主办，关东的每所高校都有机会参加。在日本，箱根驿传是新年假期必看的比赛，许多家庭会一边吃年糕汤一边欣赏激烈的比赛。

今年，京都大学也想派出长跑队参加箱根驿传，田径部的长跑教练组织起一批预备役运动员，并开展了严苛的训练。

### 题目描述

京都大学的训练一共会持续  $m$  天，在训练过程中正式队员的名单可能发生变化。简单起见，我们约定在且仅在第  $t$  ( $1 \leq t \leq m$ ) 天结束时，会有以下三种事件之一发生：

1. 有一个学生跑 10km 的速度达到了正式队员要求，教练将其作为最后一名纳入正式队员的名单中，这个学生的强度为  $x$ ；或者速度排名在最后一位的正式队员，由于速度过慢，而被从正式队员的名单中淘汰。
  - 在训练过程中，我们假定队员的速度的相对排名不会发生变化，与强度无关。
  - 严苛的教练制订了残酷的规则：被淘汰的学生虽然依然会跟大家一起训练，但将不能再次加入本年度参加箱根驿传的正式队员的名单中。
2. 由于近日的训练，第  $s$  天结束时速度排名为  $l$  至  $r$  的选手的强度有了变化，变为此前的  $y$  倍。
3. 教练在深夜想知道近日训练的效果，于是他统计了第  $s$  天结束时速度排名为  $l$  至  $r$  的选手目前（即第  $t$  天结束时）强度的和。由于这个结果可能很大，方便起见我们只考虑其模  $p$  的值。

出于学生们的隐私考虑，事件日志有可能会被加密。

### 输入格式

从标准输入读入数据。

第一行为三个用空格隔开的整数  $m$ ， $p$  和  $T$ 。

如果  $T = 0$ ，事件 1 中  $x = x'$ ，事件 2 中  $y = y'$ ；如果  $T = 1$ ，表示事件日志被加密了，事件 1 中  $x = x' \oplus A$ ，事件 2 中  $y = y' \oplus A$ ，其中  $\oplus$  为按位异或运算， $A$  为此前最后一次事件 3 所统计出的结果。如果此前没有事件 3 发生，则  $A = 0$ 。

接下来  $m$  行，第  $t$  行表示在第  $t$  天结束时发生的事件：

- $1\ x'$ ：表示事件 1 发生。若  $x > 0$ ，表示有一个强度为  $x$  的学生作为最后一名纳入正式队员的名单；若  $x = 0$ ，表示排名在最后的正式队员被从名单中淘汰。保证有  $0 \leq x' < 2^{30}$ 。

- $2s l r y'$ : 表示事件 2 发生。保证有  $1 \leq s \leq t$ ,  $1 \leq l \leq r \leq n$ ,  $0 \leq y' < 2^{30}$ , 其中  $n$  为在第  $s$  天结束时正式队员的人数。
- $3s l r$ : 表示事件 3 发生。保证有  $1 \leq s \leq t$ ,  $1 \leq l \leq r \leq n$ , 其中  $n$  为在第  $s$  天结束时正式队员的人数。

## 输出格式

输出到标准输出。

对于每一个事件 3, 输出一行一个数字, 为其所统计出的结果。

## 样例 #1

输入 #1:

输出 #1:

```
8 10 0
1 7
1 3
1 0
1 4
2 4 1 2 2
3 2 1 2
2 1 1 1 3
3 6 1 2
```

```
7
0
```

解释 #1:

第 1 天结束时, 有一个强度为 7 的学生被列为正式队员, 我们不妨称他小津。此时正式队员名单依次为: 小津。

第 2 天结束时, 有一个强度为 3 的学生被列为正式队员, 我们不妨称他为城崎。此时正式队员名单依次为: 小津、城崎。

第 3 天结束时, 城崎被淘汰了。此时正式队员名单为: 小津。

第 4 天结束时, 有一个强度为 4 的学生被列为正式队员, 我们不妨称他为樋口清太郎。此时正式队员名单依次为: 小津、樋口清太郎。

第 5 天结束时, 由于近日的训练, 第 4 天正式队员名单中第 1 至 2 个人——即小津和樋口清太郎——的强度乘了 2, 所以, 小津的强度达到了 14, 樋口清太郎的强度达到了 8。

第 6 天结束时, 教练统计了第 2 天正式队员名单中第 1 至 2 个人——即小津和城崎——当前的强度, 小津的强度为 14, 城崎的强度为 3, 故统计结果为 17, 模  $p$  的值为 7。

第 7 天结束时, 由于近日的训练, 第 1 天正式队员名单中的第 1 个人——即小津——的强度乘了 3, 所以, 小津的强度达到了 42。

第 8 天结束时, 教练统计了第 6 天正式队员名单中第 1 至 2 个人——即小津和樋口清太郎——当前的强度, 小津的强度为 42, 樋口清太郎的强度为 8, 故统计结果为 50, 模  $p$  的值为 0。

## 样例 #2

输入 #2:

输出 #2:

见 data/23/5-2.in

见 data/23/5-2.out

## 子任务

$$1 \leq m \leq 3 \times 10^5, 2 \leq p < 2^{30}, mode \in \{0, 1\}$$

测试点	特殊性质	<i>mode</i>
1	$m \leq 5000$	1
2	事件 1 中 $x > 0$	1
3	没有事件 2	1
4	事件 1 $x$ 在 $0, 1$ 中随机选取	0
5	$r - l \leq 10$	0
6	$r - l \leq 10$	1
7,8	无	0
9,10	无	1

## 第 4 章 第 22 次认证（2021 年 04 月）

### 4.1 题目及设计知识点

题目编号	题目名称	知识点
202104-1	灰度直方图	
202104-2	邻域均值	
202104-3	DHCP 服务器	
202104-4	校门外的树	
202104-5	疫苗运输	

## 4.2 202104-1 灰度直方图

### 题目描述

一幅长宽分别为  $n$  个像素和  $m$  个像素的灰度图像可以表示为一个  $n \times m$  大小的矩阵  $A$ 。

其中每个元素  $A_{ij}$  ( $0 \leq i < n$ 、 $0 \leq j < m$ ) 是一个  $[0, L)$  范围内的整数，表示对应位置像素的灰度值。

具体来说，一个 8 比特的灰度图像中每个像素的灰度范围是  $[0, 128)$ 。

一副灰度图像的灰度统计直方图（以下简称“直方图”）可以表示为一个长度为  $L$  的数组  $h$ ，其中  $h[x]$  ( $0 \leq x < L$ ) 表示该图像中灰度值为  $x$  的像素个数。显然， $h[0]$  到  $h[L-1]$  的总和应等于图像中的像素总数  $n \cdot m$ 。

已知一副图像的灰度矩阵  $A$ ，试计算其灰度直方图  $h[0], h[1], \dots, h[L-1]$ 。

### 输入格式

输入共  $n+1$  行。

输入的第一行包含三个用空格分隔的正整数  $n$ 、 $m$  和  $L$ ，含义如前文所述。

第二到第  $n+1$  行输入矩阵  $A$ 。

第  $i+2$  ( $0 \leq i < n$ ) 行包含用空格分隔的  $m$  个整数，依次为  $A_{i0}, A_{i1}, \dots, A_{i(m-1)}$ 。

### 输出格式

输出仅一行，包含用空格分隔的  $L$  个整数  $h[0], h[1], \dots, h[L-1]$ ，表示输入图像的灰度直方图。

### 样例

输入 #1:

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

输出 #1:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

输入 #2:

```
7 11 8
0 7 0 0 0 7 0 0 7 7 0
7 0 7 0 7 0 7 0 7 0 7
7 0 0 0 7 0 0 0 7 0 7
7 0 0 0 0 7 0 0 7 7 0
7 0 0 0 0 0 7 0 7 0 0
7 0 7 0 7 0 7 0 7 0 0
0 7 0 0 0 7 0 0 7 0 0
```

输出 #2:

```
48 0 0 0 0 0 0 29
```



## 子任务

全部的测试数据满足  $0 < n, m \leq 500$  且  $4 \leq L \leq 256$ 。

## 4.3 202104-2 邻域均值

### 试题背景

顿顿在学习了数字图像处理后，想要对手上的一副灰度图像进行降噪处理。不过该图像仅在较暗区域有很多噪点，如果贸然对全图进行降噪，会在抹去噪点的同时也模糊了原有图像。因此顿顿打算先使用邻域均值来判断一个像素是否处于较暗区域，然后仅对处于较暗区域的像素进行降噪处理。

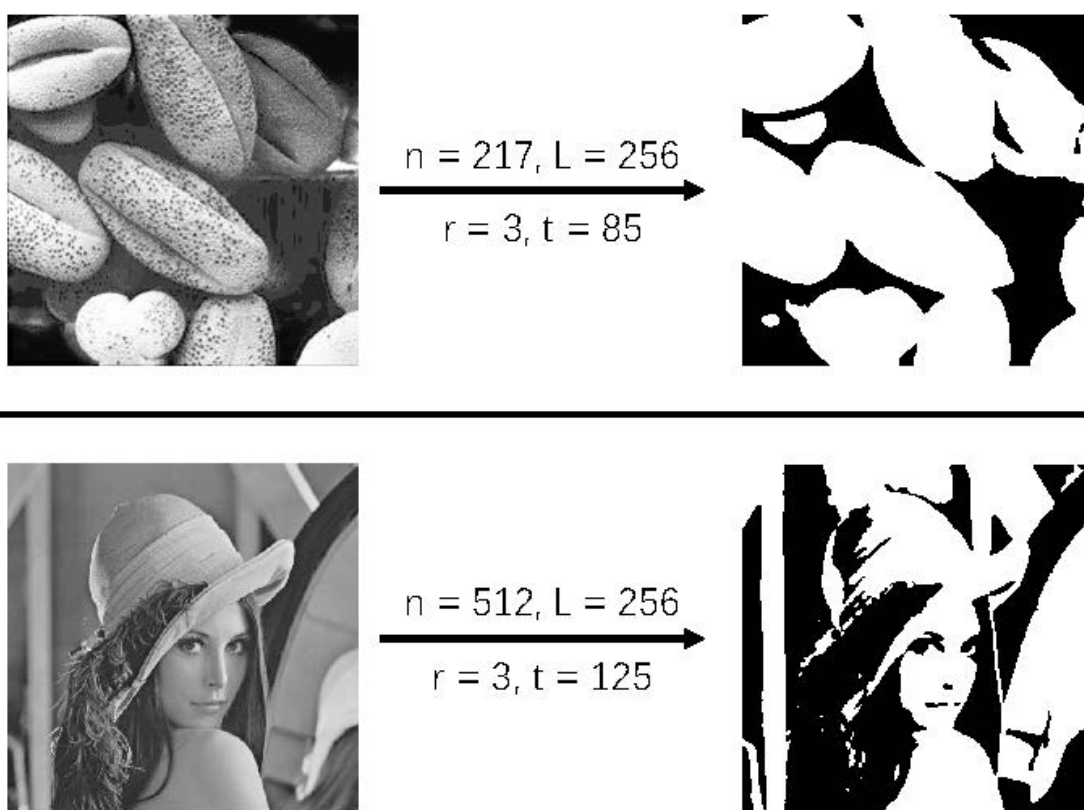
### 问题描述

待处理的灰度图像长宽皆为  $n$  个像素，可以表示为一个  $n \times n$  大小的矩阵  $A$ ，其中每个元素是一个  $[0, L)$  范围内的整数，表示对应位置像素的灰度值。对于矩阵中任意一个元素  $A_{ij}$  ( $0 \leq i, j < n$ )，其邻域定义为附近若干元素的集和：

$$Neighbor(i, j, r) = \{A_{xy} | 0 \leq x, y < n \text{ and } |x - i| \leq r \text{ and } |y - j| \leq r\}$$

这里使用了一个额外的参数  $r$  来指明  $A_{ij}$  附近元素的具体范围。根据定义，易知  $Neighbor(i, j, r)$  最多有  $(2r + 1)^2$  个元素。

如果元素  $A_{ij}$  邻域中所有元素的平均值小于或等于一个给定的阈值  $t$ ，我们就认为该元素对应位置的像素处于较暗区域。下图给出了两个例子，左侧图像的较暗区域在右侧图像中展示为黑色，其余区域展示为白色。



现给定邻域参数  $r$  和阈值  $t$ ，试统计输入灰度图像中有多少像素处于较暗区域。

### 输入格式

输入共  $n + 1$  行。

输入的第一行包含四个用空格分隔的正整数  $n$ 、 $L$ 、 $r$  和  $t$ ，含义如前文所述。

第二到第  $n+1$  行输入矩阵  $A$ 。第  $i+2$  ( $0 \leq i < n$ ) 行包含用空格分隔的  $n$  个整数, 依次为  $A_{i0}, A_{i1}, \dots, A_{i(n-1)}$ 。

## 输出格式

输出一个整数, 表示输入灰度图像中处于较暗区域的像素总数。

## 样例

输入 #1:

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

输出 #1:

```
7
```

输入 #2:

```
11 8 2 2
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 7 0 0 0 7 0 0 7 7 0
7 0 7 0 7 0 7 0 7 0 7
7 0 0 0 7 0 0 0 7 0 7
7 0 0 0 0 7 0 0 7 7 0
7 0 0 0 0 0 7 0 7 0 0
7 0 7 0 7 0 7 0 7 0 0
0 7 0 0 0 7 0 0 7 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

输出 #2:

```
83
```

## 子任务

70% 的测试数据满足  $n \leq 100$ 、 $r \leq 10$ 。

全部的测试数据满足  $0 < n \leq 600$ 、 $0 < r \leq 100$  且  $2 \leq t < L \leq 256$ 。

## 4.4 202104-3 DHCP 服务器

### 试题背景

动态主机配置协议（Dynamic Host Configuration Protocol, DHCP）是一种自动为网络客户端分配 IP 地址的网络协议。当支持该协议的计算机刚刚接入网络时，它可以启动一个 DHCP 客户端程序。后者可以通过一定的网络报文交互，从 DHCP 服务器上获得 IP 地址等网络配置参数，从而能够在用户不干预的情况下，自动完成对计算机的网络设置，方便用户连接网络。DHCP 协议的工作过程如下：

1. 当 DHCP 协议启动的时候，DHCP 客户端向网络中广播发送 Discover 报文，请求 IP 地址配置；
2. 当 DHCP 服务器收到 Discover 报文时，DHCP 服务器根据报文中的参数选择一个尚未分配的 IP 地址，分配给该客户端。DHCP 服务器用 Offer 报文将这个信息传达给客户端；
3. 客户端收集收到的 Offer 报文。由于网络中可能存在多于一个 DHCP 服务器，因此客户端可能收集到多个 Offer 报文。客户端从这些报文中选择一个，并向网络中广播 Request 报文，表示选择这个 DHCP 服务器发送的配置；
4. DHCP 服务器收到 Request 报文后，首先判断该客户端是否选择本服务器分配的地址：如果不是，则在本服务器上解除对那个 IP 地址的占用；否则则再次确认分配的地址有效，并向客户端发送 Ack 报文，表示确认配置有效，Ack 报文中包括配置的有效时间。如果 DHCP 发现分配的地址无效，则返回 Nak 报文；
5. 客户端收到 Ack 报文后，确认服务器分配的地址有效，即确认服务器分配的地址未被其它客户端占用，则完成网络配置，同时记录配置的有效时间，出于简化的目的，我们不考虑被占用的情况。若客户端收到 Nak 报文，则从步骤 1 重新开始；
6. 客户端在到达配置的有效时间前，再次向 DHCP 服务器发送 Request 报文，表示希望延长 IP 地址的有效期。DHCP 服务器按照步骤 4 确定是否延长，客户端按照步骤 5 处理后续的配置；

在本题目中，你需要理解 DHCP 协议的工作过程，并按照题目的要求实现一个简单的 DHCP 服务器。

### 问题描述

#### 报文格式

为了便于实现，我们简化地规定 DHCP 数据报文的格式如下：

<发送主机> <接收主机> <报文类型> <IP 地址> <过期时刻>

DHCP 数据报文的各个部分由空格分隔，其各个部分的定义如下：

- 发送主机：是发送报文的主机名，主机名是由小写字母、数字组成的字符串，唯一地表示了一个主机；
- 接收主机：当有特定的接收主机时，是接收报文的主机名；当没有特定的接收主机时，为一个星号（\*）；
- 报文类型：是三个大写字母，取值如下：
  - DIS：表示 Discover 报文；
  - OFR：表示 Offer 报文；
  - REQ：表示 Request 报文；
  - ACK：表示 Ack 报文；
  - NAK：表示 Nak 报文；
- IP 地址，是一个非负整数：
  - 对于 Discover 报文，该部分在发送的时候为 0，在接收的时候忽略；
  - 对于其它报文，为正整数，表示一个 IP 地址；
- 过期时刻，是一个非负整数：
  - 对于 Offer、Ack 报文，是一个正整数，表示服务器授予客户端的 IP 地址的过期时刻；
  - 对于 Discover、Request 报文，若为正整数，表示客户端期望服务器授予的过期时刻；

- 对于其它报文，该部分在发送的时候为 0，在接收的时候忽略。

例如下列都是合法的 DHCP 数据报文：

```
a * DIS 0 0
d a ACK 50 1000
```

## 服务器配置

为了 DHCP 服务器能够正确分配 IP 地址，DHCP 需要接受如下配置：

- 地址池大小  $N$ ：表示能够分配给客户端的 IP 地址的数目，且能分配的 IP 地址是  $1, 2, \dots, N$ ；
- 默认过期时间  $T_{def}$ ：表示分配给客户端的 IP 地址的默认的过期时间长度；
- 过期时间的上限和下限  $T_{max}$ 、 $T_{min}$ ：表示分配给客户端的 IP 地址的最长过期时间长度和最短过期时间长度，客户端不能请求比这个更长或更短的过期时间；
- 本机名称  $H$ ：表示运行 DHCP 服务器的主机名。

## 分配策略

当客户端请求 IP 地址时，首先检查此前是否给该客户端分配过 IP 地址，且该 IP 地址在此后没有被分配给其它客户端。如果是这样的情况，则直接将 IP 地址分配给它，否则，总是分配给它最小的尚未占用过的那个 IP 地址。如果这样的地址不存在，则分配给它最小的此时未被占用的那个 IP 地址。如果这样的地址也不存在，说明地址池已经分配完毕，因此拒绝分配地址。

## 实现细节

在 DHCP 启动时，首先初始化 IP 地址池，将所有地址设置状态为未分配，占用者为空，并清零过期时刻。其中地址的状态有未分配、待分配、占用、过期四种。处于未分配状态的 IP 地址没有占用者，而其余三种状态的 IP 地址均有一名占用者。处于待分配和占用状态的 IP 地址拥有一个大于零的过期时刻。在到达该过期时刻时，若该地址的状态是待分配，则该地址的状态会自动变为未分配，且占用者清空，过期时刻清零；否则该地址的状态会由占用自动变为过期，且过期时刻清零。处于未分配和过期状态的 IP 地址过期时刻为零，即没有过期时刻。

对于收到的报文，设其收到的时刻为  $t$ 。处理细节如下：

1. 判断接收主机是否为本机，或者为  $*$ ，若不是，则判断类型是否为 Request，若不是，则不处理；
2. 若类型不是 Discover、Request 之一，则不处理；
3. 若接收主机为  $*$ ，但类型不是 Discover，或接收主机是本机，但类型是 Discover，则不处理。

对于 Discover 报文，按照下述方法处理：

1. 检查是否有占用者为发送主机的 IP 地址：
  - 若有，则选取该 IP 地址；
  - 若没有，则选取最小的状态为未分配的 IP 地址；
  - 若没有，则选取最小的状态为过期的 IP 地址；
  - 若没有，则不处理该报文，处理结束；
2. 将该 IP 地址状态设置为待分配，占用者设置为发送主机；
3. 若报文中过期时刻为 0，则设置过期时刻为  $t + T_{def}$ ；否则根据报文中的过期时刻和收到报文的时刻计算过期时间，判断是否超过上下限：若没有超过，则设置过期时刻为报文中的过期时刻；否则则根据超限情况设置为允许的最早或最晚的过期时刻；
4. 向发送主机发送 Offer 报文，其中，IP 地址为选定的 IP 地址，过期时刻为所设定的过期时刻。

对于 Request 报文，按照下述方法处理：

## 1. 检查接收主机是否为本机：

- 若不是，则找到占用者为发送主机的所有 IP 地址，对于其中状态为待分配的，将其状态设置为未分配，并清空其占用者，清零其过期时刻，处理结束；

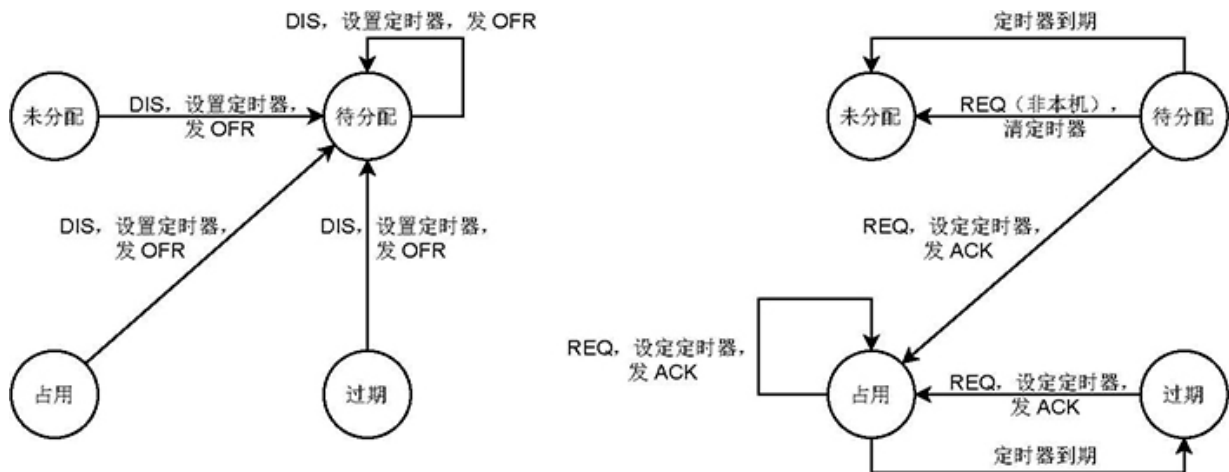
## 2. 检查报文中的 IP 地址是否在地址池内，且其占用者为发送主机，若不是，则向发送主机发送 Nak 报文，处理结束；

## 3. 无论该 IP 地址的状态为何，将该 IP 地址的状态设置为占用；

## 4. 与 Discover 报文相同的方法，设置 IP 地址的过期时刻；

## 5. 向发送主机发送 Ack 报文。

上述处理过程中，地址池中地址的状态的变化可以概括为如下图所示的状态转移图。为了简洁，该图中没有涵盖需要回复 Nak 报文的情况。



## 输入格式

输入的第一行包含用空格分隔的四个正整数和一个字符串，分别是： $N$ 、 $T_{def}$ 、 $T_{max}$ 、 $T_{min}$  和  $H$ ，保证  $T_{min} \leq T_{def} \leq T_{max}$ 。

输入的第二行是一个正整数  $n$ ，表示收到了  $n$  个报文。

输入接下来有  $n$  行，第  $(i+2)$  行有空格分隔的正整数  $t_i$  和约定格式的报文  $P_i$ 。表示收到的第  $i$  个报文是在  $t_i$  时刻收到的，报文内容是  $P_i$ 。保证  $t_i < t_{i+1}$ 。

## 输出格式

输出有若干行，每行是一个约定格式的报文。依次输出 DHCP 服务器发送的报文。

## 4.5 202104-4 校门外的树

### 问题描述

X 校最近打算美化一下校园环境。前段时间因为修地铁，X 校大门外种的行道树全部都被移走了。现在 X 校打算重新再种一些树，为校园增添一抹绿意。

X 校大门外的道路是东西走向的，我们可以将其看成一条数轴。在这条数轴上有  $n$  个障碍物，例如电线杆之类的。虽然障碍物会影响树的生长，但是障碍物不一定能被随便移走，所以 X 校规定在障碍物的位置上不能种树。 $n$  个障碍物的坐标都是整数；如果规定向东为正方向，则  $n$  个障碍物的坐标按照从西到东的顺序分别为  $a_1, a_2, \dots, a_n$ 。X 校打算在  $[a_1, a_n]$  之间种一些树，使得这些树看起来比较美观。

X 校希望，在一定范围内，树应该是等间隔的。更具体地说，如果把  $[a_1, a_n]$  划分成一些区间  $[a_{p_1}, a_{p_2}), \dots, [a_{p_{m-1}}, a_{p_m})$  ( $1 = p_1 < p_2 < \dots < p_m = n$ )，那么每个区间  $[a_{p_i}, a_{p_{i+1}})$  内需要至少种一棵树，且该区间内种的树的坐标连同区间端点  $a_{p_i}, a_{p_{i+1}}$  应该构成一个等差数列。不同区间的公差，也就是树的间隔可以不相同。

例如，如果障碍物位于 0, 2, 6 这三处，那么我们可以选择在  $[0, 2)$  和  $[2, 6)$  分别种树，也可以选择在  $[0, 6)$  等间隔种树。如果是分别在  $[0, 2)$  和  $[2, 6)$  种树，由于每个区间内至少要种一棵树，坐标 1 上必须种树；而  $[2, 6)$  上的树可以按照 1 的间隔种下，也可以按照 2 的间隔种下。下图表示了这两种美观的种树方案，其中橙色的圆表示障碍物，绿色的圆表示需要在这个位置种树，箭头上的数字表示种下这棵树时对应的间隔为多少。



对区间  $[0, 2)$  和  $[2, 6)$  分别以 1 和 2 的间隔种树是美观的



对区间  $[0, 2)$  和  $[2, 6)$  分别以 1 的间隔种树也是美观的

而如果选择在  $[0, 6)$  区间等间隔种树，我们只能以 3 的间隔种树，因为无论是选择间隔 1 或者间隔 2，都需要在坐标 2 上种树，而这个位置已经有障碍物了。下图分别表示了间隔为 3, 2, 1 时的种树情况，红色箭头表示不能在这里种树。





对区间  $[0, 6)$  以 3 的间隔种树是美观的



对区间  $[0, 6)$  以 2 的间隔种树是不美观的



对区间  $[0, 6)$  以 1 的间隔种树也是不美观的

一般地，给定一个区间  $[a_l, a_r)$ ，对于树的坐标的集合  $T \subset (a_l, a_r)$  ( $T \subset \mathbb{Z}$ )，归纳定义  $T$  在  $[a_l, a_r)$  上是美观的：

1. 如果  $T \neq \emptyset$ ， $T \cap \{a_l, a_{l+1}, \dots, a_r\} = \emptyset$ ，并且存在一个公差  $d \geq 1$ ，使得  $T \cup \{a_l, a_r\}$  中的元素按照从小到大的顺序排序后，可以构成一个公差为  $d$  的等差数列（显然，这个等差数列的首项为  $a_l$ ，末项为  $a_r$ ），则  $T$  在  $[a_l, a_r)$  上是美观的；
2. 如果  $T \cap \{a_l, a_{l+1}, \dots, a_r\} = \emptyset$ ，并且存在一个下标  $m$  ( $l < m < r$ )，使得  $T \cap (a_l, a_m)$  在  $[a_l, a_m)$  上是美观的，且  $T \cap (a_m, a_r)$  在  $[a_m, a_r)$  上是美观的，则  $T$  在  $[a_l, a_r)$  上是美观的。

根据这一定义，空集在任意区间上都不是美观的；另外，如果存在下标  $i$  使得  $a_i \in T$ ，那么  $T$  一定不是美观的。

我们称两种种树的方案是**本质不同的**，当且仅当两种方案中，种树的坐标集合不同。请帮助 X 校对  $[a_1, a_n)$  求出所有本质不同的美观的种树方案。当然，由于方案可能很多，你只需要输出总方案数对  $10^9 + 7$  取模的结果。



### 输入格式

输入的第一行包含一个正整数  $n$ ，表示障碍物的数量。

输入的第二行包括  $n$  个非负整数  $a_1, \dots, a_n$ ，表示每个障碍物的坐标。

保证对  $i = 1, 2, \dots, n-1$ ， $a_i < a_{i+1}$ 。

### 输出格式

输出一个非负整数，表示本质不同的美观的种树方案的数量对  $10^9 + 7$  取模的结果。

## 4.6 202104-5 疫苗运输

### 问题描述

X 市最近生产了一批疫苗，需要运往各地使用。疫苗的运输是一个困难的问题：既要实现尽快时间送达，又要保证全程冷链，否则疫苗会损坏。

X 市的物流系统并不发达，只有  $n$  个物流站点（以下简称“站点”）和  $m$  条物流线路（以下简称“线路”），且该物流系统具有以下几个特点：

1. 每条线路都是环线。即，从某个站点出发，经过一系列不重复的站点，最终回到出发站点。
2. 每条线路上有且仅有一辆运输车，以固定的时刻表（相邻站间的时间间隔）在环线上不断运行。在 0 时刻时，运输车在出发站点。
3. 运输车上配备了容量足够大的制冷系统，疫苗可以在车上长时间存放。但是换乘（从一条线路切换到另一条线路）必须在同一个站点同一个时刻发生——因为各个站点没有独立的制冷系统，疫苗不能在站点内下车等待。

现在 X 市想要从 1 号站点开始，经过若干条线路的运输和换乘，将疫苗运输到各个其他站点。与其他站点不同，1 号站点配有冷库。也就是说，从 0 时刻开始，可以在 1 号站点等待某条线路运输车的到来，再开始疫苗运输。问对于 2 号  $n$  号站点，分别最早可以在什么时刻将疫苗送到该站点。

注意：每个问题是独立的，即只需要求出 1 号站点到各个站点的最早送达时刻。

### 输入格式

第一行两个整数  $n, m$ 。

接下来  $m$  行，每行表示一条物流线路。对于第  $i$  ( $1 \leq i \leq m$ ) 条线路，首先有一个整数  $l_i$  ( $2 \leq l_i \leq n$ ) 表示该线路经过的站点个数。接下来  $2l_i$  个整数，第  $2j-1$  和第  $2j$  个整数分别表示该线路的第  $j$  ( $1 \leq j \leq l_i$ ) 个站点的编号  $a_{i,j}$  ( $1 \leq a_{i,j} \leq n$ )，以及该线路的第  $j$  个站点到下一个站点所需的时间  $t_{i,j}$  ( $1 \leq t_{i,j} \leq T$ )（对于第  $l_i$  个站点即为它到第 1 个站点的时间）。其中，每条线路的第 1 个站点为其出发站点。输入中同一行相邻的整数，均用一个空格隔开。

### 输出格式

输出  $n-1$  行，第  $i$  行表示将疫苗送达第  $i+1$  个站点的最早时间：如果能在有限时间内送达，输出最早的送达时刻；否则输出 `inf`。