

B. 汇编验证

考点：模拟，栈

签到题。

题外话，本题实际上是一个高度简化的Java虚拟机，本题中唯一要求实现的指令`call`实际上是Java虚拟机中的`invokestatic`指令。

正如题干所说的，你需要维护一个栈，记录虚拟机栈上各个元素的类型，遇到`call`指令时，将元素类型出栈，并与函数的参数列表中的类型逆序匹配，如果匹配成功，则将函数的返回值类型压栈，否则回答 `NO`。最后如果声明的返回值类型不是`0`，确认返回时栈上只有一个元素，且类型与返回值类型匹配，否则回答 `NO`，如果返回值类型是`0`，则确认栈为空，否则回答 `NO`。都检查通过则回答 `YES`。

标准程序：

```
class Func:
    def __init__(self, rval, args):
        self.rval = rval
        self.args = list(reversed(args)) # Turn the push-order into pop-order

class Stack:
    def __init__(self, init):
        self.items = init

    def call(self, func):
        for j in func.args:
            if len(self.items) == 0 or self.items.pop() != j:
                return False

        if func.rval != 0:
            self.items.append(func.rval)
        return True

n = int(input())
funcs = [None]
for i in range(n):
    rval, _, *args = map(int, input().split())
    funcs.append(Func(rval, args))
s, r, _, *p = map(int, input().split())
stack = Stack(p)
for i in range(s):
    j = int(input())
    if not stack.call(funcs[j]):
        print("NO")
        exit(0)

if (r == 0 and len(stack.items) == 0) or (len(stack.items) == 1 and stack.items[0] == r):
    print("YES")
else:
    print("NO")
```

评分标准：

- 1. 通过所有样例：100分

C. 打牌

考点：队列

本题要求实现一个至多 q 个元素的队列，并且在达到 q 个元素时，将最后一个元素出队，并维护队列中元素中每种牌的数量加一的积。也就是实现 $push(n)$ 和 $product()$ 两个操作。

考虑一个元素入队时，积如何变化。显然，如果队列中已经有 k 张相同的牌，积将变为 $\frac{k+1}{k}$ 倍。我们开一个数组统计每种牌有多少个，每次入队时，将积乘上 $\frac{k+1}{k}$ ，然后将该牌的数量加一。出队时，将积乘上 $\frac{k-1}{k}$ ，然后将该牌的数量减一。

队列本身可以使用 $q + 1$ 个元素的数组实现，使用两个指针分别指向队列的头和尾，每次入队时，将尾指针向后移动一位，出队时，将头指针向后移动一位。当指针指向数组的末尾时，将其置为0。队列满时，头指针和尾指针相邻，队列空时，头指针和尾指针相等。每次入队时，如果队是满的，就先出队一个，如题目所说。

标准程序：

```
class Queue:
    def __init__(self, sz):
        self.items = [None] * (sz + 1)
        self.head = 0
        self.tail = 0
        self.stat = [1] * 6
        self.prod = 1

    def push(self, n):
        if (self.head + 1) % len(self.items) == self.tail:
            self._pop() # Maintain fixed size queue
        self.items[self.head] = n
        self.head = (self.head + 1) % len(self.items)
        self.prod = self.prod // self.stat[n] * (self.stat[n] + 1)
        self.stat[n] += 1

    def _pop(self):
        n = self.items[self.tail]
        self.prod = self.prod // self.stat[n] * (self.stat[n] - 1)
        self.stat[n] -= 1
        self.tail = (self.tail + 1) % len(self.items)

d = 0
n, q = map(int, input().split())
qu = Queue(q)
for _ in range(n):
    z = int(input())
    x = (d + z) % 6
    qu.push(x)
    d = qu.prod
print(d)
```

评分标准：

- 1. print打印给出的样例输出：5分
- 2. 每次暴力遍历队列：60分
- 2. 通过所有样例：100分

D. 勒让德级数展开

考点：逼近，分部积分，勒让德多项式，列表

先来考虑怎么算勒让德多项式的问题，勒让德多项式的递推式为

$$\begin{aligned}P_0(x) &= 1 \\P_1(x) &= x \\P_n(x) &= \frac{2n-1}{n}xP_{n-1}(x) - \frac{n-1}{n}P_{n-2}(x)\end{aligned}$$

我们用一个浮点数的列表来保存勒让德多项式，列表的第*i*个（从0开始编号）元素表示 x^i 的系数，于是三项递推式可以表达为

$$P_n[i] = \frac{2n-1}{n}P_{n-1}[i-1] - \frac{n-1}{n}P_{n-2}[i]$$

*n*次勒让德多项式，我们数组开*n* + 1就可以了。用代码实现没啥难度，就是写的多了点。

然后我们知道系数

$$a_k = \frac{(f, P_k)}{(P_k, P_k)} = \frac{2k+1}{2} \int_{-1}^1 P_k(x) \cos(ax+b) dx$$

积分的话，很容易想到，把多项式的每一项拿出来和 $\cos(ax+b)$ 积分，然后再加起来就行了。问题转化为求一系列 $\int_{-1}^1 x^i \cos(ax+b) dx$ 的值。考虑分部积分，我们有

$$\begin{aligned}&\int_{-1}^1 x^i \cos(ax+b) dx \\&= \frac{1}{a} \int_{-1}^1 x^i d \sin(ax+b) \\&= \frac{1}{a} [x^i \sin(ax+b)]_{-1}^1 - \frac{1}{a} \int_{-1}^1 \sin(ax+b) dx^i \\&= \frac{1}{a} [\sin(a+b) - (-1)^i \sin(-a+b)] - \frac{i}{a} \int_{-1}^1 x^{i-1} \sin(ax+b) dx \\&= \frac{1}{a} [\sin(a+b) + (-1)^i \sin(a-b)] + \frac{i}{a^2} \int_{-1}^1 x^{i-1} d \cos(ax+b) \\&= \frac{1}{a} [\sin(a+b) + (-1)^i \sin(a-b)] + \frac{i}{a^2} [x^{i-1} \cos(ax+b)]_{-1}^1 - \frac{i}{a^2} \int_{-1}^1 \cos(ax+b) dx^{i-1} \\&= \frac{1}{a} [\sin(a+b) + (-1)^i \sin(a-b)] + \frac{i}{a^2} [\cos(a+b) + (-1)^i \cos(a-b)] - \frac{i(i-1)}{a^2} \int_{-1}^1 x^{i-2} \cos(ax+b) dx\end{aligned}$$

我们把 $\frac{1}{a} [\sin(a+b) + (-1)^i \sin(a-b)] + \frac{i}{a^2} [\cos(a+b) + (-1)^i \cos(a-b)]$ 写进程序里加到积分结果上，然后把 $\frac{i(i-1)}{a^2}$ 加到低次积分的系数上，然后从高次向低次计算，就可以求出这一堆积分的和了。需要注意的是，*i*为0或1时，就不需要继续操作其他项的系数了，因为 $\frac{i(i-1)}{a^2}$ 肯定为0，并且*i* - 2数组越界就不好玩了。

最后把积分结果代回去，就可以得到系数了。

你也可以直接进行数值积分，但本题没有这个必要。

标准程序：

```
from math import sin, cos

def legendre(n):
    ppp, pp, p = [1], [0, 1], None
    if n == 0:
        return ppp
    elif n == 1:
        return pp
    for i in range(2, n + 1):
        p = [0] * (len(pp) + 1)
        for j in range(len(pp)):
            p[j + 1] += (2 * i - 1) / i * pp[j]

        for j in range(len(ppp)):
            p[j] -= (i - 1) / i * ppp[j]

        ppp, pp = pp, p

    return p

def fit(a, b, order):
    if a == 0:
        return cos(b) if order == 1 else 0
    p = legendre(order)
    r1, r2 = 0, 0
    ans = 0
    for i in range(len(p) - 1, -1, -1):
        f = r1 + p[i]
        n = (-1) ** i
        ans += f * (\
            1 / a * (sin(a + b) + n * sin(a - b)) + \
            i / a ** 2 * (cos(a + b) + n * cos(a - b))\
        )
        r1, r2 = r2, -f * i * (i - 1) / a ** 2
    return ans * (2 * order + 1) / 2

print(fit(*map(int, input().split(" "))))
```

评分标准：

- 1. print打印给出的样例输出：5分
- 2. 只能求解*n* = 0的情况：20分
- 3. 只能算出课本上给出的前5项勒让德多项式的系数：40分
- 4. 通过所有样例：100分

E. 谁活到最后

考点：递归

本题是课本Chapter 2.4 Exercise 67的变形，如果你做了作业，这题应该有一个大致的思路。

这题一看典中典约瑟夫问题。递归三大经典题：汉诺塔问题，直线分割平面，约瑟夫问题。

这题就是约瑟夫问题的一个经典变形，应该能在网上找到很多解法。至于为什么叫这个名字，是模仿数据结构老师的命名。

什么是递归？

递归（归纳性定义）和数学归纳法是高度一致的，它们都由**基础步**、**归纳步/递归步**两部分组成。区别在于，数学归纳法希望证明一个结论，而递归希望定义某种东西（比如函数等）。

递归举例： 基础步： $0! = 1$ 递归步： $n! = n(n - 1)!$ ，其中 $n \geq 1$
--

好，我们现在来看这题怎么解，首先，我们令 $J(x)$ 表示 x 个人约瑟夫环，最后活下来的人的编号。为了方便，我们把 $J(x)$ 向下移动一个单位，即所有人从0开始编号，记这个函数为 $J_0(x)$ 。那么我们有 $J(x) = J_0(x) + 1$ 。这种平移的目的是为了 $0 \leq J_0(x) < x$ ，与取模运算结果的范围一致。

我们需要希望递归定义出 J_0 。为了确定基础步需要几步，我们先来看递归步。

显然，当 $x \geq 2$ 时，我们需要杀死一个人，这个人肯定不是存活的，存活的人将通过剩下的人约瑟夫环选出，只不过，剩下的 $x - 1$ 个人每个人的编号不是0到 $x - 2$ ，所以我们没办法直接复用 $J(x - 1)$ ，而是需要把剩下的人的编号从0到 $x - 2$ 映射回 x 个人时每个人的编号，除了编号以外这个约瑟夫过程和原来的一样。假设这个从 $x - 1$ 个人时每个人的编号映射回 x 个人时每个人的编号的函数叫 $R(m, x)$ ，我们有：

$$J_0(x) = R(J_0(x - 1), x)$$

这个 $R(m, x)$ 函数并不好想，我们把它的对应关系列出来，以试图找到一个解析式。为了方便，我们将被杀死的人前面的人弄到后面去，以打括号标记原来的位置。因为编号是从0开始的，所以被杀死的人是 $n - 1$ 号，我们从他的下一位开始标记，即0号。我们有：

$R(m, x)$	(0)	(1)	(...)	$n - 1$	n	$n + 1$	$n + 2$...	$x - 1$	0	1	...	$n - 2$
m				killed	0	1	2	...	$x - n - 1$	$x - n$	$x - n + 1$...	$x - 2$

这里有些不太严谨，因为如果 $x \leq n$ ，上述 n 应该是 $n \bmod x$ ，但是不影响结果，为了空间写得下，我们这张表中的 n 都是 $n \bmod x$ 。

很明显可以找到一个符合条件的函数解析式，即

$$R(m, x) = (m + n) \bmod x$$

因此我们得到了 J_0 的递归步：

$$J_0(x) = (J_0(x - 1) + n) \bmod x$$

注意到 $J_0(x)$ 只用到了 $J_0(x - 1)$ ，因此基础步只需要一步，即一个人约瑟夫环时，肯定这唯一一个人存活，他标号为0，即

$$J_0(1) = 0$$

因此我们得到了 J_0 的递归定义：

$$\begin{aligned} J_0(1) &= 0 \\ J_0(x) &= (J_0(x - 1) + n) \bmod x \end{aligned}$$

至此，你已经可以编写程序求解

```
def josephus0(x, n):  
    return 0 if x == 1 else (josephus0(x - 1, n) + n) % x  
def josephus(x, n):  
    return josephus0(x, n) + 1
```

但是，这里有个问题，它只能通过 x 比较小的测试样例，它的时间复杂度是 $O(x)$ ，而题目给出的数据范围是大 x 、小 n ，所以我们需要转向对 x 友好而不是对 n 友好的算法。

注意到，因为 n 特别小，而 x 又特别大，这就导致了99%的模 x 操作都在做无用功。我们可以考虑把这些无用功去掉，把 $J_0(x - 1)$ 展开。假设从 x 到 $x + s$ 这一段都没有有意义的模 x 操作，我们有：

$$J_0(x + s) = J_0(x) + s \cdot n$$

再假设我们只要向前走一步就会遇到有意义的模 x 操作，即

$$J_0(x + s + 1) = (J_0(x) + (s + 1) \cdot n) \bmod (x + s + 1)$$

式子感觉比较丑，令 $s_1 = s + 1$ ，我们有

$$J_0(x + s_1) = (J_0(x) + s_1 \cdot n) \bmod (x + s_1)$$

我们求解这个 s_1 ，注意到 $a \bmod b \neq a$ 等价于 $a \geq b$ ，我们有

$$J_0(x) + n \cdot s_1 \geq x + s_1$$

注意 $J_0(x) < x$ 恒成立，当 $n > 1$ 时

$$s_1 \geq \frac{x - J_0(x)}{n - 1}$$

当 $n = 1$ 时，无解，但此时 $J_0(x) = x - 1$ ，这是平凡的，到时候写程序来个特判就行了。下面的讨论都是在 $n > 1$ 的情况下进行的。

上界：

$$\begin{aligned} J_0(x) + n \cdot s &< x + s \\ s &< \frac{x - J_0(x)}{n - 1} \\ s_1 &< \frac{x - J_0(x)}{n - 1} + 1 \end{aligned}$$

结论：

$$s_1 = \left\lceil \frac{x - J_0(x)}{n - 1} \right\rceil$$

唯一与课本不同的是，课本中 $n = 2$ ，这意味着 $\frac{x - J_0(x)}{n - 1}$ 始终是整数，无需向上取整，从而我们在后面步骤中可以推出一个好看的通项公式，但其实对编程来说时间复杂度不会有任何变化。

通过这个式子，我们可以“快进”到 $J_0(x + s_1)$ ，省去中间的递归步骤。

$$s_1 = \left\lceil \frac{x_0 - J_0(x_0)}{n - 1} \right\rceil$$
$$J_0(x_0 + s_1) = (J_0(x_0) + s_1 \cdot n) \bmod (x_0 + s_1)$$

如果要求的 x 满足 $x_0 \leq x < x_0 + s_1$ ，我们有

$$J_0(x) = J_0(x_0) + (x - x_0) \cdot n$$

否则，令 $x_0 \leftarrow x_0 + s_1$ ，继续递推求解。

当然，你可以直接打表找到上述规律，然后再证明规律是正确的，这样的思维强度小很多。

标准程序：

```
def josephus(x, n):
    if n == 1:
        return x
    currx, curry = 1, 0
    while True:
        step = (currx - curry + n - 2) // (n - 1)
        nextx = currx + step
        nexty = (curry + n * step) % nextx
        if x < nextx:
            return curry + n * (x - currx) + 1
        currx, curry = nextx, nexty

print(josephus(*map(int, input().split(" "))))
```

评分标准：

- 1. 写对基础步，但是递归步写错：20分
- 2. 使用数组暴力模拟：40分
- 3. 使用链表暴力模拟：60分
- 4. 写对基础步和递归步，但是没化简：80分
- 5. 通过所有样例：100分

拓展

如果要求倒数第 m 个被杀死的，应该怎么做？（提示：基础步）