5 第二讲 作业记录（三）

笔记本： 浙江大学《数据结构》
创建时间： 2025/3/24 23:01 **更新时间：** 2025/4/6 16:13
作者： panhengye@163.com
URL： https://pintia.cn/problem-sets/1873565885118418944/exam/problems/type/7?...

题目：02-线性结构4 Pop Sequence

提交结果

题目	用户	提交时间
02-线性结构4	飞翔的小师弟	2025/03/24 22:03:13
编译器	内存	用时
Python (python3)	2992 / 65536 KB	19 / 400 ms
状态	分数	评测时间
答案正确	25 / 25	2025/03/24 22:03:14

评测详情

测试点	提示	内存(KB)	用时(ms)	结果	得分
0	sample乱序，一般的Y&N	2812	19	答案正确	15 / 15
1	达到最大size后又溢出	2992	15	答案正确	3 / 3
2	M==N	2816	15	答案正确	2 / 2
3	最大数	2980	16	答案正确	2 / 2
4	最小数	2884	15	答案正确	1 / 1
5	卡特殊错误算法（通过比较大小判断）	2932	15	答案正确	2 / 2

【感悟】

1. 当遇到需要判断正误时，一种常见的解决方案是模拟运行。因为计算机运算极快，所以可以用“复现”的方式去验证是否可行
2. 在设计逻辑分支时，注意尽量穷尽可能性。如果一时间想不清楚，可以先在else分支中做兜底性处理
3. 理解如何选择和应用适合问题的数据结构，知道它们的时间复杂度和空间复杂度特性

【算法复杂度】

- **时间复杂度**：O(N)
- **空间复杂度**：O(N)

这是一个非常高效的算法，它利用栈的特性，通过一次遍历就能验证出栈序列的有效性。值得注意的是，该算法的效率不受栈容量 M 的影响（除了用于验证栈是否溢出），主要取决于序列长度 N。

【代码】

```
def is_valid_pop_sequence(M, N, pop_sequence):
    """
    判断给定的出栈序列是否有效
    :param M: 栈的最大容量
    :param N: 入栈序列的最大长度
    :param pop_sequence: 需要验证的出栈顺序
    :return: YES if the sequence is valid or NO otherwise
    """
    stack = []  # 定义一个栈
    push_num = 1  # 入栈的值从1开始

    for num in pop_sequence:
        # 如果当前栈顶的数字不是目标数字，则开始入栈
        while (not stack or stack[-1] != num) and push_num <= N:
            stack.append(push_num)
            push_num += 1
            # 检查栈是否溢出
            if len(stack) > M:
                return "NO"  # 栈溢出代表当前序列不可能

        # 如果当前栈顶的数字就是目标数字，则将其弹出
        if stack and stack[-1] == num:
            stack.pop()
        else:
            return "NO"  # 处理有可能找不到这个数字的情况，如N为5，出栈值为6

    # 如果所有的数字都处理完
    return "YES"  # 出栈序列是有效的


def main():
    # 接收输入数据,注意第一行输入的是三个端点值
    first_line = input().strip().split()  # 用空格分开
    M = int(first_line[0])  # 栈的最大容量
    N = int(first_line[1])  # 入栈序列的最大长度
    K = int(first_line[2])  # 待测试的序列数量

    # 处理之后每一行的测试序列
    for i in range(K):
        pop_sequence = list(map(int, input().strip().split()))
        print(is_valid_pop_sequence(M, N, pop_sequence))


if __name__ == "__main__":
    main()
```

【整体思路】
模拟入栈出栈过程，通过入栈操作尝试使栈顶元素与期望出栈的元素匹配，同时检查栈容量限制和无法找到目标数字的特殊情况。

关键点解析1

```
while (not stack or stack[-1] != num) and push_num <= N:
```

条件 `(not stack or stack[-1] != num) and push_num <= N` 可以分为两部分：

1. `(not stack or stack[-1] != num)`：这部分检查当前栈是否为空或栈顶元素不是我们要找的数字
   - `not stack`：如果栈为空，这个条件为真
   - `stack[-1] != num`：如果栈不为空，检查栈顶元素是否不等于当前需要出栈的数字
2. `push_num <= N`：检查是否还有数字可以入栈（不超过序列最大长度N）

这个条件模拟了实际操作栈的过程：

- 我们只能从栈顶取出元素
- 入栈必须按照1到N的顺序
- 如果当前栈顶不是我们需要的，只能继续入栈更多的数字，看能否将需要的数字推到栈顶

这个条件是整个算法能够正确验证出栈序列的关键，它保证了我们严格按照栈的"后进先出"原则和"按顺序入栈"的约束进行操作。

关键点解析2

```
push_num = 1  # 入栈的值从1开始
```

1. **模拟按顺序入栈**：
   - 这个问题假设原始入栈序列是从1到N的顺序数字
   - `push_num` 从1开始，每次入栈后加1，确保了我们模拟的是按1,2,3...N的顺序入栈
2. **入栈决策的依据**：
   - 当我们需要某个数字但它不在栈顶时，我们需要知道应该入栈哪些数字
   - `push_num` 告诉我们下一个可以入栈的数字是什么
3. **终止入栈的条件**：
   - 在循环 `while (not stack or stack[-1] != num) and push_num <= N:` 中
   - `push_num <= N` 部分确保我们不会尝试入栈超过N的数字
   - 如果 `push_num > N` 说明我们已经尝试入栈了所有可能的数字，如果还找不到需要的元素，则序列无效

关键点解析3

```
      else:
            # 如果找不到需要的数字，序列无效
            return "NO"
```

这个 else 分支处理的是一种特殊情况：**当我们无法找到需要的数字时。**具体来说，可能会出现这样的情况：

- while 循环尝试通过不断入栈来找到目标数字
- 当 `push_num > N` 时，while 循环结束（已经尝试入栈了所有可能的数字）
- 如果此时栈顶仍然不是目标数字，那么这个出栈序列是无效的