【难度】

士 ★☆☆☆

【解答】

将要排序的栈记为 stack，申请的辅助栈记为 help。在 stack 上执行 pop 操作，弹出的元素记为 cur。

- 如果 cur 小于或等于 help 的栈顶元素，则将 cur 直接压入 help；
- 如果 cur 大于 help 的栈顶元素，则将 help 的元素逐一弹出，逐一压入 stack，直到 cur 小于或等于 help 的栈顶元素，再将 cur 压入 help。

一直执行以上操作，直到 stack 中的全部元素都压入到 help。最后将 help 中的所有元素逐一压入 stack，即完成排序。

```
public static void sortStackByStack(Stack<Integer> stack) {
        Stack<Integer> help = new Stack<Integer>();
        while (!stack.isEmpty()) {
                int cur = stack.pop();
                while (!help.isEmpty() && help.peek() > cur) {
                        stack.push(help.pop());
                }
                help.push(cur);
        }
        while (!help.isEmpty()) {
                stack.push(help.pop());
        }
}
```

用栈来求解汉诺塔问题

【题目】

汉诺塔问题比较经典，这里修改一下游戏规则：现在限制不能从最左侧的塔直接移动到最右侧，也不能从最右侧直接移动到最左侧，而是必须经过中间。求当塔有 N 层的时候，打印最优移动过程和最优移动总步数。

例如，当塔数为两层时，最上层的塔记为 1，最下层的塔记为 2，则打印：

```
Move 1 from left to mid
Move 1 from mid to right
Move 2 from left to mid
```

```
Move 1 from right to mid
Move 1 from mid to left
Move 2 from mid to right
Move 1 from left to mid
Move 1 from mid to right
It will move 8 steps.
```

注意：关于汉诺塔游戏的更多讨论，将在本书递归与动态规划的章节中继续。

【要求】

用以下两种方法解决。

- 方法一：递归的方法；
- 方法二：非递归的方法，用栈来模拟汉诺塔的三个塔。

【难度】

校 ★★★☆

【解答】

方法一：递归的方法。

首先，如果只剩最上层的塔需要移动，则有如下处理：

1．如果希望从“左”移到“中”，打印“Move 1 from left to mid”。

2．如果希望从“中”移到“左”，打印“Move 1 from mid to left”。

3．如果希望从“中”移到“右”，打印“Move 1 from mid to right”。

4．如果希望从“右”移到“中”，打印“Move 1 from right to mid”。

5．如果希望从“左”移到“右”，打印“Move 1 from left to mid”和“Move 1 from mid to right”。

6．如果希望从“右”移到“左”，打印“Move 1 from right to mid”和“Move 1 from mid to left”。

以上过程就是递归的终止条件，也就是只剩上层塔时的打印过程。

接下来，我们分析剩下多层塔的情况。

如果剩下 N 层塔，从最上到最下依次为 1~N，则有如下判断：

1．如果剩下的 N 层塔都在“左”，希望全部移到“中”，则有三个步骤。

1）将 1~N-1 层塔先全部从“左”移到“右”，明显交给递归过程。

2）将第 N 层塔从“左”移到“中”。

3）再将 1~*N*-1 层塔全部从“右”移到“中”，明显交给递归过程。

2．如果把剩下的 *N* 层塔从“中”移到“左”，从“中”移到“右”，从“右”移到“中”，过程与情况 1 同理，一样是分解为三步，在此不再详述。

3．如果剩下的 *N* 层塔都在“左”，希望全部移到“右”，则有五个步骤。

1）将 1~*N*-1 层塔先全部从“左”移到“右”，明显交给递归过程。

2）将第 *N* 层塔从“左”移到“中”。

3）将 1~*N*-1 层塔全部从“右”移到“左”，明显交给递归过程。

4）将第 *N* 层塔从“中”移到“右”。

5）最后将 1~*N*-1 层塔全部从“左”移到“右”，明显交给递归过程。

4．如果剩下的 *N* 层塔都在“右”，希望全部移到“左”，过程与情况 3 同理，一样是分解为五步，在此不再详述。

以上递归过程经过逻辑化简之后的代码请参看如下代码中的 hanoiProblem1 方法。

```
    public int hanoiProblem1(int num, String left, String mid,
                    String right) {
            if (num < 1) {
                    return 0;
            }
            return process(num, left, mid, right, left, right);
    }

    public int process(int num, String left, String mid, String right,
                    String from, String to) {
            if (num == 1) {
                    if (from.equals(mid) || to.equals(mid)) {
                            System.out.println("Move 1 from " + from + " to " + to);
                            return 1;
                    } else {
                            System.out.println("Move 1 from " + from + " to " + mid);
                            System.out.println("Move 1 from " + mid + " to " + to);
                            return 2;
                    }
            }
            if (from.equals(mid) || to.equals(mid)) {
                    String another = (from.equals(left) || to.equals(left)) ? right :
left;
                    int part1 = process(num - 1, left, mid, right, from, another);
                    int part2 = 1;
                    System.out.println("Move " + num + " from " + from + " to " + to);
                    int part3 = process(num - 1, left, mid, right, another, to);
                    return part1 + part2 + part3;
            } else {
                    int part1 = process(num - 1, left, mid, right, from, to);
                    int part2 = 1;
```

```
            System.out.println("Move " + num + " from " + from + " to " + mid);
            int part3 = process(num - 1, left, mid, right, to, from);
            int part4 = 1;
            System.out.println("Move " + num + " from " + mid + " to " + to);
            int part5 = process(num - 1, left, mid, right, from, to);
            return part1 + part2 + part3 + part4 + part5;
        }
    }
```

方法二：非递归的方法——用栈来模拟整个过程。

修改后的汉诺塔问题不能让任何塔从“左”直接移动到“右”，也不能从“右”直接移动到“左”，而是要经过中间。也就是说，实际动作只有 4 个：“左”到“中”、“中”到“左”、“中”到“右”、“右”到“中”。

现在我们把左、中、右三个地点抽象成栈，依次记为 LS、MS 和 RS。最初所有的塔都在 LS 上。那么如上 4 个动作就可以看作是：某一个栈（from）把栈顶元素弹出，然后压入到另一个栈里（to），作为这一个栈（to）的栈顶。

例如，如果是 7 层塔，在最初时所有的塔都在 LS 上，LS 从栈顶到栈底就依次是 1~7，如果现在发生了“左”到“中”的动作，这个动作对应的操作是 LS 栈将栈顶元素 1 弹出，然后 1 压入到 MS 栈中，成为 MS 的栈顶。其他的操作同理。

一个动作能发生的先决条件是不违反小压大的原则。

from 栈弹出的元素 num 如果想压入到 to 栈中，那么 num 的值必须小于当前 to 栈的栈顶。

还有一个原则不是很明显，但也是非常重要的，叫相邻不可逆原则，解释如下：

1．我们把四个动作依次定义为：L->M、M->L、M->R 和 R->M。

2．很明显，L->M 和 M->L 过程互为逆过程，M->R 和 R->M 互为逆过程。

3．在修改后的汉诺塔游戏中，如果想走出最少步数，那么任何两个相邻的动作都不是互为逆过程的。举个例子：如果上一步的动作是 L->M，那么这一步绝不可能是 M->L，直观地解释为：你上一步把一个栈顶数从“左”移动到“中”，这一步为什么又要移回去呢？这必然不是取得最小步数的走法。同理，M->R 动作和 R->M 动作也不可能相邻发生。

有了小压大和相邻不可逆原则后，可以推导出两个十分有用的结论——非递归的方法核心结论：

1．游戏的第一个动作一定是 L->M，这是显而易见的。

2．在走出最少步数过程中的任何时刻，四个动作中只有一个动作不违反小压大和相邻不可逆原则，另外三个动作一定都会违反。

对于结论 2，现在进行简单的证明。

因为游戏的第一个动作已经确定是 L->M，则以后的每一步都会有前一步的动作。

假设前一步的动作是 L->M：

1．根据小压大原则，L->M 的动作不会重复发生。

2．根据相邻不可逆原则，M->L 的动作也不该发生。

3．根据小压大原则，M->R 和 R->M 只会有一个达标。

假设前一步的动作是 M->L：

1．根据小压大原则，M->L 的动作不会重复发生。

2．根据相邻不可逆原则，L->M 的动作也不该发生。

3．根据小压大原则，M->R 和 R->M 只会有一个达标。

假设前一步的动作是 M->R：

1．根据小压大原则，M->R 的动作不会重复发生。

2．根据相邻不可逆原则，R->M 的动作也不该发生。

3．根据小压大原则，L->M 和 M->L 只会有一个达标。

假设前一步的动作是 R->M：

1．根据小压大原则，R->M 的动作不会重复发生。

2．根据相邻不可逆原则，M->R 的动作也不该发生。

3．根据小压大原则，L->M 和 M->L 只会有一个达标。

综上所述，每一步只会有一个动作达标。那么只要每走一步都根据这两个原则考查所有的动作就可以，哪个动作达标就走哪个动作，反正每次都只有一个动作满足要求，按顺序走下来即可。

非递归的具体过程请参看如下代码中的 hanoiProblem2 方法。

```
public enum Action {
        No, LToM, MToL, MToR, RToM
}

public int hanoiProblem2(int num, String left, String mid, String right) {
        Stack<Integer> lS = new Stack<Integer>();
        Stack<Integer> mS = new Stack<Integer>();
        Stack<Integer> rS = new Stack<Integer>();
        lS.push(Integer.MAX_VALUE);
        mS.push(Integer.MAX_VALUE);
        rS.push(Integer.MAX_VALUE);
        for (int i = num; i > 0; i--) {
                lS.push(i);
        }
```

```
        Action[] record = { Action.No };
        int step = 0;
        while (rS.size() != num + 1) {
            step += fStackTotStack(record, Action.MToL, Action.LToM, lS, mS,
                                   left, mid);
            step += fStackTotStack(record, Action.LToM, Action.MToL, mS, lS,
                                   mid, left);
            step += fStackTotStack(record, Action.RToM, Action.MToR, mS, rS,
                                   mid, right);
            step += fStackTotStack(record, Action.MToR, Action.RToM, rS, mS,
                                   right, mid);
        }
        return step;
    }

    public static int fStackTotStack(Action[] record, Action preNoAct,
            Action nowAct, Stack<Integer> fStack, Stack<Integer> tStack,
            String from, String to) {
        if (record[0] != preNoAct && fStack.peek() < tStack.peek()) {
            tStack.push(fStack.pop());
            System.out.println("Move " + tStack.peek() + " from " + from + "
to " + to);
            record[0] = nowAct;
            return 1;
        }
        return 0;
    }
```

生成窗口最大值数组

【题目】

有一个整型数组 arr 和一个大小为 w 的窗口从数组的最左边滑到最右边，窗口每次向右边滑一个位置。

例如，数组为[4,3,5,4,3,3,6,7]，窗口大小为 3 时：

```
[4  3  5] 4  3  3  6  7          窗口中最大值为 5
 4 [3  5  4] 3  3  6  7          窗口中最大值为 5
 4  3 [5  4  3] 3  6  7          窗口中最大值为 5
 4  3  5 [4  3  3] 6  7          窗口中最大值为 4
 4  3  5  4 [3  3  6] 7          窗口中最大值为 6
 4  3  5  4  3 [3  6  7]         窗口中最大值为 7
```

如果数组长度为 n，窗口大小为 w，则一共产生 n-w+1 个窗口的最大值。