```
                if (arr[i] > ends[m]) {
                        l = m + 1;
                } else {
                        r = m - 1;
                }
            }
            right = Math.max(right, l);
            ends[l] = arr[i];
            dp[i] = l + 1;
        }
        return dp;
}
```

时间复杂度 $O(N\log N)$方法的整个过程请参看如下代码中的 lis2 方法。

```
public int[] lis2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return null;
        }
        int[] dp = getdp2(arr);
        return generateLIS(arr, dp);
}
```

汉诺塔问题

【题目】

给定一个整数 n，代表汉诺塔游戏中从小到大放置的 n 个圆盘，假设开始时所有的圆盘都放在左边的柱子上，想按照汉诺塔游戏的要求把所有的圆盘都移到右边的柱子上。实现函数打印最优移动轨迹。

【举例】

n=1 时，打印：

move from left to right

n=2 时，打印：

move from left to mid

move from left to right

move from mid to right

【进阶题目】

给定一个整型数组 arr，其中只含有 1、2 和 3，代表所有圆盘目前的状态，1 代表左柱，2 代表中柱，3 代表右柱，arr[i]的值代表第 i+1 个圆盘的位置。比如，arr=[3,3,2,1]，代表第 1 个圆盘在右柱上、第 2 个圆盘在右柱上、第 3 个圆盘在中柱上、第 4 个圆盘在左柱上。如果 arr 代表的状态是最优移动轨迹过程中出现的状态，返回 arr 这种状态是最优移动轨迹中的第几个状态。如果 arr 代表的状态不是最优移动轨迹过程中出现的状态，则返回-1。

【举例】

arr=[1,1]。两个圆盘目前都在左柱上，也就是初始状态，所以返回 0。

arr=[2,1]。第一个圆盘在中柱上、第二个圆盘在左柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹的第 1 步，所以返回 1。

arr=[3,3]。第一个圆盘在右柱上、第二个圆盘在右柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹的第 3 步，所以返回 3。

arr=[2,2]。第一个圆盘在中柱上、第二个圆盘在中柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹从来不会出现的状态，所以返回-1。

【进阶题目要求】

如果 arr 长度为 N，请实现时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$的方法。

【难度】

校 ★★★☆

【解答】

原问题。假设有 from 柱子、mid 柱子和 to 柱子，都在 from 的圆盘 1~i 完全移动到 to，最优过程为：

步骤 1 为圆盘 1~i-1 从 from 移动到 mid。

步骤 2 为单独把圆盘 i 从 from 移动到 to。

步骤 3 为把圆盘 1~i-1 从 mid 移动到 to。如果圆盘只有 1 个，直接把这个圆盘从 from 移动到 to 即可。

打印最优移动轨迹的方法参见如下代码中的 hanoi 方法。

```
public void hanoi(int n) {
        if (n > 0) {
```

```
                func(n, "left", "mid", "right");
            }
        }

        public void func(int n, String from, String mid, String to) {
            if (n == 1) {
                System.out.println("move from " + from + " to " + to);
            } else {
                func(n - 1, from, to, mid);
                func(1, from, mid, to);
                func(n - 1, mid, from, to);
            }
        }
```

进阶题目。首先求都在 from 柱子上的圆盘 1~i，如果都移动到 to 上的最少步骤数，假设为 $S(i)$。根据上面的步骤，$S(i)$=步骤 1 的步骤总数+1+步骤 3 的步骤总数=$S(i-1)+1+S(i-1)$，$S(1)=1$。所以 $S(i)+1=2(S(i-1)+1)$，$S(1)+1==2$。根据等比数列求和公式得到 $S(i)+1=2^i$，所以 $S(i)=2^{i-1}$。

对于数组 arr 来说，arr[N-1]表示最大圆盘 N 在哪个柱子上，情况有以下三种：

- 圆盘 N 在左柱上，说明步骤 1 或者没有完成，或者已经完成，需要考查圆盘 1~N-1 的状况。
- 圆盘 N 在右柱上，说明步骤 1 已经完成，起码走完了 2^{N-1}-1 步。步骤 2 也已经完成，起码又走完了 1 步，所以当前状况起码是最优步骤的 2^{N-1} 步，剩下的步骤怎么确定还得继续考查圆盘 1~N-1 的状况。
- 圆盘 N 在中柱上，这是不可能的，最优步骤中不可能让圆盘 N 处在中柱上，直接返回-1。

所以整个过程可以总结为：对圆盘 1~i 来说，如果目标为从 from 到 to，那么情况有三种：

- 圆盘 i 在 from 上，需要继续考查圆盘 1~i-1 的状况，圆盘 1~i-1 的目标为从 from 到 mid。
- 圆盘 i 在 to 上，说明起码走完了 2^{i-1} 步，剩下的步骤怎么确定还得继续考查圆盘 1~i-1 的状况，圆盘 1~i-1 的目标为从 mid 到 to。
- 圆盘 i 在 mid 上，直接返回-1。

整个过程参看如下代码中的 step1 方法。

```
        public int step1(int[] arr) {
            if (arr == null || arr.length == 0) {
                return -1;
            }
```

```
        return process(arr, arr.length - 1, 1, 2, 3);
}

public int process(int[] arr, int i, int from, int mid, int to) {
        if (i == -1) {
                return 0;
        }
        if (arr[i] != from && arr[i] != to) {
                return -1;
        }
        if (arr[i] == from) {
                return process(arr, i - 1, from, to, mid);
        } else {
                int rest = process(arr, i - 1, mid, from, to);
                if (rest == -1) {
                        return -1;
                }
                return (1 << i) + rest;
        }
}
```

step1 方法是递归函数，递归最多调用 N 次，并且每步的递归函数再调用递归函数的次数最多一次。在每个递归过程中，除去递归调用的部分，剩下过程的时间复杂度为 $O(1)$，所以 step1 方法的时间复杂度为 $O(N)$。但是因为递归函数需要函数栈的关系，step1 方法的额外空间复杂度为 $O(N)$，所以为了达到题目的要求，需要将整个过程改成非递归的方法，具体请参看如下代码中的 step2 方法。

```
public int step2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return -1;
        }
        int from = 1;
        int mid = 2;
        int to = 3;
        int i = arr.length - 1;
        int res = 0;
        int tmp = 0;
        while (i >= 0) {
                if (arr[i] != from && arr[i] != to) {
                        return -1;
                }
                if (arr[i] == to) {
                        res += 1 << i;
                        tmp = from;
                        from = mid;
                } else {
                        tmp = to;
                        to = mid;
                }
```

```
                mid = tmp;
                i--;
            }
            return res;
        }
```

最长公共子序列问题

【题目】

给定两个字符串 str1 和 str2，返回两个字符串的最长公共子序列。

【举例】

str1="1A2C3D4B56"，str2="B1D23CA45B6A"。

"123456"或者"12C4B6"都是最长公共子序列，返回哪一个都行。

【难度】

尉 ★★☆☆

【解答】

本题是非常经典的动态规划问题，先来介绍求解动态规划表的过程。如果 str1 的长度为 M，str2 的长度为 N，生成大小为 $M \times N$ 的矩阵 dp，行数为 M，列数为 N。dp[i][j]的含义是 str1[0..i]与 str2[0..j]的最长公共子序列的长度。从左到右，再从上到下计算矩阵 dp。

1．矩阵 dp 第一列即 dp[0..M-1][0]，dp[i][0]的含义是 str1[0..i]与 str2[0]的最长公共子序列长度。str2[0]只有一个字符，所以 dp[i][0]最大为 1。如果 str1[i]==str2[0]，令 dp[i][0]=1，一旦 dp[i][0]被设置为 1，之后的 dp[i+1..M-1][0]也都为 1。比如，str1[0..M-1]="ABCDE"，str2[0]="B"。str1[0]为"A"，与 str2[0]不相等，所以 dp[0][0]=0。str1[1]为"B"，与 str2[0]相等，所以 str1[0..1]与 str2[0]的最长公共子序列为"B"，令 dp[1][0]=1。之后的 dp[2..4][0]肯定都是 1，因为 str[0..2]、str[0..3]和 str[0..4]与 str2[0]的最长公共子序列肯定有"B"。

2．矩阵 dp 第一行即 dp[0][0..N-1]与步骤 1 同理，如果 str1[0]==str2[j]，则令 dp[0][j]=1，一旦 dp[0][j]被设置为 1，之后的 dp[0][j+1..N-1]也都为 1。

3．对其他位置(i,j)，dp[i][j]的值只可能来自以下三种情况：

- 可能是 dp[i-1][j]，代表 str1[0..i-1]与 str2[0..j]的最长公共子序列长度。比如，