

剑指Offer(第2版)

剑指Offer(第2版)

A1. 动态规划DP

A2. 递归

Q3. 数组中重复的数字：遍历计数、排序找重、原地交换

Q4. 二维数组中的查找：暴力求解、类二叉查找树

Q5. 替换空格：遍历替换、原地修改

Q6. 从尾到头打印链表：栈、递归

Q7. 重建二叉树 - 递归、迭代

Q9. 用两个栈实现队列：栈

Q10 - I. 斐波那契数列：递归、动态规划DP

Q10 - II. 青蛙跳台阶问题：递归、动态规划DP

Q11. 旋转数组的最小数字：遍历寻找、二分法

Q12. 矩阵中的路径：DFS+剪枝

Q13. 机器人的运动范围：DFS（数位和增量公式、可达性分析）、BFS

Q14 - I. 剪绳子：数学推导、贪心、动态规划

Q14 - II. 剪绳子 II：循环求余、快速求余

Q15. 二进制中1的个数：巧用 $n \& (n - 1)$ 、逐位判断

Q16. 数值的整数次方：循环求幂、二进制快速幂、二分法快速幂

Q17. 打印从1到最大的n位数：循环打印、大数打印（递归）

Q18. 删除链表的节点：遍历对比删除

Q19. 正则表达式匹配：动态规划

Q20. 表示数值的字符串：逐位判断、有限状态自动机

Q21. 调整数组顺序使奇数位于偶数前面：遍历找奇偶并用数组存储、双指针交换

Q22. 链表中倒数第k个节点：先求长度再顺序找、双指针寻找

Q24. 反转链表：栈、数组、双指针、递归

Q25. 合并两个排序的链表：遍历比较合并、递归

Q26. 树的子结构：两个递归

Q27. 二叉树的镜像：递归、栈

Q28. 对称的二叉树：BFS、递归

Q29. 顺时针打印矩阵：设定四边界

Q31. 栈的压入、弹出序列：双指针模拟、栈辅助

Q32 - I. 从上到下打印二叉树：队列 BFS

Q32 - II. 从上到下打印二叉树 II：队列 BFS、递归 BFS

Q32 - III. 从上到下打印二叉树 III：队列 BFS、递归 BFS

Q33. 二叉搜索树的后序遍历序列

注O(N) O(N)：先时间复杂度，再空间复杂度，全文如此。

A1. 动态规划DP

1. 状态定义

2. 转移方程

3. 初始化
4. 返回值

A2. 递归

1. 递推参数
2. 终止条件
3. 递推工作
4. 返回值

Q3. 数组中重复的数字：遍历计数、排序找重、原地交换

在一个长度为 n 的数组 `nums` 里的所有数字都在 $0 \sim n-1$ 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

```
1  输入：
2  [2, 3, 1, 0, 2, 5, 3]
3  输出：2 或 3
```

解：

1. 遍历计数 - 遍历 `nums[]`，利用 `countArray[]` 记录数出现的次数。

```
1  /**
2   * 遍历 nums[]，利用 countArray[] 记录数出现的次数
3   * @author PAN
4   * @param nums int[]
5   * @return repeatNumber: 第一个重复出现的数字
6   */
7  public static int findRepeatNumber(int[] nums) {
8      int[] countArray = new int[nums.length];
9      int repeatNumber = -1;
10     for(int num : nums) {
11         countArray[num]++;
12         if(countArray[num] > 1) {
13             repeatNumber = num;
14             break;
15         }
16     }
17     return repeatNumber;
18 }
```

2. 排序后再找重 - 排序 `nums[]`，再遍历找到第一个重复数字。

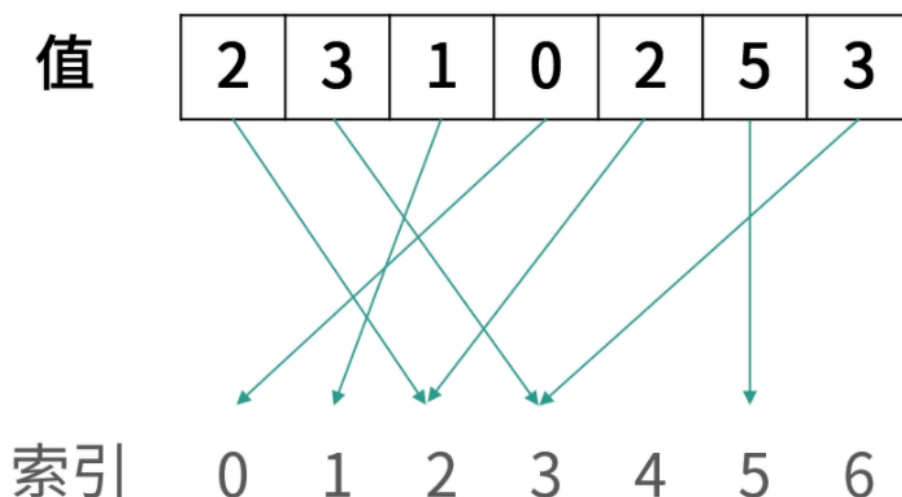
```
1  /**
2   * 排序 nums[]，再遍历找到第一个重复数字
3   * @author PAN
4   * @param nums int[]
5   * @return repeatNumber: 第一个重复出现的数字
```

```

6  */
7  public static int findRepeatNumber2(int[] nums) {
8      int repeatNumber = -1;
9      Arrays.sort(nums);
10     int record = nums[0];
11     for(int i = 1; i < nums.length; i++) {
12         if(nums[i] == record) {
13             repeatNumber = record;
14             break;
15         } else {
16             record = nums[i];
17         }
18     }
19     return repeatNumber;
20 }

```

3. 原地交换 - 遍历中，第一次遇到数字 xx 时，将其交换至索引 xx 处，而当第二次遇到数字 xx 时，一定有 $\text{nums}[x] = x$ ，此时即可得到一组重复数字。



- ∴ 在一个长度为 n 的数组 `nums` 里的所有数字都在 $0 \sim n-1$ 的范围内
- ∴ 数组元素的索引和值是一对多的关系，因此可建立索引和值的映射

```

1  /**
2   * 原地交换
3   * 遍历中，第一次遇到数字 xx 时，将其交换至索引 xx 处；
4   * 而当第二次遇到数字 xx 时，一定有  $\text{nums}[x] = x$ ，此时即可得到一组重复数字。
5   * @author 网友
6   */
7  public static int findRepeatNumber0(int[] nums) {
8      int i = 0;
9      while(i < nums.length) {

```

```

10         if(nums[i] == i) {
11             i++;
12             continue;
13         }
14         if(nums[nums[i]] == nums[i]) return nums[i];
15         int tmp = nums[i];
16         nums[i] = nums[tmp];
17         nums[tmp] = tmp;
18     }
19     return -1;
20 }

```

Q4. 二维数组中的查找：暴力求解、类二叉查找树

在一个 $n * m$ 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```

1  示例：
2
3  现有矩阵 matrix 如下：
4
5  [
6      [1,   4,   7, 11, 15],
7      [2,   5,   8, 12, 19],
8      [3,   6,   9, 16, 22],
9      [10, 13, 14, 17, 24],
10     [18, 21, 23, 26, 30]
11 ]
12
13 给定 target = 5, 返回 true。
14 给定 target = 20, 返回 false。

```

解：

1. 暴力求解 $O(N + M)$ - 双重循环遍历整个数组，与所有元素一一比较。

```

1  /**
2   * 暴力求解
3   * @author PAN
4   * @param matrix int[][]
5   * @param target int
6   * @return true/false: 查找结果
7   * @time  $O(N * M)$ 
8   */
9  public static boolean findNumberIn2DArray(int[][] matrix, int
target) {
10     for (int i = 0; i < matrix.length; i++) {

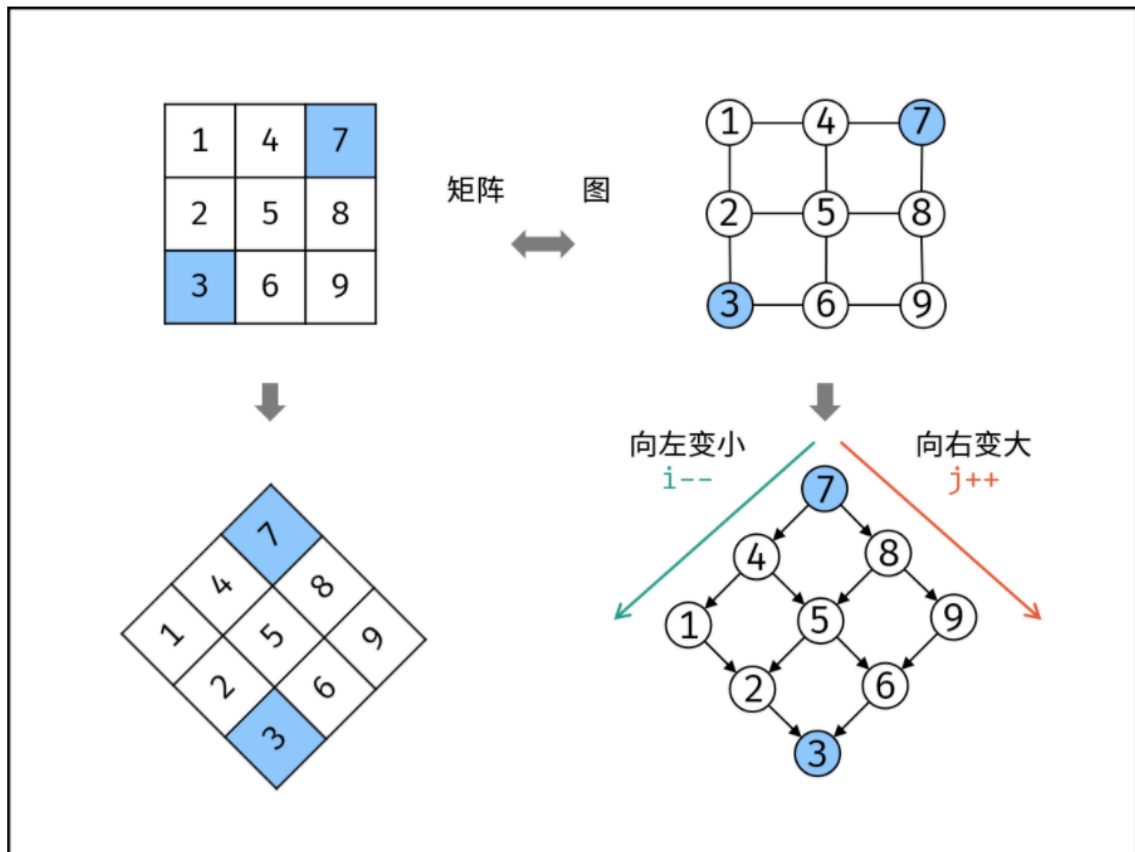
```

```

11         for (int j = 0; j < matrix[i].length; j++) {
12             if (target == matrix[i][j]) {
13                 return true;
14             }
15         }
16     }
17     return false;
18 }

```

2. **类二叉查找树 $O(N + M)$** - 将矩阵逆时针旋转 45° ，并将其转化为图形式，发现其类似于二叉搜索树，即对于每个元素，其左分支元素更小、右分支元素更大。从二维数组右上角元素开始与 target 比较，target 大则向下比较，target 小则向左比较。



```

1  /**
2   * 线性查找：从二维数组右上角元素开始与 target 比较，target 大则向下比较，
   target 小则向左比较
3   * @author 网友
4   * @param matrix 二维数组
5   * @param target 查找目标
6   * @return true/false
7   * @time O(N + M)
8   */
9  public static boolean findNumberIn2DArray2(int[][] matrix, int
target) {
10     if (matrix == null || matrix.length == 0 || matrix[0].length
    == 0) {
11         return false;

```

```

12     } else {
13         int i = 0, j = matrix[0].length - 1;
14         do {
15             if (target < matrix[i][j]) {
16                 j--;
17             } else if (target == matrix[i][j]) {
18                 return true;
19             } else {
20                 i++;
21             }
22         } while (i < matrix.length && j > -1);
23
24         return false;
25     }
26 }

```

Q5. 替换空格：遍历替换、原地修改

请实现一个函数，把字符串 *s* 中的每个空格替换成"%20"。

```

1  输入: s = "We are happy."
2  输出: "We%20are%20happy."

```

解：

1. 遍历查找替换 $O(N)$ $O(N)$ - 逐个字符遍历字符串查找空格，字符串不为空格追加到新字符串，为空格追加"%20"。

```

1  /**
2   * 遍历查找替换：逐个字符遍历字符串查找空格，字符串不为空格追加到新字符串，为空格追加"%20"
3   * @author PAN
4   * @param s 原字符串
5   * @return 替换后字符串
6   * @time  $O(N)$ 
7   * @space  $O(N)$ 
8   */
9  public static String replaceSpace(String s) {
10     StringBuilder newS = new StringBuilder();
11     for (int i = 0; i < s.length(); i++) {
12         char c = s.charAt(i);
13         if (c != ' ') {
14             newS.append(c);
15         } else {
16             newS.append("%20");
17         }
18     }
19 }

```

```
20     return newS.toString();
21 }
```

2. 调用 `replace()`

```
1 public static String replaceSpace2(String s) {
2     return s.replace(" ", "%20");
3 }
```

3. 原地修改 $O(N)$ $O(1)$ - 不使用新字符串来存储，但在 Java Python 中不行，因它们字符串建立后不可改变，在 C++ 中可以通过两个指针来原地修改

Q6. 从尾到头打印链表：栈、递归

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

```
1 输入：head = [1,3,2]
2 输出：[2,3,1]
```

解：

1. 栈 $O(N)$ $O(N)$ - 先入后出实现从尾到头打印。

```
1  /**
2   * 栈：先入后出实现从尾到头打印
3   * @author PAN
4   * @param head 单链表头
5   * @return 反转后单链表（以数组形式）
6   * @time  $O(N)$ 
7   * @space  $O(N)$ 
8   */
9  public static int[] reversePrint(ListNode head) {
10     Stack<ListNode> s = new Stack<ListNode>();
11     while (head != null) {
12         s.push(head);
13         head = head.next;
14     }
15     int len = s.size();
16     int[] array = new int[len];
17     for (int i = 0; i < len; i++) {
18         array[i] = s.pop().val;
19     }
20     return array;
21 }
```

2. 递归 $O(N)$ $O(N)$ 。

Q7. 重建二叉树 - 递归、迭代

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

```
1  例如，给出
2
3  前序遍历 preorder = [3,9,20,15,7]
4  中序遍历 inorder = [9,3,15,20,7]
5
6  返回如下的二叉树：
7
8      3
9     / \
10    9  20
11   /  \
12  15   7
```

解：

1. 递归 $O(N)$ $O(N)$

分治算法解析：

- 递推参数：根节点在前序遍历的索引 root、子树在中序遍历的左边界 left、子树在中序遍历的右边界 right；
- 终止条件：当 $left > right$ ，代表已经越过叶节点，此时返回 null；
- 递推工作：
 - 建立根节点 node：节点值为 `preorder[root]`；
 - 划分左右子树：查找根节点在中序遍历 `inorder` 中的索引 `i`；
为了提升效率，本文使用哈希表 `dic` 存储中序遍历的值与索引的映射，查找操作的时间复杂度为 $O(1)$ ；
 - 构建左右子树：开启左右子树递归；几个索引值的确定容易出错！！

	根节点索引	中序遍历左边界	中序遍历右边界
左子树	$root + 1$	left	$i - 1$
右子树	$i - left + root + 1$	$i + 1$	right

- $i - left + root + 1$ 含义为 根节点索引 + 左子树长度 + 1

- 返回值：回溯返回 node，作为上一层递归中根节点的左 / 右子节点；

```
1  /**
2   * 递归重建二叉树
3   * @author 网友
4   * @param root 前序中根索引
5   * @param left 前序中左子树索引
6   * @param right 前序中右子树索引
7   * @return 树结点
```



```

8      * @time O(N)
9      * @space O(N)
10     */
11     TreeNode recur(int root, int left, int right) {
12         if (left > right) return null;
13         TreeNode node = new TreeNode(preorder[root]);
14         int i = dic.get(preorder[root]);
15         node.left = recur(root + 1, left, i - 1);
16         node.right = recur(i - left + root + 1, i + 1, right);
17         return node;
18     }

```

2. 迭代 O(N) O(N), 不好理解

- 对于前序遍历中的任意两个连续节点 uu 和 vv, 根据前序遍历的流程, 我们可以知道 uu 和 vv 只有两种可能的关系:
 - vv 是 uu 的左儿子。这是因为在遍历到 uu 之后, 下一个遍历的节点就是 uu 的左儿子, 即 vv;
 - uu 没有左儿子, 并且 vv 是 uu 的某个祖先节点 (或者 uu 本身) 的右儿子。
- 算法:
 - 用一个栈和一个指针辅助进行二叉树的构造。初始时栈中存放了根节点 (前序遍历的第一个节点), 指针指向中序遍历的第一个节点;
 - 我们依次枚举前序遍历中除了第一个节点以外的每个节点。如果 index 恰好指向栈顶节点, 那么我们不断地弹出栈顶节点并向右移动 index, 并将当前节点作为最后一个弹出的节点的右儿子; 如果 index 和栈顶节点不同, 我们将当前节点作为栈顶节点的左儿子;
 - 无论是哪一种情况, 我们最后都将当前的节点入栈。

Q9. 用两个栈实现队列: 栈

用两个栈实现一个队列。队列的声明如下, 请实现它的两个函数 appendTail 和 deleteHead, 分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素, deleteHead 操作返回 -1)

```

1  示例 1:
2
3  输入:
4  ["CQueue","appendTail","deleteHead","deleteHead"]
5  [[],[3],[],[[]]]
6  输出: [null,null,3,-1]
7
8  示例 2:
9
10 输入:
11 ["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]
12 [[],[],[5],[2],[],[[]]]
13 输出: [null,-1,null,null,5,2]

```

解:

1. 一栈负责进，一栈复杂出 $O(N)$ $O(N)$

```

1  /**
2   * 一栈负责进，一栈复杂出
3   * @author PAN
4   * @return 队列头
5   * @time  $O(N)$ 
6   * @space  $O(N)$ 
7   */
8  public int deleteHead() {
9      if (s2.size() == 0) {
10         if (s1.size() == 0) return -1;
11         else {
12             while (!s1.isEmpty()) {
13                 s2.push(s1.pop());
14             }
15         }
16     }
17     return s2.pop();
18 }

```

Q10 - I. 斐波那契数列：递归、动态规划DP

写一个函数，输入 n ，求斐波那契（Fibonacci）数列的第 n 项（即 $F(N)$ ）。斐波那契数列的定义如下：

$$F(0) = 0, F(1) = 1$$

$$F(N) = F(N - 1) + F(N - 2), \text{ 其中 } N > 1.$$

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 $1e9+7$ (1000000007)，如计算初始结果为：1000000008，请返回 1。

```
1  示例 1:
2
3  输入: n = 2
4  输出: 1
5
6  示例 2:
7
8  输入: n = 5
9  输出: 5
```

解:

1. 递归 - 效率低, 存在大量重复计算;

```
1  /**
2   * 递归求解 - 将斐波那契公式转换为递归形式
3   * @author PAN
4   * @param n 数列第 n 项
5   * @return 斐波那契值
6   */
7  public static long fib(int n) {
8      if (n == 0) return 0;
9      else if (n == 1) return 1;
10     else {
11         if (fib(n - 1) + fib(n - 2) > 1000000007) return (fib(n -
12         1) + fib(n - 2)) % 1000000007;
13         else return fib(n - 1) + fib(n - 2);
14     }
15 }
```

2. 动态规划 $O(N)$ $O(N)$ - 某一项等于前两项之和, 用循环依次计算。【注意!!!】先求余与最后求余返回结果一致, 但先求可以防止 int 溢出

```
1  /**
2   * 动态规划 - 某一项等于前两项之和, 用循环依次计算。【注意!!!】先求余与最后
3   * 求余返回结果一致, 但先求可以防止 int 溢出
4   * @author PAN
5   * @param n
6   * @return
7   * @time  $O(N)$ 
8   * @space  $O(1)$ 
9   */
10 public static int fib2(int n) {
11     int a = 0, b = 1;
12     switch (n) {
13         case 0: return 0;
14         case 1: return 1;
15         default: {
16             int tmp;
```

```

16         for (int i = 1; i < n; i++) {
17             tmp = (a + b) % 1000000007;
18             a = b;
19             b = tmp;
20         }
21         return b;
22     }
23 }
24 }
25
26 /**
27  * 动态规划 - 精简代码
28  * @author 网友
29  * @param n
30  * @return
31  */
32 public static int fib3(int n) {
33     int a = 0, b = 1, sum;
34     for(int i = 0; i < n; i++){
35         sum = (a + b) % 1000000007;
36         a = b;
37         b = sum;
38     }
39     return a;
40 }

```

Q10 - II. 青蛙跳台阶问题：递归、动态规划DP

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

答案需要取模 $1e9+7$ (1000000007)，如计算初始结果为：1000000008，请返回 1。

```

1  示例 1:
2
3  输入: n = 2
4  输出: 2
5
6  示例 2:
7
8  输入: n = 7
9  输出: 21
10
11 示例 3:
12
13 输入: n = 0
14 输出: 1

```

解：

1. 递归

2. 动态规划

- 此题就是斐波那契数列的变形（区别在于初始值不一样），设跳上 n 级台阶有 $f(n)$ 种跳法。在所有跳法中，青蛙的最后一步只有两种情况：跳上 1 级或 2 级台阶。
 - 当为 1 级台阶：剩 $n-1$ 个台阶，此情况共有 $f(n-1)$ 种跳法；
 - 当为 2 级台阶：剩 $n-2$ 个台阶，此情况共有 $f(n-2)$ 种跳法。
 - $f(n)$ 为以上两种情况之和，即 $f(n)=f(n-1)+f(n-2)$ 。

```
1 public static int numWays(int n) {
2     int a = 1, b = 1, sum;
3     for (int i = 0; i < n; i++) {
4         sum = (a + b) % 1000000007;
5         a = b;
6         b = sum;
7     }
8     return a;
9 }
```

Q11. 旋转数组的最小数字：遍历寻找、二分法

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 $[3,4,5,1,2]$ 为 $[1,2,3,4,5]$ 的一个旋转，该数组的最小值为 1。

```
1 示例 1：
2
3 输入：[3,4,5,1,2]
4 输出：1
5
6 示例 2：
7
8 输入：[2,2,2,0,1]
9 输出：0
```

解：

1. 遍历直到第一个变小的数 $O(N)$ $O(1)$ - 首先用 min 记录数组中第一个元素的值，之后遍历数组，一一与 min 比较，第一个比 min 小的即是结果；

```
1 /**
2  * 遍历直到第一个变小的数
3  * @author PAN
4  * @param numbers 两个非递减数列构成的数组
5  * @return min
6  * @time O(N)
7  * @space O(1)
```

```

8  */
9  public static int minArray(int[] numbers) {
10     int min = numbers[0];
11     for (int i = 1; i < numbers.length; i++) {
12         if (numbers[i] < min) {
13             min = numbers[i];
14             break;
15         }
16     }
17     return min;
18 }

```

2. 二分法 $O(\log N)$ $O(1)$ - 因为整个数组是由两个非递减数列构成的，所以可以用二分来缩小范围

1) $i = 0, j = \text{length} - 1, m = (i + j) / 2$;

2) 比较索引为m和j数组元素大小，大于则 $i = m + 1$ ，等于则 $j--$ ，小于则 $j = m$ （或者调用上述遍历找min）

```

1  /**
2   * 二分法：因为整个数组是由两个非递减数列构成的，所以可以用二分来缩小范围
3   * @author 网友
4   * @param numbers 两个非递减数列构成的数组
5   * @return min
6   * @time  $O(\log N)$ 
7   * @space  $O(1)$ 
8   */
9  public static int minArray2(int[] numbers){
10     int i = 0, j = numbers.length - 1;
11     int m = 0;
12     while (i < j) {
13         m = (i + j) / 2;
14         if (numbers[m] > numbers[j]) i = m + 1;
15         else if (numbers[m] == numbers[j]) j--;
16         else {
17             int min = numbers[i];
18             for (int k = i + 1; k < m + 1; k++) {
19                 if (numbers[k] < min){
20                     min = numbers[k];
21                     break;
22                 }
23             }
24             return min;
25         }
26     }
27     return numbers[i];
28 }

```

Q12. 矩阵中的路径：DFS+剪枝

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

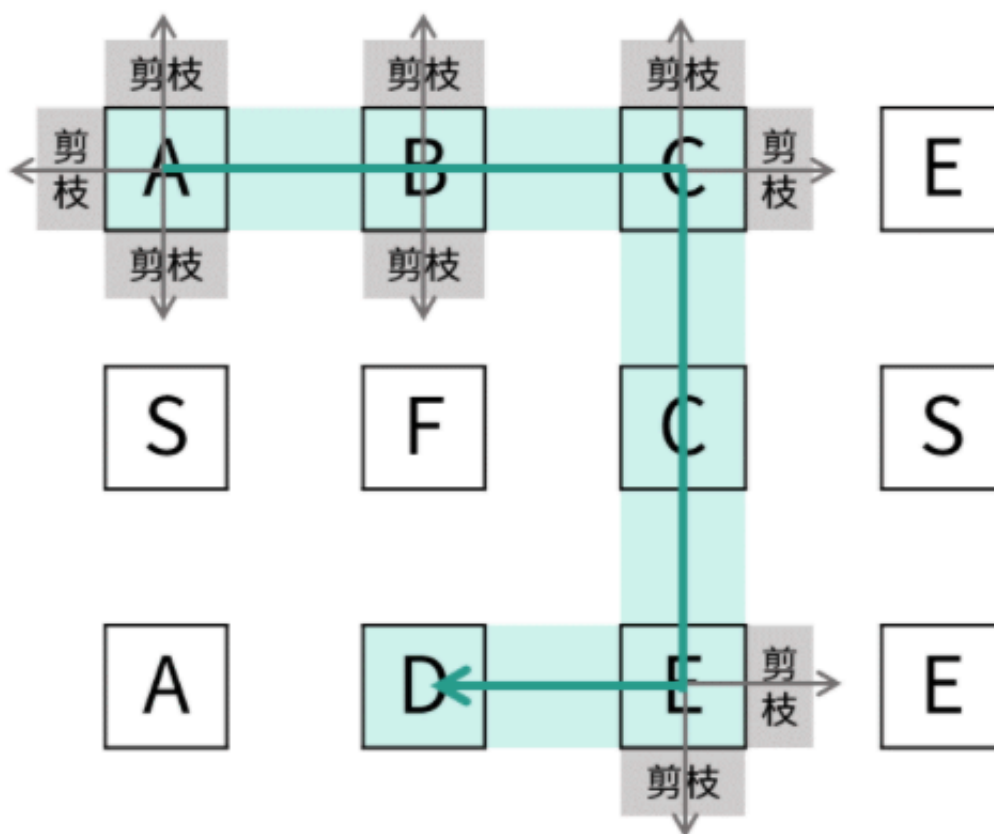
```
[[ "a", "b", "c", "e"],  
[ "s", "f", "c", "s"],  
[ "a", "d", "e", "e"]]
```

但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

```
1  示例 1：  
2  
3  输入：board = [ ["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"] ], word =  
   "ABCCED"  
4  输出：true  
5  
6  示例 2：  
7  
8  输入：board = [ ["a","b"], ["c","d"] ], word = "abcd"  
9  输出：false
```

解：

1. DFS + 剪枝 $O(3^K \cdot MN)$ $O(K)$:



word = " A B C C E D "

1) DFS暴力遍历矩阵中所有字符串可能性，通过递归先朝一个方向搜到底，再回溯至上个节点，沿另一个方向搜索，以此类推；

2) 在搜索中，遇到这条路不可能和目标字符串匹配成功的情况，则立即返回，称可行性剪枝；

o DFS解析：

- 递归参数：当前元素在矩阵 board 中的行索引 i 和 j，当前目标字符在 word 中的索引 k。
- 终止条件：
 1. 返回 false（满足任一一一个）：（1）行或列索引越界（2）当前矩阵元素与目标字符不同（3）当前矩阵元素已访问
 2. 返回 true: `k = len(word) - 1`，即字符串 word 已全部匹配
- 递推工作：
 1. 标记当前矩阵元素：将 `board[i][j]` 修改为空 `'\0'`，代表已访问；
 2. 搜索下一单元格：上右下左；
 3. 还原当前矩阵元素：将 `board[i][j]` 还原为初始值；
- 返回值：是否搜索到目标字符串
- 时间复杂度 $O(3^K * MN)$ ：字符串长度为K，搜索中每个字符有四个方向选
- 空间复杂度 $O(K)$ ：搜索过程中的递归深度不超过K，系统因函数调用累计使用栈空间 $O(K)$


```

1  boolean existPath(char[][] board, String word) {
2      char[] wordArray = word.toCharArray();
3      for (int i = 0; i < board.length; i++) {
4          for (int j = 0; j < board[i].length; j++) {
5              if (dfs(board, wordArray, i, j, 0)) return true;
6          }
7      }
8      return false;
9  }
10 boolean dfs(char[][] board, char[] word, int i, int j, int k) {
11     if (i < 0 || i >= board.length || j < 0 || j >=
board[i].length || board[i][j] != word[k]) return false;
12     if (k == word.length - 1) return true;
13     board[i][j] = '\0';
14     boolean result = dfs(board, word, i, j - 1, k + 1) ||
dfs(board, word, i + 1, j, k + 1) ||
15         dfs(board, word, i, j + 1, k + 1) || dfs(board, word,
i - 1, j, k + 1);
16     board[i][j] = word[k];
17     return result;
18 }

```

Q13. 机器人的运动范围：DFS（数位和增量公式、可达性分析）、BFS

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1]。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37]，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

```

1  示例 1:
2
3  输入: m = 2, n = 3, k = 1
4  输出: 3
5
6  示例 2:
7
8  输入: m = 3, n = 1, k = 0
9  输出: 1

```

解：

1. 深度优先遍历DFS $O(MN)$ $O(MN)$
 - 数位和增量公式：不需要每次都整除和求余去计算位数和

由于机器人每次只能移动一格（即只能从 x 运动至 $x \pm 1$ ），因此每次只需计算 x 到 $x \pm 1$ 的数位和增量。本题说明 $1 \leq n, m \leq 100$ ，以下公式仅在此范围适用。

数位和增量公式： 设 x 的数位和为 s_x ， $x+1$ 的数位和为 s_{x+1} ；

1. 当 $(x+1) \odot 10 = 0$ 时： $s_{x+1} = s_x - 8$ ，例如 19, 20 的数位和分别为 10, 2；
2. 当 $(x+1) \odot 10 \neq 0$ 时： $s_{x+1} = s_x + 1$ ，例如 1, 2 的数位和分别为 1, 2。

- 可达解分析： 仅需向右、向下移动
- 与Q12类似，采用递归求解即可，代码十分类似

```
1  /**
2   * 递归求解，与Q12类似，仅需向右、向下移动
3   * @author PAN
4   * @param m 行数
5   * @param n 列数
6   * @param k 位数和上限
7   * @return 可达数
8   * @time O(MN)
9   * @space O(MN)
10  */
11 int movingCount(int m, int n, int k) {
12     DFS(m, n, 0, 0, k);
13     return xy.size();
14 }
15
16 /**
17  * DFS: 递归函数
18  * @author PAN
19  * @param m 行数
20  * @param n 列数
21  * @param i 初始位置
22  * @param j 初始位置
23  * @param k 位数和上限
24  */
25 void DFS(int m, int n, int i, int j, int k) {
26     int sum = i / 100 + i / 10 + i % 10 + j / 100 + j / 10 + j % 10;
27     boolean flag = sum > k ? false : true;
28     String tmp = i + "," + j;
29     if (i < 0 || i >= m || j < 0 || j >= n || !flag || xy.contains(tmp)) return;
30     xy.add(tmp);
31     DFS(m, n, i, j + 1, k);
32     DFS(m, n, i + 1, j, k);
33
34     // 优化，不必向上、向左，也意味不需要判断 i < 0 || j < 0
35     // DFS(m, n, i, j - 1, k);
36     // DFS(m, n, i - 1, j, k);
37 }
```

```

38
39 /**
40  * 递归求解优化：用 visited 数组记录是否访问，并精简代码
41  * @author PAN
42  * @param m 行数
43  * @param n 列数
44  * @param k 位数和上限
45  * @return 可达数
46  * @time O(MN)
47  * @space O(MN)
48  */
49 int movingCount2(int m, int n, int k) {
50     boolean[][] visited = new boolean[m][n];
51     return DFS2(m, n, 0, 0, k, visited);
52 }
53 int DFS2(int m, int n, int i, int j, int k, boolean[][] visited){
54     int sum = i / 100 + i / 10 + i % 10 + j / 100 + j / 10 + j %
55     10;
56     if (i >= m || j >= n || sum > k || visited[i][j]) return 0;
57     visited[i][j] = true;
58     return DFS2(m, n, i, j + 1, k, visited) + DFS2(m, n, i + 1, j,
k, visited) + 1;
59 }

```

2. 广度优先遍历BFS O(MN) O(MN)

◦ 用队列实现：

- 1) 将机器人初始点加入队列；
- 2) 将队首单元格的索引、数位弹出；
- 3) 判断是否越界或超出k或已访问；
- 4) 对未访问的单元格进行标记，(i,j)存入visited中；
- 5) 将当前元素的下方、右方单元格数位入队；
- 6) 队列为空时，停止迭代

```

1 /**
2  * BFS：用队列实现，重点在于数位和增量公式(不需要每次都整除和求余去计算位数和)
3  * @author 网友
4  * @param m 行数
5  * @param n 列数
6  * @param k 位数和上限
7  * @return 可达数
8  * @time O(MN)
9  * @space O(MN)
10 */
11 public int movingCount3(int m, int n, int k) {
12     boolean[][] visited = new boolean[m][n];

```

```

13     int res = 0;
14     Queue<int[]> queue= new LinkedList<int[]>();
15     queue.add(new int[] { 0, 0, 0, 0 });
16     while(queue.size() > 0) {
17         int[] x = queue.poll();
18         int i = x[0], j = x[1], si = x[2], sj = x[3];
19         if(i >= m || j >= n || k < si + sj || visited[i][j])
20             continue;
21         visited[i][j] = true;
22         res ++;
23         queue.add(new int[] { i + 1, j, (i + 1) % 10 != 0 ? si + 1
24 : si - 8, sj }); // 数位和增量公式
25         queue.add(new int[] { i, j + 1, si, (j + 1) % 10 != 0 ? sj
26 + 1 : sj - 8 });
27     }
28     return res;
29 }

```

Q14 - I. 剪绳子：数学推导、贪心、动态规划

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段 (m, n 都是整数, $n > 1$ 并且 $m > 1$)，每段绳子的长度记为 $k[0], k[1] \dots k[m-1]$ 。请问 $k[0]k[1] \dots k[m-1]$ 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

```

1  示例 1:
2
3  输入: 2
4  输出: 1
5  解释: 2 = 1 + 1, 1 × 1 = 1
6
7  示例 2:
8
9  输入: 10
10 输出: 36
11 解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36

```

解：

1. 数学推导 $O(1)$ $O(1)$

- 由重要不等式的推论可以证明，绳子越等分乘积越大；
- 求导也可以证明，同时可以求得驻点为 e ，即2.7左右，通过带入2、3可以得到绳子越多切成长度3，乘积越大；

切分规则：

- 最优：** 3。把绳子尽可能切为多个长度为3的片段，留下的最后一段绳子的长度可能为0, 1, 2三种情况。
- 次优：** 2。若最后一段绳子长度为2；则保留，不再拆为 $1+1$ 。
- 最差：** 1。若最后一段绳子长度为1；则应把一份 $3+1$ 替换为 $2+2$ ，因为 $2 \times 2 > 3 \times 1$ 。

o 那么转换成以下算法：

- 1) $3a + b = n$ ，当 $n \leq 3$ 时，由于 $m > 1$ ，那么必须有一段绳子长度为1，乘积即为 $n - 1$ ；
- 2) 当 $n > 3$ 时，对 b 讨论，即 $n \% 3$ 讨论： $b = 0$ 时，乘积为 3^a ； $b = 1$ 时，乘积为 $3^{a-1} * 4$ ，即有一段长度为3的绳子要拿出来和长度为1的绳子形成 $2 + 2$ ； $b = 2$ 时，乘积为 $3^a * 2$ ；

```
1  /**
2   * 数学推导/贪心：
3   *   1. 绳子越等分乘积越大；
4   *   2. 越多切分成长度 3，乘积越大；
5   *   3. 以长度 3 为基础去切分，对最后一段长度（可能：0， 1， 2）进行讨论计
      算；
6   * @author PAN & 网友：自己推断了1，但2不太确定，可以用求导得驻点是e（2.7）
7   * @param n 绳子长度
8   * @return 最大乘积
9   * @time O(1)
10  * @space O(1)
11  */
12 public static int cuttingRope(int n) {
13     if (n <= 3) return n - 1;
14     int a = n / 3, b = n % 3;
15     if(b == 0) return (int) Math.pow(3, a);
16     else if(b == 1) return 4 * (int) Math.pow(3, a - 1);
17     else return 2 * (int) Math.pow(3, a);
18 }
```

2. 贪心

贪心思路：

设一绳子长度为 n ($n > 1$)，则其必可被切分为两段 $n = n_1 + n_2$ 。

根据经验推测，切分的两数字乘积往往原数字更大，即往往有 $n_1 \times n_2 > n_1 + n_2 = n$ 。

- 例如绳子长度为 6： $6 = 3 + 3 < 3 \times 3 = 9$ ；
- 也有少数反例，例如 2： $2 = 1 + 1 > 1 \times 1 = 1$ 。

- 推论一： 合理的切分方案可以带来更大的乘积。

设一绳子长度为 n ($n > 1$)，切分为两段 $n = n_1 + n_2$ ，切分为三段 $n = n_1 + n_2 + n_3$ 。

根据经验推测，三段的乘积往往更大，即往往有 $n_1 n_2 n_3 > n_1 n_2$ 。

- 例如绳子长度为 9： 两段 $9 = 4 + 5$ 和 三段 $9 = 3 + 3 + 3$ ，则有 $4 \times 5 < 3 \times 3 \times 3$ 。
- 也有少数反例，例如 6： 两段 $6 = 3 + 3$ 和 三段 $6 = 2 + 2 + 2$ ，则有 $3 \times 3 > 2 \times 2 \times 2$ 。

- 推论二： 若切分方案合理，绳子段切分的越多，乘积越大。

总体上看，貌似长绳子切分为越多段乘积越大，但其实到某个长度分界点后，乘积到达最大值，就不应再切分了。

问题转化： 是否有优先级最高的长度 x 存在？若有，则应该尽可能把绳子以 x 长度切为多段，以获取最大乘积。

- 推论三： 为使乘积最大，只有长度为 2 和 3 的绳子不应再切分，且 3 比 2 更优（详情见下表）。

绳子切分方案	乘积	结论
$2 = 1 + 1$	$1 \times 1 = 1$	2 不应切分
$3 = 1 + 2$	$1 \times 2 = 2$	3 不应切分

3. 动态规划 $O(N^2)$ $O(N)$

思路一：动态规划

这题用动态规划是比较好理解的

1. 我们要求长度为 n 的绳子剪掉后的最大乘积，可以从前面比 n 小的绳子转移而来
2. 用一个 dp 数组 记录 从 0 到 n 长度的绳子剪掉后的最大乘积，也就是 $dp[i]$ 表示长度为 i 的绳子剪成 m 段后的最大乘积，初始化 $dp[2] = 1$
3. 我们先把绳子剪掉第一段（长度为 j ），如果只剪掉长度为 1，对最后的乘积无任何增益，所以从长度为 2 开始剪
4. 剪了第一段后，剩下 $(i - j)$ 长度可以剪也可以不剪。如果不剪的话长度乘积即为 $j * (i - j)$ ；如果剪的话长度乘积即为 $j * dp[i - j]$ 。取两者最大值 $\max(j * (i - j), j * dp[i - j])$
5. 第一段长度 j 可以取的区间为 $[2, i)$ ，对所有 j 不同的情况取最大值，因此最终 $dp[i]$ 的转移方程为
$$dp[i] = \max(dp[i], \max(j * (i - j), j * dp[i - j]))$$
6. 最后返回 $dp[n]$ 即可

```

1  class Solution {
2      public int cuttingRope(int n) {
3          int[] dp = new int[n + 1];
4          dp[2] = 1;
5          for(int i = 3; i < n + 1; i++){
6              for(int j = 2; j < i; j++){
7                  dp[i] = Math.max(dp[i], Math.max(j * (i - j), j *
dp[i - j]));
8              }
9          }
10         return dp[n];
11     }
12 }

```

Q14 - II. 剪绳子 II：循环求余、快速求余

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（ m 、 n 都是整数， $n > 1$ 并且 $m > 1$ ），每段绳子的长度记为 $k[0], k[1] \dots k[m - 1]$ 。请问 $k[0]k[1] \dots k[m - 1]$ 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18。

答案需要取模 $1e9+7$ (1000000007)，如计算初始结果为：1000000008，请返回 1。

```

1  示例 1：
2
3  输入：2
4  输出：1
5  解释：2 = 1 + 1, 1 × 1 = 1
6
7  示例 2：
8
9  输入：10
10 输出：36
11 解释：10 = 3 + 3 + 4, 3 × 3 × 4 = 36

```

解（此题跟上一题差别在于 n 的范围）：

1. 循环求余 $O(N)$ $O(1)$ - 每一次幂运算都求一次余数；

```

1  /**
2   * 循环求余
3   * @author PAN
4   * @param n 绳子长度
5   * @return 最大乘积
6   * @time O(N)
7   * @space O(1)
8   */
9  public static int cuttingRope(int n) {
10     if (n <= 3) return n - 1;

```

```

11  int a = n / 3, b = n % 3;
12  long mul = 1;
13  int r = 1000000007;
14  for (int i = 0; i < a - 1; i++) {
15      mul = (mul * 3) % r;
16  }
17  if(b == 0) return (int)(mul * 3 % r);
18  else if(b == 1) return (int)((4 * mul) % r);
19  else return (int)(mul * 6 % r);
20  }

```

2. 快速求余 $O(\log_2 N) O(1)$

○

- 根据求余运算性质可推出:

$$x^a \odot p = (x^2)^{a/2} \odot p = (x^2 \odot p)^{a/2} \odot p$$

- 当 a 为奇数时 $a/2$ 不是整数，因此分为以下两种情况（“//”代表向下取整的除法）：

$$x^a \odot p = \begin{cases} (x^2 \odot p)^{a/2} \odot p & , a \text{ 为偶数} \\ [(x \odot p)(x^{a-1} \odot p)] \odot p = [x(x^2 \odot p)^{a/2}] \odot p & , a \text{ 为奇数} \end{cases}$$

- **解析：**利用以上公式，可通过循环操作每次把指数 a 问题降低至指数 $a/2$ 问题，只需循环 $\log_2(N)$ 次，因此可将复杂度降低至对数级别。封装方法代码如下所示。

```

1  class Solution {
2      public int cuttingRope(int n) {
3          if(n <= 3) return n - 1;
4          int b = n % 3, p = 1000000007;
5          long rem = 1, x = 3;
6          for(int a = n / 3 - 1; a > 0; a /= 2) {
7              if(a % 2 == 1) rem = (rem * x) % p;
8              x = (x * x) % p;
9          }
10         if(b == 0) return (int)(rem * 3 % p);
11         if(b == 1) return (int)(rem * 4 % p);
12         return (int)(rem * 6 % p);
13     }
14 }

```

Q15. 二进制中1的个数：巧用 $n \& (n - 1)$ 、逐位判断

请实现一个函数，输入一个整数（以二进制串形式），输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

```
1 示例 1:  
2  
3 输入: 000000000000000000000000000000001011  
4 输出: 3
```



```
5   解释：输入的二进制串 00000000000000000000000001011 中，共有三位为 '1'。
6
7   示例 2：
8
9   输入：000000000000000000000000010000000
10  输出：1
11  解释：输入的二进制串 000000000000000000000000010000000 中，共有一位为 '1'。
12
13 示例 3：
14
15 输入：1111111111111111111111111111101
16 输出：31
17 解释：输入的二进制串 1111111111111111111111111111101 中，共有 31 位为 '1'。
```

解：

1. 巧用 $n \& (n - 1)$ $O(M)$ $O(1)$, M 为 1 的个数。

- (n-1) 解析：二进制数字 n 最右边的 1 变成 0，此 1 右边的 0 都变成 1。
- n & (n - 1) 解析：二进制数字 n 最右边的 1 变成 0，其余不变。

```

1  /**
2   * 巧用  $n \& (n - 1)$ 
3   *  $(n-1)$ : 二进制数字  $n$  最右边的 1 变成 0 , 此 1 右边的 0 都变成 1 。
4   *  $n \& (n - 1)$ : 二进制数字  $n$  最右边的 1 变成 0 , 其余不变。
5   * @author 网友 & PAN, 借鉴  $n \& (n - 1)$  的思想
6   * @param n 无符号二进制数
7   * @return 1 的个数
8   * @time  $O(M)$   $M$  为 1 的个数
9   * @space  $O(1)$ 
10  */
11 public static int hammingWeight(int n) {
12     int count = 0;
13     while (n != 0) {
14         n = n & (n - 1);
15         count++;
16     }
17     return count;
18 }

```

2. 逐位判断 $O(\log_2 N)$ $O(1)$ - 将 n 和 1 做与操作, 结果为 1 时计数, 再将 n 向右移位重复操作。

```
1  /**
2   * 逐位判断：将 n 和 1 做与操作，结果为 1 时计数，再将 n 向右移位重复操作。
3   * @author 网友
4   * @param n 无符号二进制数
5   * @return 1 的个数
6   * @time O(log2 N) M 为 1 的个数
7   * @space O(1)
```

```

8      */
9      public static int hammingWeight2(int n) {
10         int count = 0;
11         while (n != 0) {
12             count += n & 1;
13             n >>= 1; // 这里需要写成>>=, Java中的无符号右移
14             // n = n >> 1; // 自己的写法
15         }
16         return count;
17     }

```

Q16. 数值的整数次方：循环求幂、二进制快速幂、二分法快速幂

实现 $\text{pow}(x, n)$ ，即计算 x 的 n 次幂函数（即， x^n ）。不得使用库函数，同时不需要考虑大数问题。

```

1  示例 1:
2
3  输入: x = 2.00000, n = 10
4  输出: 1024.00000
5  示例 2:
6
7  输入: x = 2.10000, n = 3
8  输出: 9.26100
9  示例 3:
10
11 输入: x = 2.00000, n = -2
12 输出: 0.25000
13 解释:  $2^{-2} = 1/2^2 = 1/4 = 0.25$ 

```

解：

1. 循环求解 $O(N)$ $O(1)$ - 用循环一次次累乘，以求幂运算；

```

1  /**
2   * 循环求解：用循环一次次累乘，以求幂运算。
3   * 该法有几个坑：
4   * 1. 需要对  $x = -1, 0, 1$  的特殊情况进行判断；
5   * 2. 测试用例中有一项  $n$  为 2147483648, 超过 int 范围 (2147483647),
6   *    需要用 long 或者判断 double 精度不够直接置 0；
7   * @author PAN
8   * @param x 底数
9   * @param n 幂次
10  * @return  $x^n$ 
11  * @time  $O(N)$ 
12  * @space  $O(1)$ 
13  */
14  public static double myPow(double x, int n) {

```

```

15     if (x == 0 || x == 1) return x;
16     double pow = 1;
17     if (n < 0) {
18         n = n * (-1);
19         x = 1.0 / x;
20     }
21     if (x == -1 && n % 2 == 0) return -x;
22     else if (x == -1 && n % 2 != 0) return x;
23     if (n < 0) return 0;
24     for (int i = 0; i < n; i++) {
25         pow *= x;
26         if (pow == 0) break;
27     }
28     return pow;
29 }

```

2. 二进制快速幂 $O(\log_2 N)$ $O(1)$ - 利用十进制数字 n 的二进制表示，可对快速幂进行数学化解释。

- 对于任何十进制正整数 n ，设其二进制为 " $b_m \dots b_3 b_2 b_1$ " (b_i 为二进制某位值, $i \in [1, m]$)，则有：
 - 二进制转十进制： $n = 1b_1 + 2b_2 + 4b_3 + \dots + 2^{m-1}b_m$ (即二进制转十进制公式)；
 - 幂的二进制展开： $x^n = x^{1b_1 + 2b_2 + 4b_3 + \dots + 2^{m-1}b_m} = x^{1b_1} x^{2b_2} x^{4b_3} \dots x^{2^{m-1}b_m}$ ；
- 根据以上推导，可把计算 x^n 转化为解决以下两个问题：
 - 计算 $x^1, x^2, x^4, \dots, x^{2^{m-1}}$ 的值：循环赋值操作 $x = x^2$ 即可；
 - 获取二进制各位 $b_1, b_2, b_3, \dots, b_m$ 的值：循环执行以下操作即可。
 - $n \& 1$ (与操作)：判断 n 二进制最右一位是否为 1；
 - $n >> 1$ (移位操作)： n 右移一位 (可理解为删除最后一位)。

```

1  /**
2   * 二进制快速幂：利用十进制数字 n 的二进制表示，可对快速幂进行数学化解释
3   * 或二分法快速幂：对 n 不断除以 2 来快速计算，只需要判断 n 的奇偶
4   * @author 网友
5   * @param x 底数
6   * @param n 幂次
7   * @return x ^ n
8   * @time O(log2 N)
9   * @space O(1)
10  */
11 public static double myPow2(double x, int n) {
12     if (x == 0) return x;
13     double pow = 1.0;
14     long newN = n;
15     if (newN < 0) {
16         newN = -newN;
17         x = 1.0 / x;
18     }
19     while (newN != 0) {
20         if ((newN & 1) == 1) pow *= x;

```

```

21         x *= x;
22         newN >>= 1;
23     }
24     return pow;
25 }

```

3. 二分法快速幂 $O(\log_2 N)$ $O(1)$ - 与上个方法类似

- 二分推导: $x^n = x^{n/2} \times x^{n/2} = (x^2)^{n/2}$, 令 $n/2$ 为整数, 则需要分为奇偶两种情况 (设向下取整除法符号为 $"/"$) :
 - 当 n 为偶数: $x^n = (x^2)^{n/2}$;
 - 当 n 为奇数: $x^n = x(x^2)^{n/2}$, 即会多出一项 x ;

Q17. 打印从1到最大的n位数：循环打印、大数打印（递归）

输入数字 n , 按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3, 则打印出 1、2、3 一直到最大的 3 位数 999。

```

1  示例 1:
2
3  输入: n = 1
4  输出: [1,2,3,4,5,6,7,8,9]

```

解:

1. 循环打印 $O(10^N)$ $O(1)$ (列表作为返回结果, 不计入额外空间) - 直接构建数组循环输出。

```

1  /**
2   * 循环打印: 直接构建数组循环输出
3   * @author PAN
4   * @param n 最大位数
5   * @return 从 1 到最大的 n 位十进制数
6   * @time  $O(10^N)$ 
7   * @space  $O(1)$ 
8   */
9  public static int[] printNumbers(int n) {
10     int max = (int) Math.pow(10, n) - 1;
11     int[] printArray = new int[max];
12     for (int i = 0; i < max; i++) {
13         printArray[i] = i + 1;
14     }
15     return printArray;
16 }

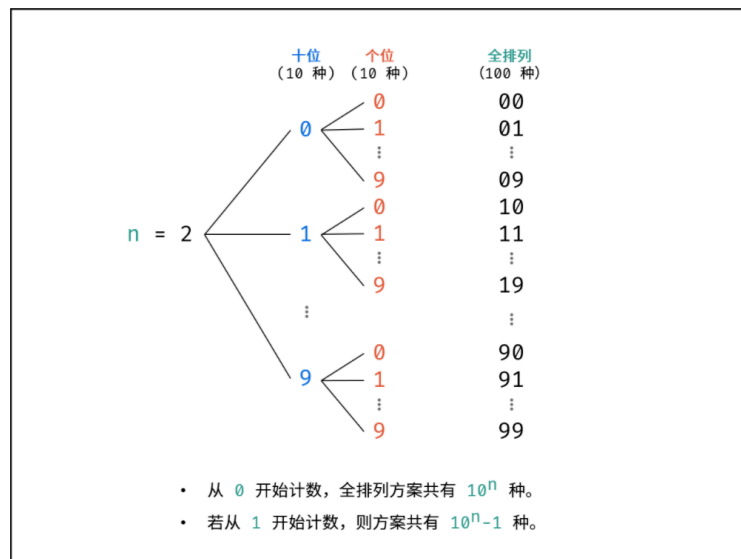
```

2. 大数打印 $O(10^N)$ $O(10^N)$ - 题目给定了 `int` 范围, 但实际情况可能会考大数, 这时候用 `int` 无法解, 需要用 `String`。

○

递归生成全排列：

- 基于分治算法的思想，先固定高位，向低位递归，当个位已被固定时，添加数字的字符串。例如当 $n = 2$ 时（数字范围 $1 - 99$ ），固定十位为 $0 - 9$ ，按顺序依次开启递归，固定个位 $0 - 9$ ，终止递归并添加数字字符串。



○ 主要处理两个问题：

1. 删除高位多余的 0：

1. 删除高位多余的 0：

- 字符串左边界定义：声明变量 `start` 规定字符串的左边界，以保证添加的数字字符串 `num[start:]` 中无高位多余的 0。例如当 $n = 2$ 时， $1 - 9$ 时 `start = 1`， $10 - 99$ 时 `start = 0`。
- 左边界 `start` 变化规律：观察可知，当输出数字的所有位都是 9 时，则下个数字需要向更高位进 1，此时左边界 `start` 需要减 1（即高位多余的 0 减少一个）。例如当 $n = 3$ （数字范围 $1 - 999$ ）时，左边界 `start` 需要减 1 的情况有：“009”进位至“010”，“099”进位至“100”。设数字各位中 9 的数量为 `nine`，所有位都为 9 的判断条件可用以下公式表示：

$$n - start = nine$$

- 统计 `nine` 的方法：固定第 x 位时，当 $i = 9$ 则执行 `nine = nine + 1`，并在回溯前恢复 `nine = nine - 1`。

2. 列表从 1 开始：添加字符串前判断是否为“0”，是则跳过。

```
1  /**
2   * 大数打印：题目给定了 int 范围，但实际情况可能会考大数，这时候用 int 无法
   * 解，需要用 String。
3   * @author 网友 & PAN，根据思路改了一些写法，
4   * 但还有问题：这样默认还是 int 范围，应该将 printNumber 返回结果也改为
   String
5   * @param n 最大位数
6   * @return 从 1 到最大的 n 位十进制数
7   * @time O(10^N)
8   * @space O(10^N)
9   */
10 public int[] printNumbers2(int n) {
11     this.n = n;
```

```

12     this.num = new char[n];
13     this.printArray = new int[(int) Math.pow(10, n) - 1];
14     dfs(0);
15     return this.printArray;
16 }
17
18 /**
19  * 递归主体：先固定高位，向低位递归，当个位已被固定时，添加数字的字符串
20  * @param x 递归位数 (0 - n)
21  */
22 public void dfs(int x) {
23     if (x == this.n) {
24         if (Integer.parseInt(String.valueOf(num)) != 0) {
25             this.printArray[count] =
Integer.parseInt(String.valueOf(num));
26             count++;
27         }
28         return;
29     }
30     for (char c : this.loop) {
31         this.num[x] = c;
32         dfs(x + 1);
33     }
34 }

```

Q18. 删除链表的节点：遍历对比删除

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。返回删除后的链表的头节点。

```

1  示例 1：
2
3  输入：head = [4,5,1,9], val = 5
4  输出：[4,1,9]
5  解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -
    > 9。
6
7  示例 2：
8
9  输入：head = [4,5,1,9], val = 1
10 输出：[4,5,9]
11 解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -
    > 9。

```

解：

1. 遍历对比删除 $O(N)$ $O(1)$ - 从头节点开始依次比较，找到要删除的节点进行删除，即将上一个节点指向下一个节点。

```

1  /**
2   * 遍历对比删除：从头节点开始依次比较，找到要删除的节点进行删除，即将上一个节点
   指向下一个节点。
3   * @author PAN
4   * @param head 头节点
5   * @param val 待删除节点的值
6   * @return 头节点
7   * @time O(N)
8   * @space O(1)
9   */
10 public ListNode deleteNode(ListNode head, int val) {
11     if (head.val == val) {
12         head = head.next;
13         return head;
14     }
15     ListNode point = head;
16     while (point.next != null) {
17         if (point.next.val == val) point.next = point.next.next;
18         else point = point.next;
19     }
20     return head;
21 }

```

Q19. 正则表达式匹配：动态规划

请实现一个函数用来匹配包含 `.` 和 `*` 的正则表达式。模式中的字符 `.` 表示任意一个字符，而 `*` 表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串 `"aaa"` 与模式 `"a.a"` 和 `"ab * ac * a"` 匹配，但与 `"aa.a"` 和 `"ab*a"` 均不匹配。

```

1  示例 1：
2
3  输入：
4  s = "aa"
5  p = "a"
6  输出：false
7  解释："a" 无法匹配 "aa" 整个字符串。
8
9  示例 2：
10
11 输入：
12 s = "aa"
13 p = "a*"
14 输出：true
15 解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
16
17 示例 3：
18

```

```

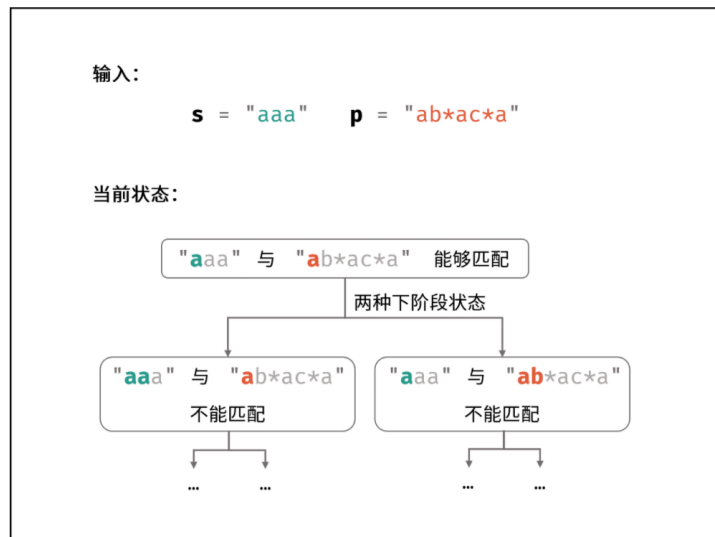
19 输入：
20 s = "ab"
21 p = ".*"
22 输出：true
23 解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
24
25 示例 4：
26
27 输入：
28 s = "aab"
29 p = "c*a*b"
30 输出：true
31 解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因此可以匹配字符串
    "aab"。
32
33 示例 5：
34
35 输入：
36 s = "mississippi"
37 p = "mis*is*p*."
38 输出：false

```

解：

1. 动态规划 $O(NM)$ $O(NM)$

- 总体思路：每轮添加一个字符并判断是否能匹配，直至添加完整整个字符串 s 和 p 。
-



因此，本题的状态共有 $m \times n$ 种，应定义状态矩阵 dp ， $dp[i][j]$ 代表 $s[:i]$ 与 $p[:j]$ 是否可以匹配。

做好状态定义，接下来就是根据「普通字符」，「`.`」，「`*`」三种字符的功能定义，分析出动态规划的转移方程。

动态规划解析：

- **状态定义：** 设动态规划矩阵 `dp` , `dp[i][j]` 代表字符串 `s` 的前 `i` 个字符和 `p` 的前 `j` 个字符能否匹配。
- **转移方程：** 需要注意, 由于 `dp[0][0]` 代表的是空字符的状态, 因此 `dp[i][j]` 对应的添加字符是 `s[i - 1]` 和 `p[j - 1]` 。
 - 当 `p[j - 1] == '*'` 时, `dp[i][j]` 在当以下任一情况为 `true` 时等于 `true` :
 1. `dp[i][j - 2]` : 即将字符组合 `p[j - 2] *` 看作出现 0 次时, 能否匹配;
 2. `dp[i - 1][j]` 且 `s[i - 1] == p[j - 2]` : 即让字符 `p[j - 2]` 多出现 1 次时, 能否匹配;
 3. `dp[i - 1][j]` 且 `p[j - 2] == '.'` : 即让字符 `'.'` 多出现 1 次时, 能否匹配;
 - 当 `p[j - 1] != '*'` 时, `dp[i][j]` 在当以下任一情况为 `true` 时等于 `true` :
 1. `dp[i - 1][j - 1]` 且 `s[i - 1] == p[j - 1]` : 即让字符 `p[j - 1]` 多出现一次时, 能否匹配;
 2. `dp[i - 1][j - 1]` 且 `p[j - 1] == '.'` : 即将字符 `.` 看作字符 `s[i - 1]` 时, 能否匹配;
- **初始化：** 需要先初始化 `dp` 矩阵首行, 以避免状态转移时索引越界。
 - `dp[0][0] = true` : 代表两个空字符串能够匹配。
 - `dp[0][j] = dp[0][j - 2]` 且 `p[j - 1] == '*'` : 首行 `s` 为空字符串, 因此当 `p` 的偶数位为 `*` 时才能够匹配 (即让 `p` 的奇数位出现 0 次, 保持 `p` 是空字符串)。因此, 循环遍历字符串 `p` , 步长为 2 (即只看偶数位) 。
- **返回值：** `dp` 矩阵右下角字符, 代表字符串 `s` 和 `p` 能否匹配。

```
1  /**
2   * 动态规划：每轮添加一个字符并判断是否能匹配，直至添加完整整个字符串 s 和 p
3   * @author 网友
4   * @param s 字符串
5   * @param p 正则表达式
6   * @return 匹配成功与否
7   * @time O(NM)
8   * @space O(NM)
9   */
10 public static boolean isMatch2(String s, String p) {
11     boolean[][] dp = new boolean[s.length() + 1][p.length() + 1];
12     dp[0][0] = true;
13     for (int j = 2; j < p.length() + 1; j += 2) {
14         dp[0][j] = dp[0][j - 2] && p.charAt(j - 1) == '*';
15     }
16     for (int i = 1; i < s.length() + 1; i++) {
17         for (int j = 1; j < p.length() + 1; j++) {
18             if (p.charAt(j - 1) == '*') {
19                 if (dp[i][j - 2]) dp[i][j] = true;
20                 else if (dp[i - 1][j] && s.charAt(i - 1) ==
21 p.charAt(j - 2)) dp[i][j] = true;
22                 else if (dp[i - 1][j] && p.charAt(j - 2) == '.')
23 dp[i][j] = true;
24             } else {
25                 if (dp[i - 1][j - 1] && s.charAt(i - 1) ==
26 p.charAt(j - 1)) dp[i][j] = true;
```

```

24         else if (dp[i - 1][j - 1] && p.charAt(j - 1) ==
    '.'') dp[i][j] = true;
25     }
26 }
27 }
28 return dp[s.length()][p.length()];
29 }

```

Q20. 表示数值的字符串：逐位判断、有限状态自动机

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100"、"5e2"、"-123"、"3.1416"、"-1E-16"、"0123"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"及"12e+5.4"都不是。

解：

1. 遍历判断 $O(N)$ $O(1)$ - 逐位遍历判断是否符合数字要求

- '.'出现正确的情况：只出现一次，且在e/E前；
- 'e/E'出现正确的情况：只出现一次，且出现前有数字；
- '+/-'出现正确的情况：只能在开头或者e/E后一位；

```

1 public static boolean isNumber(String s) {
2     boolean hasNum = false, hasDecimal = false, hasE = false; //
    是否有数字、小数、e/E
3     s = s.trim(); // 删除前后多余空格
4     for (int i = 0; i < s.length(); i++) {
5         if (s.charAt(i) >= '0' && s.charAt(i) <= '9') hasNum =
        true;
6         else if (s.charAt(i) == '.' && !hasDecimal && !hasE) {
7             hasDecimal = true;
8         } else if ((s.charAt(i) == 'e' || s.charAt(i) == 'E') &&
        !hasE && hasNum){
9             hasE = true;
10            hasNum = false;
11        } else if ((s.charAt(i) == '+' || s.charAt(i) == '-') &&
        (i == 0 || s.charAt(i - 1) == 'e' || s.charAt(i - 1) == 'E')) {
12
13        } else {
14            return false;
15        }
16    }
17    return hasNum;
18 }

```

2. 有限状态自动机 $O(N)$ $O(1)$

◦

解题思路：

本题使用有限状态自动机。根据字符类型和合法数值的特点，先定义状态，再画出状态转移图，最后编写代

码即可。

字符类型：

空格「」、数字「`0—9`」、正负号「`+-`」、小数点「`.`」、幂符号「`eE`」。

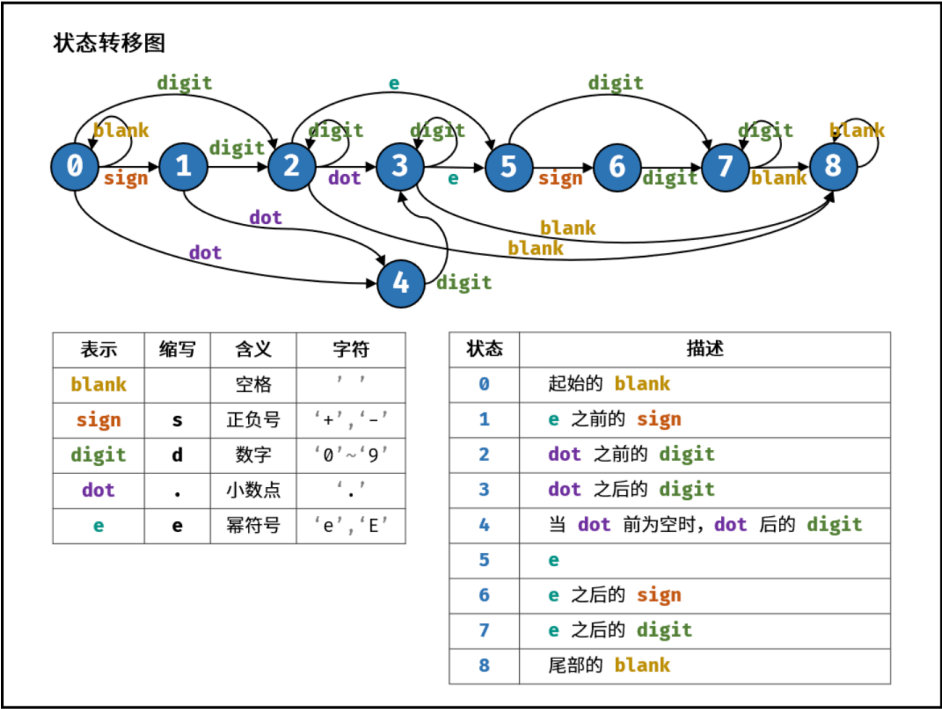
状态定义：

按照字符串从左到右的顺序，定义以下 9 种状态。

- 0. 开始的空格
- 1. 幂符号前的正负号
- 2. 小数点前的数字
- 3. 小数点、小数点后的数字
- 4. 当小数点前为空格时，小数点、小数点后的数字
- 5. 幂符号
- 6. 幂符号后的正负号
- 7. 幂符号后的数字
- 8. 结尾的空格

结束状态：

合法的结束状态有 2, 3, 7, 8。



算法流程：

1. 初始化：

- 1. 状态转移表 `states`：设 `states[i]`，其中 `i` 为所处状态，`states[i]` 使用哈希表存储可转移至的状态。键值对 (`key, value`) 含义：若输入 `key`，则可从状态 `i` 转移至状态 `value`。
- 2. 当前状态 `p`：起始状态初始化为 `p = 0`。

2. 状态转移循环：遍历字符串 `s` 的每个字符 `c`。

1. 记录字符类型 `t`：分为四种情况。

- 当 `c` 为正负号时，执行 `t = 's'`；
- 当 `c` 为数字时，执行 `t = 'd'`；
- 当 `c` 为 `e`，`E` 时，执行 `t = 'e'`；
- 当 `c` 为 `.`，空格 时，执行 `t = c`（即用字符本身表示字符类型）；
- 否则，执行 `t = '?'`，代表为不属于判断范围的非法字符，后续直接返回 `false`。

2. 终止条件：若字符类型 `t` 不在哈希表 `states[p]` 中，说明无法转移至下一状态，因此直接返回

False。

3. 状态转移：状态 p 转移至 $states[p][t]$ 。

3. 返回值：跳出循环后，若状态 $p \in 2, 3, 7, 8$ ，说明结尾合法，返回 *True*，否则返回 *False*。

复杂度分析：

- 时间复杂度 $O(N)$ ：其中 N 为字符串 s 的长度，判断需遍历字符串，每轮状态转移的使用 $O(1)$ 时间。
- 空间复杂度 $O(1)$ ： $states$ 和 p 使用常数大小的额外空间。

```
1 class Solution {
2     public boolean isNumber(String s) {
3         Map[] states = {
4             new HashMap<>() {{ put(' ', 0); put('s', 1); put('d',
5 2); put('.', 4); }}, // 0.
6             new HashMap<>() {{ put('d', 2); put('.', 4); }},
7                 // 1.
8             new HashMap<>() {{ put('d', 2); put('.', 3); put('e',
9 5); put(' ', 8); }}, // 2.
10            new HashMap<>() {{ put('d', 3); put('e', 5); put(' ',
11 8); }}, // 3.
12            new HashMap<>() {{ put('d', 3); }},
13                // 4.
14            new HashMap<>() {{ put('s', 6); put('d', 7); }},
15                // 5.
16            new HashMap<>() {{ put('d', 7); }},
17                // 6.
18            new HashMap<>() {{ put('d', 7); put(' ', 8); }},
19                // 7.
20            new HashMap<>() {{ put(' ', 8); }}
21                // 8.
22        };
23        int p = 0;
24        char t;
25        for(char c : s.toCharArray()) {
26            if(c >= '0' && c <= '9') t = 'd';
27            else if(c == '+' || c == '-') t = 's';
28            else if(c == 'e' || c == 'E') t = 'e';
29            else if(c == '.' || c == ' ') t = c;
30            else t = '?';
31            if(!states[p].containsKey(t)) return false;
32            p = (int)states[p].get(t);
33        }
34        return p == 2 || p == 3 || p == 7 || p == 8;
35    }
36 }
```

Q21. 调整数组顺序使奇数位于偶数前面：遍历找奇偶并用数组存储、双指针交换

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

```
1  示例：
2
3  输入：nums = [1,2,3,4]
4  输出：[1,3,2,4]
5  注：[3,1,2,4] 也是正确的答案之一。
```

解：

1. 遍历找奇偶并用数组存储 $O(N)$ $O(N)$ - 遍历数组，用两个指针指向新数组头尾，遇到奇数往头部放，遇到偶数往尾部放。

```
1  /**
2   * 遍历找奇偶并用数组存储：遍历数组，用两个指针指向新数组头尾，遇到奇数往头部
   * 放，遇到偶数往尾部放。
3   * @author PAN
4   * @param nums 待调整的数组
5   * @return 按奇偶调整后的数组
6   * @time  $O(N)$ 
7   * @space  $O(N)$ 
8   */
9  public static int[] exchange(int[] nums) {
10     int i = 0, j = nums.length - 1;
11     int[] exchangeNums = new int[nums.length];
12     for (int k = 0; k < nums.length; k++) {
13         if (nums[k] % 2 == 0) {
14             exchangeNums[j] = nums[k];
15             j--;
16         }
17         else {
18             exchangeNums[i] = nums[k];
19             i++;
20         }
21     }
22     return exchangeNums;
23 }
```

2. 双指针交换 $O(N)$ $O(1)$ - 一头一尾指针指向原数组，头指针为奇数时右移，尾指针为偶数时左移，找到第一个不满足条件的两个值交换，然后继续循环。

```
1  /**
2   * 双指针交换：一头一尾指针指向原数组，头指针为奇数时右移，尾指针为偶数时左移，
   * 找到第一个不满足条件的两个值交换，然后继续循环。
```

```

3  * @author PAN
4  * @param nums 待调整的数组
5  * @return 按奇偶调整后的数组
6  * @time O(N)
7  * @space O(1)
8  */
9  public static int[] exchange2(int[] nums) {
10     int i = 0, j = nums.length - 1;
11     int tmp;
12     while (i < j) {
13         boolean flagI = (nums[i] % 2 == 0);
14         boolean flagJ = (nums[j] % 2 == 1);
15         if (flagI && flagJ) {
16             tmp = nums[i];
17             nums[i] = nums[j];
18             nums[j] = tmp;
19             i++;
20             j--;
21         } else {
22             if (!flagI) i++;
23             if (!flagJ) j--;
24         }
25     }
26     return nums;
27 }
28
29 /**
30  * 双指针交换2
31  * @author 网友
32  * @param nums 待调整的数组
33  * @return 按奇偶调整后的数组
34  * @time O(N)
35  * @space O(1)
36  */
37 public int[] exchange3(int[] nums) {
38     int left = 0;
39     int right = nums.length - 1;
40     while (left < right) {
41         while (left < right && nums[left] % 2 != 0) {
42             left++;
43         }
44         while (left < right && nums[right] % 2 == 0) {
45             right--;
46         }
47         if (left < right) {
48             int temp = nums[left];
49             nums[left] = nums[right];
50             nums[right] = temp;
51         }

```

```
52     }
53     return nums;
54 }
```

Q22. 链表中倒数第k个节点：先求长度再顺序找、双指针寻找

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

例如，一个链表有 6 个节点，从头节点开始，它们的值依次是 1、2、3、4、5、6。这个链表的倒数第 3 个节点是值为 4 的节点。

```
1  示例：
2
3  给定一个链表：1->2->3->4->5，和 k = 2。
4
5  返回链表 4->5。
```

1. 先求链表长度再找 $O(N) O(1)$ - 先遍历一边链表求出长度 len ，在利用 $len - k$ 去找该倒数节点；

```
1  /**
2   * 先求链表长度再找：先遍历一边链表求出长度 len，在利用 len - k 去找该倒数
   节点
3   * @author PAN
4   * @param head 链表头节点
5   * @param k 倒数第 k
6   * @return 链表倒数第 k 个节点
7   * @time  $O(N)$ 
8   * @space  $O(1)$ 
9   */
10 public ListNode getKthFromEnd(ListNode head, int k) {
11     int len = 0;
12     ListNode point = head;
13     while (point != null) {
14         len++;
15         point = point.next;
16     }
17     if (k > len) return null;
18     int i = 0;
19     point = head;
20     while (i != len - k) {
21         point = point.next;
22         i++;
23     }
24     return point;
25 }
```

2. 双指针 $O(N)$ $O(1)$ - 利用快慢指针，先使得 $\text{latter} = \text{former} + k$ ，再两个指针同步后移，当 latter 为空时 former 即为要找的节点；

```
1  /**
2   * 双指针：利用快慢指针，先使得 latter = former + k，再两个指针同步后移，当
   latter 为空时 former 即为要找的节点
3   * @author 网友 & PAN，参考思路自己实现
4   * @param head 链表头节点
5   * @param k 倒数第 k
6   * @return 链表倒数第 k 个节点
7   * @time  $O(N)$ 
8   * @space  $O(1)$ 
9   */
10 public ListNode getKthFromEnd2(ListNode head, int k) {
11     ListNode former = head, latter = head;
12     for (int i = 0; i < k; i++) {
13         if (latter == null) return null;
14         latter = latter.next;
15     }
16     while (latter != null) {
17         former = former.next;
18         latter = latter.next;
19     }
20     return former;
21 }
```

3. 栈 $O(N)$ $O(N)$ - 先遍历节点逐个压栈，再弹出第 k 个即为所需；

Q24. 反转链表：栈、数组、双指针、递归

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

```
1  示例：
2
3  输入：1->2->3->4->5->NULL
4  输出：5->4->3->2->1->NULL
```

1. 栈/数组顺序存储再反转 $O(N)$ $O(N)$ - 用栈或者数组先按照原先顺序存储，再按照倒序弹出栈或者数组反向操作；

```
1  /**
2   * 栈：用栈先按照原先顺序存储，再按照倒序弹出栈
3   * @author PAN
4   * @param head 链表头节点
5   * @return 反转后链表头
6   * @time  $O(N)$ 
7   * @space  $O(N)$ 
8   */
```



```

9 public ListNode reverseList(ListNode head) {
10     if (head == null) return null;
11     Stack s = new Stack();
12     while (head != null) {
13         s.push(head);
14         head = head.next;
15     }
16     ListNode newHead = (ListNode) s.pop();
17     ListNode point = newHead;
18     while (!s.isEmpty()) {
19         point.next = (ListNode) s.pop();
20         point = point.next;
21     }
22     point.next = null;
23     return newHead;
24 }
25
26 /**
27  * 数组：用数组按原先顺序存储，再按倒序反向操作
28  * @author PAN
29  * @param head 链表头节点
30  * @return 反转后链表头
31  * @time O(N)
32  * @space O(N)
33  */
34 public ListNode reverseList2(ListNode head){
35     if (head == null) return null;
36     ListNode[] array = new ListNode[5000];
37     int i = 0, len = 0;
38     while (head != null) {
39         array[i] = head;
40         i++;
41         len++;
42         head = head.next;
43     }
44     for (i = len - 1; i > 0; i--) {
45         array[i].next = array[i - 1];
46     }
47     array[0].next = null;
48     return array[len - 1];
49 }

```

2. 双指针 $O(N)$ $O(1)$ - 用前后相差一位的指针反复移动来实现反转；

```

1 /**
2  * 双指针：用前后相差一位的指针反复移动来实现反转
3  * @author PAN
4  * @param head 链表头节点
5  * @return 反转后链表头

```

```

6      * @time O(N)
7      * @space O(1)
8      */
9      public ListNode reverseList3(ListNode head) {
10         if (head == null) return null;
11         ListNode former = head, latter = head.next, tmp;
12         if (latter == null) return head;
13         former.next = null;
14         while (latter != null) {
15             tmp = latter.next;
16             latter.next = former;
17             former = latter;
18             latter = tmp;
19         }
20         return former;
21     }
22
23     /**
24      * 双指针精简版：将两个指针的初始值设为 null, head 代替自己的 head,
25      * head.next 以省略一些多余的判断
26      * @author 网友
27      * @param head 链表头节点
28      * @return 反转后链表头
29      * @time O(N)
30      * @space O(1)
31      */
32     public ListNode reverseList4(ListNode head) {
33         ListNode cur = head, pre = null;
34         while (cur != null) {
35             ListNode tmp = cur.next; // 暂存后继节点 cur.next
36             cur.next = pre;           // 修改 next 引用指向
37             pre = cur;                // pre 暂存 cur
38             cur = tmp;                // cur 访问下一节点
39         }
40         return pre;
41     }

```

3. 递归 $O(N)$ $O(N)$ - 考虑使用递归法遍历链表，当越过尾节点后终止递归，在回溯时修改各节点的 `next` 引用指向。

```

1      /**
2      * 递归：考虑使用递归法遍历链表，当越过尾节点后终止递归，在回溯时修改各节点的
3      * next 引用指向。
4      * @author 网友
5      * @param head 链表头节点
6      * @return 反转后链表头
7      * @time O(N)
8      * @space O(N)
9      */

```

```

9 public ListNode reverseList5(ListNode head) {
10     return recur(head, null);    // 调用递归并返回
11 }
12 private ListNode recur(ListNode cur, ListNode pre) {
13     if (cur == null) return pre; // 终止条件
14     ListNode res = recur(cur.next, cur); // 递归后继节点
15     cur.next = pre;                    // 修改节点引用指向
16     return res;                       // 返回反转链表的头节点
17 }

```

Q25. 合并两个排序的链表：遍历比较合并、递归

输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。

```

1  示例1：
2
3  输入：1->2->4, 1->3->4
4  输出：1->1->2->3->4->4

```

解：

1. 遍历比较合并 $O(N + M)$ $O(1)$ - 遍历逐个比较大小，将小的加入到新建的头节点后面，大的继续比较；

```

1  /**
2   * 遍历比较合并：遍历逐个比较大小，将小的加入到新建的头节点后面，大的继续比较
3   * @author PAN
4   * @param l1 非递减链表 1
5   * @param l2 非递减链表 2
6   * @return 非递减合并链表
7   * @time  $O(M + N)$ 
8   * @space  $O(1)$ 
9   */
10 public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
11     if (l1 == null) return l2;
12     if (l2 == null) return l1;
13     ListNode newHead = l1.val < l2.val ? l1 : l2, point = newHead;
14     if (l1.val < l2.val) l1 = l1.next;
15     else l2 = l2.next;
16
17     while (l1 != null && l2 != null) {
18         if (l1.val < l2.val) {
19             point.next = l1;
20             point = point.next;
21             l1 = l1.next;
22         } else {
23             point.next = l2;
24             point = point.next;

```

```

25         l2 = l2.next;
26     }
27 }
28 if (l1 == null) point.next = l2;
29 if (l2 == null) point.next = l1;
30 return newHead;
31 }

```

2. 递归 $O(N + M)$ $O(1)$;

```

1  /**
2   * 递归
3   * @author 网友
4   * @param l1 非递减链表 1
5   * @param l2 非递减链表 2
6   * @return 非递减合并链表
7   * @time  $O(M + N)$ 
8   * @space  $O(1)$ 
9   */
10 public ListNode mergeTwoLists2(ListNode l1, ListNode l2) {
11     if (l1 == null) {
12         return l2;
13     }
14     if (l2 == null) {
15         return l1;
16     }
17     if (l1.val <= l2.val) {
18         l1.next = mergeTwoLists2(l1.next, l2);
19         return l1;
20     } else {
21         l2.next = mergeTwoLists2(l1, l2.next);
22         return l2;
23     }
24 }

```

Q26. 树的子结构：两个递归

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)B是A的子结构，即 A中有出现和B相同的结构和节点值。

```

1  例如：
2  给定的树 A：
3
4      3
5     / \
6
7    4   5
8   / \

```

```

9      1   2
10     给定的树 B:
11
12         4
13        /
14       1
15     返回 true, 因为 B 与 A 的一个子树拥有相同的结构和节点值。

```

解:

1. 递归 $O(MN)$ $O(M)$ - M : A的节点数, N : B的节点数

1) 先序遍历 A 中每个节点 nA (对应 `isSubStructure(A, B)`) ;

2) 判断树 A 中以 nA 为根节点子树是否包含 B (对应 `recur(A, B)`) 。

```

1  /**
2   * 递归: 1. 先序遍历 A 中每个节点 nA (对应 isSubStructure(A, B)); 2. 判
   断树 A 中以 nA 为根节点子树是否包含 B (对应 recur(A, B)) 。
3   * @author 网友
4   * @param A 二叉树 (节点数 M)
5   * @param B 二叉树 (节点数 N)
6   * @return B 是否是 A 子树
7   * @time O(MN)
8   * @space O(M)
9   */
10 public boolean isSubStructure(TreeNode A, TreeNode B) {
11     if (A == null || B == null) return false; // 空树不为任何树子树
12     return recur(A, B) || isSubStructure(A.left, B) ||
isSubStructure(A.right, B);
13 }
14 public boolean recur(TreeNode A, TreeNode B) {
15     if (B == null) return true;
16     if (A == null) return false;
17     return A.val == B.val && recur(A.left, B.left) &&
recur(A.right, B.right);
18 }

```

Q27. 二叉树的镜像：递归、栈

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

```

      4
     / \
    2   7
   /\  /\
  1 3 6 9

```

镜像输出：

4

```
 / \
7  2
/\  /\
9 63 1
```

```
1  示例 1:
2
3  输入: root = [4,2,7,1,3,6,9]
4  输出: [4,7,2,9,6,3,1]
```

1. 递归 $O(N)$ $O(N)$;

```
1  /**
2   * 递归
3   * @author PAN
4   * @param root 树根
5   * @return 镜像后的树根
6   * @time  $O(N)$ 
7   * @space  $O(N)$ 
8   */
9  public TreeNode mirrorTree(TreeNode root) {
10     if (root == null) return null;
11     return change(root, root.left, root.right);
12 }
13 public TreeNode change(TreeNode root, TreeNode left, TreeNode
right) {
14     if (left == null && right == null) return root; // 叶子节点
15     else if (left != null && right == null) {          // 有左子, 无右
子
16         root.left = null;
17         root.right = change(left, left.left, left.right);
18     }
19     else if (left == null && right != null) {          // 有右子, 无左
子
20         root.left = change(right, right.left, right.right);
21         root.right = null;
22     }
23     else { // 双子
24         root.left = change(right, right.left, right.right);
25         root.right = change(left, left.left, left.right);
26     }
27     return root;
28 }
29
30 /**
31 * 递归 2
32 * @author 网友
```

```

33  * @param root 树根
34  * @return 镜像后的树根
35  * @time O(N)
36  * @space O(N)
37  */
38  public TreeNode mirrorTree2(TreeNode root) {
39      if(root == null) return null;
40      TreeNode tmp = root.left;
41      root.left = mirrorTree2(root.right);
42      root.right = mirrorTree2(tmp);
43      return root;
44  }

```

2. 栈 $O(N)$ $O(N)$ - 从上到下，每次用栈保存一层的节点，然后弹出栈进行左右节点的交换；

```

1  /**
2   * 栈：从上到下，每次用栈保存一层的节点，然后弹出栈进行左右节点的交换
3   * @author 网友 & PAN, 借鉴思路实现
4   * @return 镜像后的树根
5   * @time O(N)
6   * @space O(N)
7   */
8  public TreeNode mirrorTree3(TreeNode root) {
9      if (root == null) return root;
10     Stack<TreeNode> s = new Stack<>();
11     TreeNode node, tmp;
12     s.push(root);
13     while (!s.isEmpty()) {
14         node = s.pop();
15         if (node.left != null) s.push(node.left);
16         if (node.right != null) s.push(node.right);
17         tmp = node.left;
18         node.left = node.right;
19         node.right = tmp;
20     }
21     return root;
22 }

```

Q28. 对称的二叉树：BFS、递归

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```

1  / \
2  2   2
3  / \ / \
4  3  4 4  3

```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

1

```

1  / \
2  2   2
3  \   \
4  3     3

```

```

1  示例 1:
2
3  输入: root = [1,2,2,3,4,4,3]
4  输出: true
5
6  示例 2:
7
8  输入: root = [1,2,2,null,3,null,3]
9  输出: false

```

解:

1. BFS $O(N)$ $O(N)$ - 逐层将该层所有节点 (包括 null) 加入 List, 对 List 进行镜像判断;

```

1  /**
2   * BFS的思想: 逐层将该层所有节点 (包括 null) 加入 List, 对 List 进行镜像判断
3   * @author PAN
4   * @param root 树根
5   * @return 是否是镜像树
6   * @time 感觉是  $O(N)$ , 不太确定
7   * @space  $O(N)$ 
8   */
9  public boolean isSymmetric(TreeNode root) {
10     if (root == null) return true;
11     ArrayList<TreeNode> treeList = new ArrayList<TreeNode>();
12     treeList.add(root);
13     TreeNode point;
14     while (!treeList.isEmpty()) {
15         // 先判断 List 中是否镜像
16         for (int i = 0; i < treeList.size() / 2; i++) {
17             if ((treeList.get(i) != null &&
treeList.get(treeList.size() - 1 - i) != null

```



```

18         && treeList.get(i).val ==
treeList.get(treeList.size() - 1 - i).val)    // List 头尾项都不为
空, 且值相等
19         || (treeList.get(i) == null &&
treeList.get(treeList.size() - 1 - i) == null )) { // 或者List 头尾
项都为空
20             // 满足两个条件则符合镜像规则
21             } else return false;    // 否则不对称
22     }
23     // 在上一层是镜像的情况下, 将下一层节点加入 List, 并将上一层的节点移
出 List
24     int len = treeList.size();
25     for (int i = 0; i < len; len--) {
26         point = treeList.get(i);
27         if (point != null) {
28             treeList.add(point.left);
29             treeList.add(point.right);
30         }
31         treeList.remove(i);
32     }
33 }
34 return true;
35 }

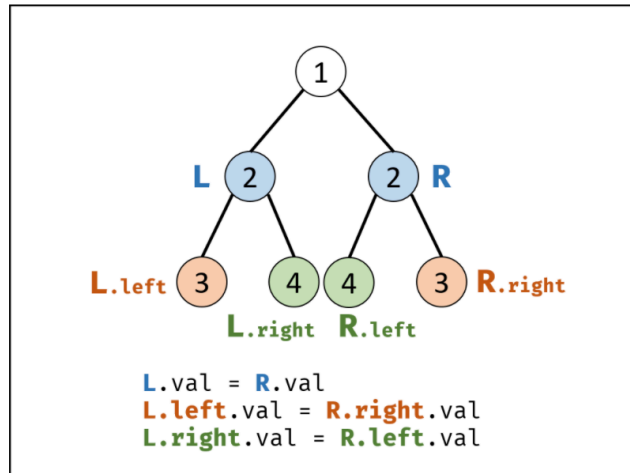
```

2. 递归 $O(N)$ $O(N)$

o

解题思路：

- 对称二叉树定义：对于树中任意两个对称节点 L 和 R ，一定有：
 - $L.val = R.val$ ：即此两对称节点值相等。
 - $L.left.val = R.right.val$ ：即 L 的左子节点和 R 的右子节点对称；
 - $L.right.val = R.left.val$ ：即 L 的右子节点和 R 的左子节点对称。
- 根据以上规律，考虑从顶至底递归，判断每对节点是否对称，从而判断树是否为对称二叉树。



算法流程：

`isSymmetric(root)`：

- 特例处理：若根节点 `root` 为空，则直接返回 `true`。
- 返回值：即 `recur(root.left, root.right)`；

`recur(L, R)`：

- 终止条件：
 - 当 L 和 R 同时越过叶节点：此树从顶至底的节点都对称，因此返回 `true`；
 - 当 L 或 R 中只有一个越过叶节点：此树不对称，因此返回 `false`；
 - 当节点 L 值 \neq 节点 R 值：此树不对称，因此返回 `false`；
- 递推工作：
 - 判断两节点 $L.left$ 和 $R.right$ 是否对称，即 `recur(L.left, R.right)`；
 - 判断两节点 $L.right$ 和 $R.left$ 是否对称，即 `recur(L.right, R.left)`；
- 返回值：两对节点都对称时，才是对称树，因此用与逻辑符 `&&` 连接。

```
1  /**
2   * 递归：任意俩节点 L、R，有 L 左子和 R 右子对称、L 右子和 R 左子对称
3   * @author 网友
4   * @param root 树根
5   * @return 是否是镜像树
6   * @time O(N)
7   * @space O(N)
8   */
9  public boolean isSymmetric2(TreeNode root) {
10     return root == null || recur(root.left, root.right);
11 }
```

```

12 public boolean recur(TreeNode left, TreeNode right) {
13     if (left == null && right == null) return true;
14     if (left == null || right == null || left.val != right.val)
        return false;
15     return recur(left.left, right.right) && recur(left.right,
        right.left);
16 }

```

Q29. 顺时针打印矩阵：设定四边界

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

```

1  示例 1:
2
3  输入: matrix = [[1,2,3],[4,5,6],[7,8,9]]
4  输出: [1,2,3,6,9,8,7,4,5]
5
6  示例 2:
7
8  输入: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
9  输出: [1,2,3,4,8,12,11,10,9,5,6,7]

```

解：

1. 设定四边界 $O(MN)$ $O(1)$ - 设定右、下、左、上边界，一个小人从左上角开始顺时针由外向里走，每转一圈边界缩小一圈；

```

1  /**
2   * 设定四边界：设定右、下、左、上边界，一个小人从左上角开始顺时针由外向里走，每
   * 转一圈边界缩小一圈
3   * @author PAN
4   * @param matrix 二维数组
5   * @return 顺时针由外向里遍历结果
6   * @time  $O(M * N)$ : M: 行数, N: 列数
7   * @space  $O(1)$ : 题目本身要求返回  $int[]$ , 这个数组不算在 space 里
8   */
9  public static int[] spiralOrder(int[][] matrix) {
10     if (matrix == null || matrix.length == 0 || matrix[0].length
        == 0) return new int[0];
11     // m: 右边界, n: 下边界, mm: 左边界, nn: 上边界, 因为从左上顶点开始, 所以
        mm 初始值为 1 (防止重复)
12     int m = matrix.length, n = matrix[0].length, mm = 1, nn = 0;
13     int num = m * n;
14     int[] array = new int[num];
15     int x = 0, y = 0; // 小人坐标(x, y)
16     for (int i = 0; i < num; ) {
17         while (y < n) { // 向右
18             array[i] = matrix[x][y];

```

```

19         i++;
20         y++;
21     }
22     if (i == num) break;    // 已遍历完
23     y--;    // 使得超过下标的 y 复原成不超过下标
24     x++;    // 防止重复添加顶点
25     while (x < m) { // 向下
26         array[i] = matrix[x][y];
27         i++;
28         x++;
29     }
30     if (i == num) break;
31     x--;
32     y--;
33     while (y >= nn) { // 向左
34         array[i] = matrix[x][y];
35         i++;
36         y--;
37     }
38     if (i == num) break;
39     y++;
40     x--;
41     while (x >= mm) { // 向上
42         array[i] = matrix[x][y];
43         i++;
44         x--;
45     }
46     if (i == num) break;
47     x++;
48     y++;
49     // 右、下、左、上一次顺时针后，将边界范围缩小一圈
50     m--;
51     n--;
52     mm++;
53     nn++;
54 }
55 return array;
56 }

```

Q31. 栈的压入、弹出序列：双指针模拟、栈辅助

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否是该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

```

1  示例 1:
2
3  输入: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
4  输出: true
5  解释: 我们可以按以下顺序执行:
6  push(1), push(2), push(3), push(4), pop() -> 4,
7  push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
8
9  示例 2:
10
11 输入: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
12 输出: false
13 解释: 1 不能在 2 之前弹出。

```

解:

1. 双指针模拟 $O(N^2)$ $O(1)$ - 双指针模拟栈操作, 将 pushed 数组当成栈操作: top1、top2 分别指向两个数组模拟栈顶位置, 俩 top 不相等则入栈, 相等则出栈;

```

1  /**
2   * 双指针模拟栈操作, 将 pushed 数组当成栈操作:
3   * 1. top1、top2 分别指向两个数组模拟栈顶位置;
4   * 2. 俩 top 不相等, 则入栈, 相等则出栈
5   * @author PAN
6   * @param pushed 入栈
7   * @param popped 出栈
8   * @return 是否成立
9   * @time  $O(N^2)$ 
10  * @space  $O(1)$  时间换空间, 最大程度节省空间
11  */
12  public static boolean validateStackSequences(int[] pushed, int[]
13  popped) {
14      int top1 = 0, top2 = 0;
15      int len1 = pushed.length, len2 = popped.length;
16      if (len1 == 0) return true;
17      while (top1 < len1 && top2 < len2) {
18          if (top1 < 0) top1++; // 防止 top1 = -1
19          if (pushed[top1] != popped[top2]) { // 入栈
20              top1++;
21          } else { // 出栈
22              top1--;
23              len1--;
24              top2++;
25              for (int i = top1 + 1; i < len1; i++) { // 出栈点后的数
26                  pushed[i] = pushed[i + 1];
27              }
28          }
29      }
30      if (len1 == 0) return true;

```

```

29     }
30     return false;
31 }

```

2. 栈辅助 $O(N)$ $O(N)$ - 利用一个栈辅助，先将 pushed 元素入栈，再判断栈顶与 popped 是否相等，相等则出栈，不等则继续进栈；

```

1  /**
2   * 栈辅助：利用一个栈辅助，先将 pushed 元素入栈，再判断栈顶与 popped 是否相
   * 等，相等则出栈，不等则继续进栈
3   * @author 网友
4   * @param pushed 入栈
5   * @param popped 出栈
6   * @return 是否成立
7   * @time  $O(N)$ 
8   * @space  $O(N)$ 
9   */
10 public static boolean validateStackSequences2(int[] pushed, int[]
   popped) {
11     Stack<Integer> s = new Stack<>();
12     int j = 0;
13     for (int i = 0; i < pushed.length; i++) {
14         s.push(pushed[i]);
15         while (!s.isEmpty() && s.peek() == popped[j]) {
16             s.pop();
17             j++;
18         }
19     }
20     return s.isEmpty();
21 }

```

Q32 - I. 从上到下打印二叉树：队列 BFS

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

```

1  例如：
2  给定二叉树：[3,9,20,null,null,15,7],
3      3
4     / \
5    9  20
6   /  \
7  15   7
8
9  返回：
10 [3,9,20,15,7]

```

解：

1. 队列 BFS O(N) O(N) - 用队列辅助逐行加入节点

```
1  /**
2   * 队列 BFS: 用队列辅助逐行加入节点
3   * @author PAN
4   * @param root 树根
5   * @return 从上到下, 从左到右顺序打印
6   * @time O(N)
7   * @space O(N)
8   */
9  public int[] levelOrder(TreeNode root) {
10     ArrayList<TreeNode> list = new ArrayList();
11     if (root != null) list.add(root);
12     TreeNode point;
13     int i = 0;
14     while (i < list.size()) {
15         point = list.get(i);
16         if (point.left != null) list.add(point.left); // 左子不为
空
17         if (point.right != null) list.add(point.right); // 右子不为
空
18         i++;
19     }
20     int[] treeNode = new int[list.size()];
21     i = 0;
22     for (TreeNode node : list) {
23         treeNode[i] = node.val;
24         i++;
25     }
26     return treeNode;
27 }
```

Q32 - II. 从上到下打印二叉树 II: 队列 BFS、递归 BFS

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

```
1  例如:
2  给定二叉树: [3,9,20,null,null,15,7],
3      3
4     /\
5    9 20
6   /\  \
7  15 7
8
9  返回其层次遍历结果:
10 [
11   [3],
12   [9,20],
```

```
13 | [15,7]
14 | ]
```

解：

1. 队列 BFS O(N) O(N) - 用队列辅助逐行加入节点，用上一问代码改动（加入分层）；

```
1  /**
2   * 队列 BFS - 用队列辅助逐行加入节点，用上一问代码改动（加入分层）
3   * @author PAN
4   * @param root 树根
5   * @return 从上到下从左到右打印节点
6   * @time O(N)
7   * @space O(N)
8   */
9  public List<List<Integer>> levelOrder2(TreeNode root) {
10      List<List<Integer>> treeNodeVal2 = new ArrayList<>();
11      ArrayList<TreeNode> list = new ArrayList<>();
12      if (root != null) list.add(root);
13      TreeNode point;
14      while (!list.isEmpty()) {
15          List<Integer> tmp = new ArrayList<>();
16          int len = list.size();
17          for (int j = len; j > 0; j--){
18              point = list.remove(0);
19              tmp.add(point.val);
20              if (point.left != null) list.add(point.left); // 左子
                不为空
21              if (point.right != null) list.add(point.right); // 右子
                不为空
22          }
23          treeNodeVal2.add(tmp);
24      }
25
26      return treeNodeVal2;
27  }
```

2. 递归 BFS O(N) O(N) - 递归完成 BFS；

```
1  public List<List<Integer>> treeNodeVal = new ArrayList<>();
2
3  /**
4   * 递归 - 递归完成BFS
5   * @author 网友
6   * @param root 树根
7   * @return 从上到下从左到右打印节点
8   * @time O(N)
9   * @space O(N)
10  */
```



```

11 public List<List<Integer>> levelOrder(TreeNode root) {
12     BFS(root, 0);
13     return treeNodeVal;
14 }
15 public void BFS(TreeNode root, int k) { // k 记录层数
16     if (root == null) return;
17     if (treeNodeVal.size() <= k) treeNodeVal.add(new ArrayList<>
18         ());
19     treeNodeVal.get(k).add(root.val);
20     if (root.left != null) BFS(root.left, k + 1);
21     if (root.right != null) BFS(root.right, k + 1);
22 }

```

Q32 - III. 从上到下打印二叉树 III：队列 BFS、递归 BFS

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

```

1  例如：
2  给定二叉树：[3,9,20,null,null,15,7],
3      3
4     /\
5    9 20
6   /\  \
7  15 7
8
9  返回其层次遍历结果：
10 [
11   [3],
12   [20,9],
13   [15,7]
14 ]

```

解：

1. 队列 BFS O(N) O(N) - 用队列辅助逐行加入节点，用上一问代码改动（奇数层向队头添加，偶数层向队尾添加）；

```

1  /**
2   * 队列 BFS - 用队列辅助逐行加入节点，用上一问代码改动（奇数层向队头添加，偶数层向队尾添加）
3   * @author PAN
4   * @param root 树根
5   * @return 奇数层从左到右，偶数层从右到左打印节点
6   * @time O(N)
7   * @space O(N)
8   */
9  public List<List<Integer>> levelOrder2(TreeNode root) {

```

```

10     List<List<Integer>> treeNodeVal2 = new ArrayList<>();
11     ArrayList<TreeNode> list = new ArrayList();
12     if (root != null) list.add(root);
13     TreeNode point;
14     while (!list.isEmpty()) {
15         List<Integer> tmp = new ArrayList<>();
16         for (int j = list.size(); j > 0; j--){
17             point = list.remove(0);
18             // 奇数层向队头添加, 欧偶数层向队尾添加
19             if (treeNodeVal2.size() % 2 == 1) tmp.add(0,
point.val);
20             else tmp.add(point.val);
21             if (point.left != null) list.add(point.left); // 左子
不为空
22             if (point.right != null) list.add(point.right); // 右子
不为空
23         }
24         // 或者用倒序函数
25         // if (treeNodeVal2.size() % 2 == 1)
Collections.reverse(tmp);
26         treeNodeVal2.add(tmp);
27     }
28     return treeNodeVal2;
29 }

```

2. 递归 BFS O(N) O(N) - 递归完成BFS, 奇数层向队头添加, 欧偶数层向队尾添加;

```

1  public List<List<Integer>> treeNodeVal = new ArrayList<>();
2
3  /**
4   * 递归 - 递归完成BFS, 奇数层向队头添加, 欧偶数层向队尾添加
5   * @author 网友
6   * @param root 树根
7   * @return 奇数层从左到右, 偶数层从右到左打印节点
8   * @time O(N)
9   * @space O(N)
10  */
11  public List<List<Integer>> levelOrder(TreeNode root) {
12      BFS(root, 0);
13      return treeNodeVal;
14  }
15  public void BFS(TreeNode root, int k) {
16      if (root == null) return;
17      if (treeNodeVal.size() <= k) treeNodeVal.add(new ArrayList<>
());
18      // 奇数层向队头添加, 欧偶数层向队尾添加
19      if (k % 2 == 1) treeNodeVal.get(k).add(0, root.val);
20      else treeNodeVal.get(k).add(root.val);
21  }

```

```

22     BFS(root.left, k + 1);
23     BFS(root.right, k + 1);
24 }

```

Q33. 二叉搜索树的后序遍历序列

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 `true`，否则返回 `false`。假设输入的数组的任意两个数字都互不相同。

1 参考以下这颗二叉搜索树：

```

2       5
3      / \
4     2   6
5    / \
6   1   3

```

7
8 示例 1:

9
10 输入: [1,6,3,2,5]

11 输出: false

12
13 示例 2:

14
15 输入: [1,3,2,6,5]

16 输出: true

解:

1. 递归 $O(N^2)$ $O(N)$:

```

1  /**
2   * 递归:
3   * 1. 二叉搜索树左子都比 root 小, 右子都比 root 大;
4   * 2. 数组最后一个元素为 root, 从头开始找到一直比 root 小的范围数组下标
   left, 再找比 root 大的范围数组 right;
5   * 3. 若找到的 right != len - 1, 意味不符合二叉搜索树;
6   * @author PAN
7   * @param postorder 后序遍历数组
8   * @return 后序遍历是否是二叉搜索树
9   * @time  $O(N^2)$ 
10  * @space  $O(N)$ 
11  */
12 public static boolean verifyPostorder(int[] postorder) {
13     // 为空或者长度为 0 意味到达了叶子节点, 返回 true
14     if (postorder == null || postorder.length == 0) return true;
15     int len = postorder.length;
16     int root = postorder[len - 1];
17     // 二叉搜索树左子都比 root 小, 右子都比 root 大

```

```

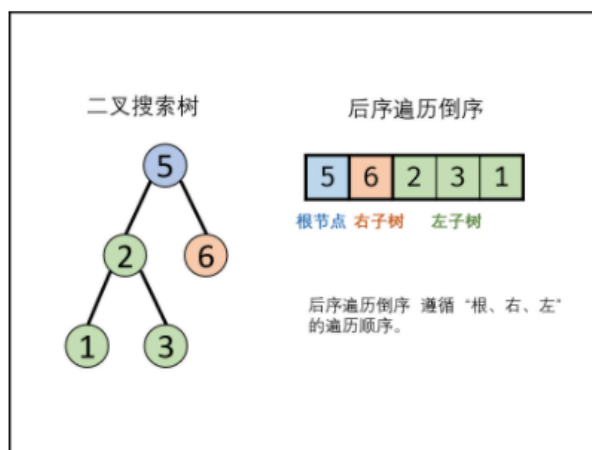
18     int left = 0;
19     while (postorder[left] < root) left++;
20     int right = left;
21     while (postorder[right] > root) right++;
22
23     if (right == len - 1) return
verifyPostorder(Arrays.copyOfRange(postorder, 0, left)) &&
verifyPostorder(Arrays.copyOfRange(postorder, left, right));
24     else return false; // 不是二叉搜索树
25 }

```

2. 辅助单调栈 $O(N)$ $O(N)$

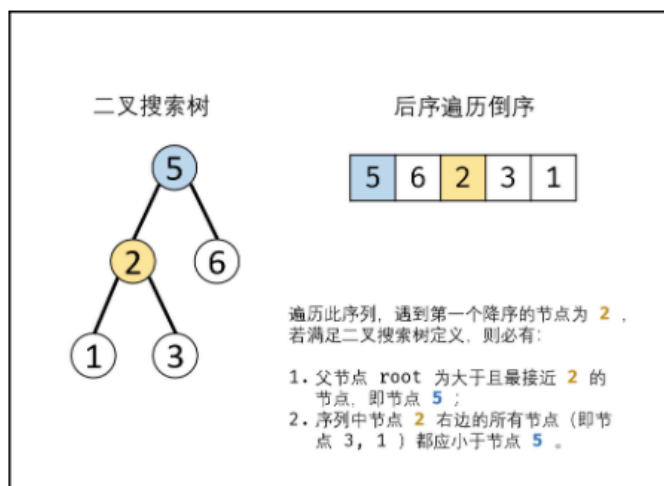
o

- **后序遍历倒序：** [根节点 | 右子树 | 左子树]。类似 先序遍历的镜像，即先序遍历为“根、左、右”的顺序，而后序遍历的倒序为“根、右、左”顺序。



- 设后序遍历倒序列表为 $[r_n, r_{n-1}, \dots, r_1]$ ，遍历此列表，设索引为 i ，若为 二叉搜索树，则有：
 - 当节点值 $r_i > r_{i+1}$ 时：节点 r_i 一定是节点 r_{i+1} 的右子节点。
 - 当节点值 $r_i < r_{i+1}$ 时：节点 r_i 一定是某节点 $root$ 的左子节点，且 $root$ 为节点 $r_{i+1}, r_{i+2}, \dots, r_n$ 中值大于且最接近 r_i 的节点（ $\because root$ 直接连接左子节点 r_i ）。
- 当遍历时遇到递减节点 $r_i < r_{i+1}$ ，若为二叉搜索树，则对于后序遍历中节点 r_i 右边的任意节点 $r_x \in [r_{i-1}, r_{i-2}, \dots, r_1]$ ，必有节点值 $r_x < root$ 。

节点 r_x 只可能为以下两种情况：① r_x 为 r_i 的左、右子树的各节点；② r_x 为 $root$ 的父节点或更高层父节点的左子树的各节点。在二叉搜索树中，以上节点都应小于 $root$ 。



- 遍历“后序遍历的倒序”会多次遇到递减节点 r_i ，若所有的递减节点 r_i 对应的父节点 $root$ 都满足以上条件，则可判定为二叉搜索树。
- 根据以上特点，考虑借助 **单调栈** 实现：
 1. 借助一个单调栈 $stack$ 存储值递增的节点；
 2. 每当遇到值递减的节点 r_i ，则通过出栈来更新节点 r_i 的父节点 $root$ ；
 3. 每轮判断 r_i 和 $root$ 的值关系：
 1. 若 $r_i > root$ 则说明不满足二叉搜索树定义，直接返回 $false$ 。
 2. 若 $r_i < root$ 则说明满足二叉搜索树定义，则继续遍历。

算法流程：

1. **初始化**：单调栈 $stack$ ，父节点值 $root = +\infty$ （初始值为正无穷大，可把树的根节点看为此无穷大节点的左孩子）；
2. **倒序遍历 $postorder$** ：记每个节点为 r_i ；
 1. **判断**：若 $r_i > root$ ，说明此后序遍历序列不满足二叉搜索树定义，直接返回 $false$ ；
 2. **更新父节点 $root$** ：当栈不为空 且 $r_i < stack.peek()$ 时，循环执行出栈，并将出栈节点赋给 $root$ 。
 3. **入栈**：将当前节点 r_i 入栈；
3. 若遍历完成，则说明后序遍历满足二叉搜索树定义，返回 $true$ 。

```
1  /**
2   * 辅助单调栈：后序遍历的倒序（根、右、左）类似先序遍历的镜像；
3   * @author 网友
4   * @param postorder 后序遍历数组
5   * @return 后序遍历是否是二叉搜索树
6   * @time O(N)
7   * @space O(N)
8   */
9  public boolean verifyPostorder2(int[] postorder) {
10     Stack<Integer> stack = new Stack<>();
11     int root = Integer.MAX_VALUE;
12     for(int i = postorder.length - 1; i >= 0; i--) {
13         if(postorder[i] > root) return false;
14         while(!stack.isEmpty() && stack.peek() > postorder[i])
15             root = stack.pop();
16         stack.add(postorder[i]);
17     }
18     return true;
19 }
```

