剑指Offer(第2版)

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～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。

```
输入：
[2, 3, 1, 0, 2, 5, 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。

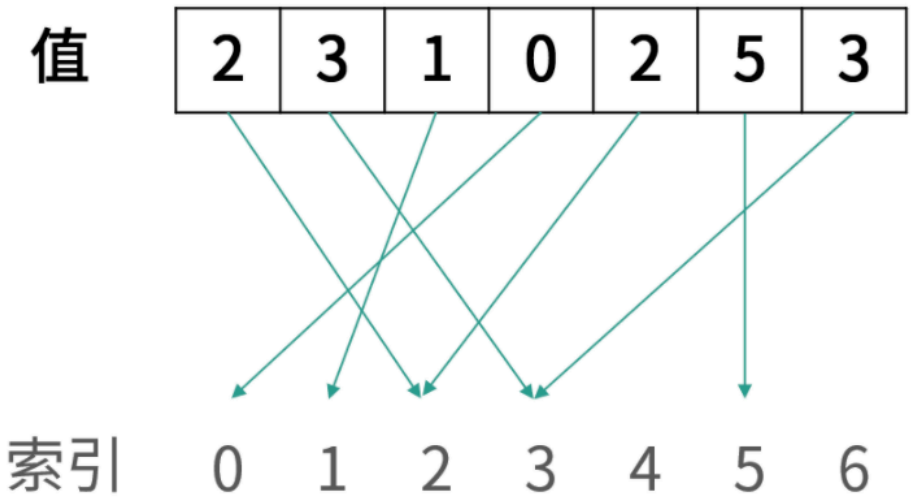
```java
/**
 * 遍历 nums[]，利用 countArray[] 记录数出现的次数
 * @author PAN
 * @param nums int[]
 * @return repeatNumber：第一个重复出现的数字
 */
public static int findReapeatNumber(int[] nums) {
    int[] countArray = new int[nums.length];
    int repeatNumber = -1;
    for(int num : nums) {
        countArray[num]++;
        if(countArray[num] > 1) {
            repeatNumber = num;
            break;
        }
    }
    return repeatNumber;
}
```

2. 排序后再找重 - 排序 nums[]，再遍历找到第一个重复数字。

```java
/**
 * 排序 nums[]，再遍历找到第一个重复数字
 * @author PAN
 * @param nums int[]
 * @return repeatNumber：第一个重复出现的数字
 */
public static int findReapeatNumber2(int[] nums) {
    int repeatNumber = -1;
    Arrays.sort(nums);
    int record = nums[0];
    for(int i = 1; i < nums.length; i++) {
        if(nums[i] == record) {
            repeatNumber = record;
            break;
```

```
        } else {
            record = nums[i];
        }
    }
    return repeatNumber;
}
```

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 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～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

```
/**
 * 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 nums[x] = x ，此时即可得到一组重复数字。
 * @author 网友
 */
public static int findReapeatNumber0(int[] nums) {
    int i = 0;
    while(i < nums.length) {
        if(nums[i] == i) {
            i++;
            continue;
        }
        if(nums[nums[i]] == nums[i]) return nums[i];
        int tmp = nums[i];
        nums[i] = nums[tmp];
        nums[tmp] = tmp;
    }
```

```
        return -1;
    }
```

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。

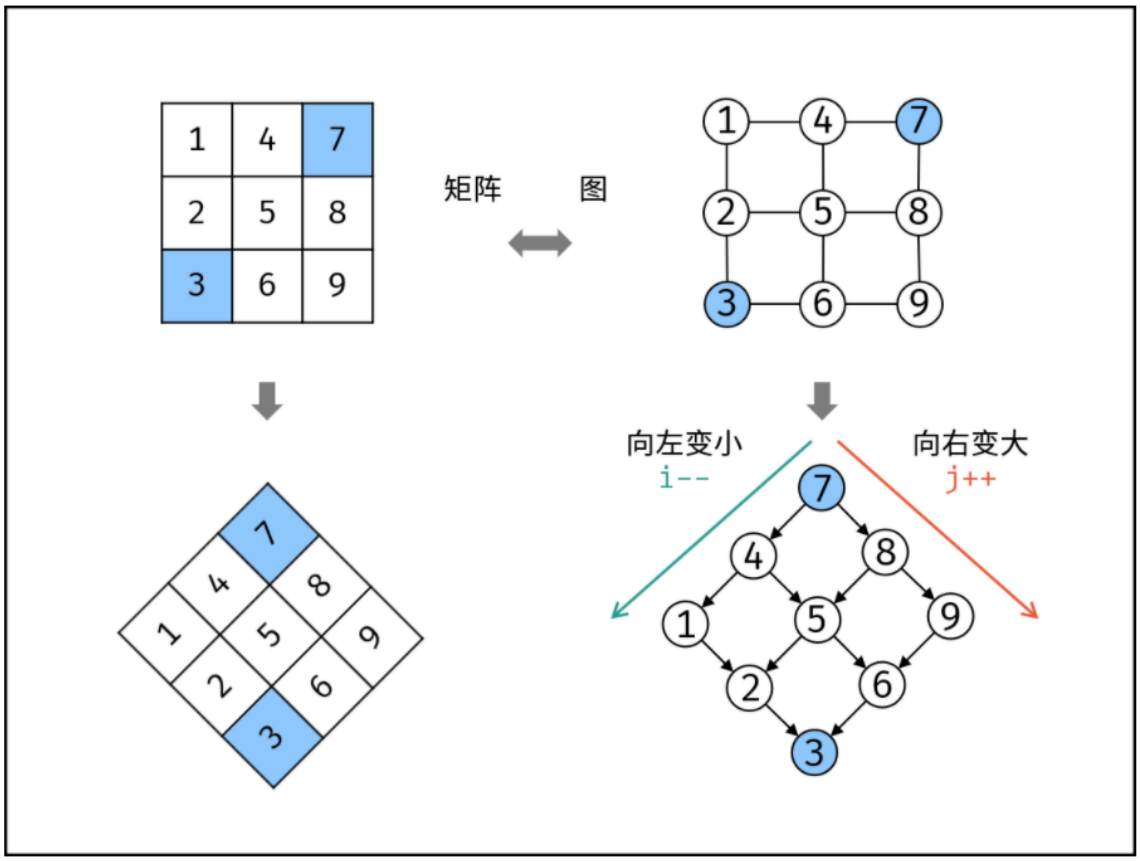
```
示例：

现有矩阵 matrix 如下：

[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]

给定 target = 5，返回 true。
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。

```
/**
 * 暴力求解
 * @author PAN
 * @param matrix int[][]
 * @param target int
 * @return true/false：查找结果
 * @time O(N * M)
 */
public static boolean findNumberIn2DArray(int[][] matrix, int target) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (target == matrix[i][j]) {
                return true;
            }
        }
    }
    return false;
}
```

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。




```
/**
 * 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
 * @author 网友
 * @param matrix 二维数组
 * @param target 查找目标
 * @return true/false
 * @time O(N + M)
 */
public static boolean findNumberIn2DArray2(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    } else {
        int i = 0, j = matrix[0].length - 1;
        do {
            if (target < matrix[i][j]) {
                j--;
            } else if (target == matrix[i][j]) {
                return true;
            } else {
                i++;
```

```
            }
        } while (i < matrix.length && j > -1);

        return false;
    }
}
```

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"。

```
输入：s = "We are happy."
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"。

```
/**
 * 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"
 * @author PAN
 * @param s 原字符串
 * @return 替换后字符串
 * @time O(N)
 * @space O(N)
 */
public static String replaceSpace(String s) {
    StringBuilder newS = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if(c != ' ') {
            newS.append(c);
        } else {
            newS.append("%20");
        }
    }

    return newS.toString();
}
```

2. 调用`replace()`

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

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）。

```
输入：head = [1,3,2]
输出：[2,3,1]
```

解：

1. 栈 O(N) O(N) - 先入后出实现从尾到头打印。

```java
/**
 * 栈：先入后出实现从尾到头打印
 * @author PAN
 * @param head 单链表头
 * @return 反转后单链表（以数组形式）
 * @time O(N)
 * @space O(N)
 */
public static int[] reversePrint(ListNode head) {
    Stack<ListNode> s = new Stack<ListNode>();
    while (head != null) {
        s.push(head);
        head = head.next;
    }
    int len = s.size();
    int[] array = new int[len];
    for (int i = 0; i < len; i++) {
        array[i] = s.pop().val;
    }
    return array;
}
```

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。

```
例如，给出

前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

    3
   / \
  9  20
    /  \
   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 nullnull ；
   - **递推工作**：
     1. 建立根节点 node ： 节点值为 preorder[root] ；
     2. 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)O(1)
     3. 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；

```
/**
 * 递归重建二叉树
 * @author 网友
 * @param root 前序中根索引
 * @param left 前序中左子树索引
 * @param right 前序中右子树索引
 * @return 树结点
 * @time O(N)
 * @space O(N)
 */
```

```java
TreeNode recur(int root, int left, int right) {
    if (left > right) return null;
    TreeNode node = new TreeNode(preorder[root]);
    int i = dic.get(preorder[root]);
    node.left = recur(root + 1, left, i -1);
    node.right = recur(i - left + root + 1, i + 1, right);
    return node;
}
```

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)

```
示例 1：

输入：
["CQueue","appendTail","deleteHead","deleteHead"]
[[],[3],[],[]]
输出：[null,null,3,-1]

示例 2：

输入：
["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"
]
[[],[],[5],[2],[],[]]
输出：[null,-1,null,null,5,2]
```

解：

1. 一栈负责进，一栈复杂出 O(N) O(N)

```java
/**
 * 一栈负责进，一栈复杂出
 * @author PAN
 * @return 队列头
 * @time O(N)
 * @space O(N)
 */
public int deleteHead() {
    if (s2.size() == 0) {
        if (s1.size() == 0) return -1;
        else {
            while (!s1.isEmpty()) {
                s2.push(s1.pop());
            }
        }
    }
    return s2.pop();
}
```

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), 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。

```
示例 1:

输入: n = 2
输出: 1

示例 2:

输入: n = 5
输出: 5
```

解：

1. 递归 - 效率低，存在大量重复计算；

```java
/**
 * 递归求解 - 将斐波那契公式转换为递归形式
 * @author PAN
```

```
 * @param n 数列第 n 项
 * @return 斐波那契值
 */
public static long fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else {
        if (fib(n - 1) + fib(n - 2) > 1000000007) return (fib(n - 1) + fib(n - 2)) % 1000000007;
        else return fib(n - 1) + fib(n - 2);
    }
}
```

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**

```
/**
 * 动态规划 - 某一项等于前两项之和，用循环依次计算。【注意！！！】先求余与最后求余返回结果一致，但先求可以防止 int 溢出
 * @author PAN
 * @param n
 * @return
 * @time O(N)
 * @space O(1)
 */
public static int fib2(int n) {
    int a = 0, b = 1;
    switch (n) {
        case 0: return 0;
        case 1: return 1;
        default: {
            int tmp;
            for (int i = 1; i < n; i++) {
                tmp = (a + b) % 1000000007;
                a = b;
                b = tmp;
            }
            return b;
        }
    }
}

/**
 * 动态规划 - 精简代码
 * @author 网友
 * @param n
 * @return
 */
public static int fib3(int n) {
```

```
    int a = 0, b = 1, sum;
    for(int i = 0; i < n; i++){
        sum = (a + b) % 1000000007;
        a = b;
        b = sum;
    }
    return a;
}
```

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。

```
示例 1：

输入：n = 2
输出：2

示例 2：

输入：n = 7
输出：21

示例 3：

输入：n = 0
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)。

```java
public static int numWays(int n) {
    int a = 1, b = 1, sum;
    for (int i = 0; i < n; i++) {
        sum = (a + b) % 1000000007;
        a = b;
        b = sum;
    }
    return a;
}
```

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。

```
示例 1：

输入：[3,4,5,1,2]
输出：1

示例 2：

输入：[2,2,2,0,1]
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，之后便利数组，一一与min比较，第一个比min小的即是结果；

```java
/**
 * 遍历直到第一个变小的数
 * @author PAN
 * @param numbers 两个非递减数列构成的数组
 * @return min
 * @time O(N)
 * @space O(1)
 */
public static int minArray(int[] numbers) {
    int min = numbers[0];
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
            break;
        }
    }
    return min;
```

```
}
```

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 = 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）

```java
/**
 * 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
 * @author 网友
 * @param numbers 两个非递减数列构成的数组
 * @return min
 * @time O(logN)
 * @space O(1)
 */
public static int minArray2(int[] numbers){
    int i = 0, j = numbers.length - 1;
    int m = 0;
    while (i < j) {
        m = (i + j) / 2;
        if (numbers[m] > numbers[j]) i = m + 1;
        else if (numbers[m] == numbers[j]) j--;
        else {
            int min = numbers[i];
            for (int k = i + 1; k < m + 1; k++) {
                if (numbers[k] < min){
                    min = numbers[k];
                    break;
                }
            }
            return min;
        }
    }
    return numbers[i];
}
```

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）。

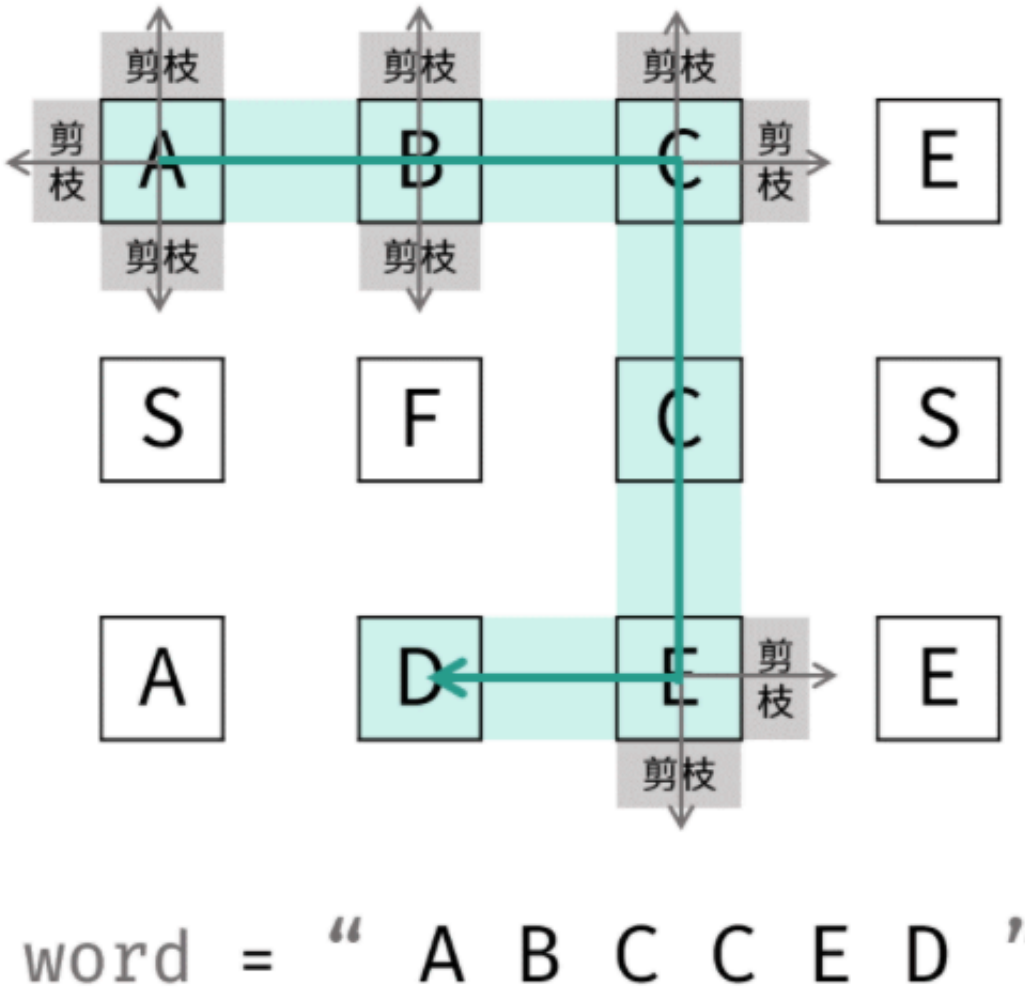
[["a","b","c","e"],
["s","f","c","s"],
["a","d","e","e"]]

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。

```
示例 1：

输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
输出：true

示例 2：

输入：board = [["a","b"],["c","d"]], word = "abcd"
输出：false
```

解：

1. **DFS + 剪枝 O(3^K*MN) O(K)**：



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；

- 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)

```java
boolean existPath(char[][] board, String word) {
    char[] wordArray = word.toCharArray();
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[i].length; j++) {
            if (dfs(board, wordArray, i, j, 0)) return true;
        }
    }
    return false;
}
boolean dfs(char[][] board, char[] word, int i, int j, int k) {
    if (i < 0 || i >= board.length || j < 0 || j >= board[i].length || board[i][j] != word[k]) return false;
    if (k == word.length - 1) return true;
    board[i][j] = '\0';
    boolean result = dfs(board, word, i, j - 1, k + 1) || dfs(board, word, i + 1, j, k + 1) ||
            dfs(board, word, i, j + 1, k + 1) || dfs(board, word, i - 1, j, k + 1);
    board[i][j] = word[k];
    return result;
}
```

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?

```
示例 1:

输入: m = 2, n = 3, k = 1
输出: 3

示例 2:

输入: m = 3, n = 1, k = 0
输出: 1
```

解：

1. 深度优先遍历DFS O(MN) O(MN)
    - **数位和增量公式：不需要每次都用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 $x\pm1$），因此每次只需计算 x 到 $x\pm1$ 的**数位和增量**。本题说明 $1\le n,m\le100$，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** $(x+1)\odot10=0$ **时：** $s_{x+1}=s_x-8$，例如 $19,20$ 的数位和分别为 $10,2$；
      2. **当** $(x+1)\odot10\neq0$ **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

```c
/**
 * 递归求解，与Q12类似，仅需向右、向下移动
 * @author PAN
 * @param m 行数
 * @param n 列数
 * @param k 位数和上限
 * @return 可达数
 * @time O(MN)
 * @space O(MN)
 */
int movingCount(int m, int n, int k) {
    DFS(m, n, 0, 0, k);
    return xy.size();
}

/**
 * DFS：递归函数
 * @author PAN
 * @param m 行数
 * @param n 列数
 * @param i 初始位置
 * @param j 初始位置
 * @param k 位数和上限
 */
```

```java
void DFS(int m, int n, int i, int j, int k) {
    int sum = i / 100 + i / 10 + i % 10 + j / 100 + j / 10 + j % 10;
    boolean flag = sum > k ? false : true;
    String tmp = i + "," + j;
    if (i < 0 || i >= m || j < 0 || j >= n || !flag || xy.contains(tmp)) return;
    xy.add(tmp);
    DFS(m, n, i, j + 1, k);
    DFS(m, n, i + 1, j, k);

    // 优化，不必向上、向左，也意味不需要判断 i < 0 || j < 0
    // DFS(m, n, i, j - 1, k);
    // DFS(m, n, i - 1, j, k);
}

/**
 * 递归求解优化：用 visited 数组记录是否访问，并精简代码
 * @author PAN
 * @param m 行数
 * @param n 列数
 * @param k 位数和上限
 * @return 可达数
 * @time O(MN)
 * @space O(MN)
 */
int movingCount2(int m, int n, int k) {
    boolean[][] visited = new boolean[m][n];
    return DFS2(m, n, 0, 0, k, visited);
}
int DFS2(int m, int n, int i, int j, int k, boolean[][] visited){
    int sum = i / 100 + i / 10 + i % 10 + j / 100 + j / 10 + j % 10;
    if (i >= m || j >= n || sum > k || visited[i][j]) return 0;
    visited[i][j] = true;
    return DFS2(m, n, i, j + 1, k, visited) + DFS2(m, n, i + 1, j, k, visited) + 1;
}
```

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

```java
/**
 * BFS：用队列实现，重点在于数位和增量公式(不需要每次都用整除和求余去计算位数和)
 * @author 网友
 * @param m 行数
 * @param n 列数
 * @param k 位数和上限
 * @return 可达数
 * @time O(MN)
 * @space O(MN)
 */
public int movingCount3(int m, int n, int k) {
    boolean[][] visited = new boolean[m][n];
    int res = 0;
    Queue<int[]> queue= new LinkedList<int[]>();
    queue.add(new int[] { 0, 0, 0, 0 });
    while(queue.size() > 0) {
        int[] x = queue.poll();
        int i = x[0], j = x[1], si = x[2], sj = x[3];
        if(i >= m || j >= n || k < si + sj || visited[i][j]) continue;
        visited[i][j] = true;
        res ++;
        queue.add(new int[] { i + 1, j, (i + 1) % 10 != 0 ? si + 1 : si - 8, sj }); // 数位和增量公式
        queue.add(new int[] { i, j + 1, si, (j + 1) % 10 != 0 ? sj + 1 : sj - 8 });
    }
    return res;
}
```

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]...k[m-1] 。请问 k[0]*k[1]*...*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。

```
示例 1：

输入：2
输出：1
解释：2 = 1 + 1, 1 × 1 = 1

示例 2：

输入：10
输出：36
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分成长度3，乘积越大；

     切分规则：

     1. **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。
     2. **次优：** 2 。若最后一段绳子长度为 2 ；则保留，不再拆为 $1+1$ 。
     3. **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 $2\times2>3\times1$。

   - 那么转换成以下算法：

     1）3a + b = n，当 n <= 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；

```java
/**
 * 数学推导/贪心：
 *  1. 绳子越等分乘积越大；
 *  2. 越多切分成长度 3 ，乘积越大；
 *  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；
 * @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）
 * @param n 绳子长度
 * @return 最大乘积
 * @time O(1)
 * @space O(1)
 */
public static int cuttingRope(int n) {
    if (n <= 3) return n - 1;
    int a = n / 3, b = n % 3;
    if(b == 0) return (int) Math.pow(3, a);
    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 $n=n1+n2$ 。
> 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 $n1\times n2>n1+n2=n$ 。
>
> - **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** $n=n1+n2$ ，**切分为三段** $n=n1+n2+n3$ 。
> 根据经验推测，**三段** 的乘积往往更大，即往往有 $n1n2n3>n1n2$ 。
>
> - **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. 我们想要求长度为 `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

```java
class Solution {
    public int cuttingRope(int n) {
        int[] dp = new int[n + 1];
        dp[2] = 1;
        for(int i = 3; i < n + 1; i++){
            for(int j = 2; j < i; j++){
                dp[i] = Math.max(dp[i], Math.max(j * (i - j), j * dp[i - j]));
            }
        }
        return dp[n];
    }
}
```

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]...k[m - 1] 。请问 k[0]*k[1]*...*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。

```
示例 1:

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1

示例 2:

输入: 10
输出: 36
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；

```java
/**
 * 循环求余
 * @author PAN
 * @param n 绳子长度
 * @return 最大乘积
 * @time O(N)
 * @space O(1)
 */
public static int cuttingRope(int n) {
    if (n <= 3) return n - 1;
```

```
        int a = n / 3, b = n % 3;
        long mul = 1;
        int r = 1000000007;
        for (int i = 0; i < a - 1; i++) {
            mul = (mul * 3) % r;
        }
        if(b == 0) return (int)(mul * 3 % r);
        else if(b == 1) return (int)((4 * mul) % r);
        else return (int)(mul * 6 % r);
    }
```

2. **快速求余 O(log2 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。

```
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        int b = n % 3, p = 1000000007;
        long rem = 1, x = 3;
        for(int a = n / 3 - 1; a > 0; a /= 2) {
            if(a % 2 == 1) rem = (rem * x) % p;
            x = (x * x) % p;
        }
        if(b == 0) return (int)(rem * 3 % p);
        if(b == 1) return (int)(rem * 4 % p);
        return (int)(rem * 6 % p);
    }
}
```

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。

```
示例 1：

输入：00000000000000000000000000001011
输出：3
```

```
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

示例 2：

输入：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。

示例 3：

输入：11111111111111111111111111111101
输出：31
解释：输入的二进制串 11111111111111111111111111111101 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。

```java
/**
 * 巧用 n & (n - 1)
 * (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 。
 * n & (n - 1)： 二进制数字 n 最右边的 1 变成 0 ，其余不变。
 * @author 网友 & PAN，借鉴 n & (n - 1) 的思想
 * @param n 无符号二进制数
 * @return 1 的个数
 * @time O(M) M 为 1 的个数
 * @space O(1)
 */
public static int hammingWeight(int n) {
    int count = 0;
    while (n != 0) {
        n = n & (n - 1);
        count++;
    }
    return count;
}
```

2. 逐位判断 O(log2 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。

```java
/**
 * 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。
 * @author 网友
 * @param n 无符号二进制数
 * @return 1 的个数
 * @time O(log2 N) M 为 1 的个数
 * @space O(1)
```

```
 */
public static int hammingWeight2(int n) {
    int count = 0;
    while (n != 0) {
        count += n & 1;
        n >>>= 1; // 这里需要写成>>>，Java中的无符号右移
        // n = n >> 1; // 自己的写法
    }
    return count;
}
```

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 pow(x, n) ，即计算 x 的 n 次幂函数（即，xn）。不得使用库函数，同时不需要考虑大数问题。

```
示例 1:

输入: x = 2.00000, n = 10
输出: 1024.00000
示例 2:

输入: x = 2.10000, n = 3
输出: 9.26100
示例 3:

输入: x = 2.00000, n = -2
输出: 0.25000
解释: 2-2 = 1/22 = 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；

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

```
    if (x == 0 || x == 1) return x;
    double pow = 1;
    if (n < 0) {
        n = n * (-1);
        x = 1.0 / x;
    }
    if (x == -1 && n % 2 == 0) return -x;
    else if (x == -1 && n % 2 != 0) return x;
    if (n < 0) return 0;
    for (int i = 0; i < n; i++) {
        pow *= x;
        if (pow == 0) break;
    }
    return pow;
}
```

2. **二进制快速幂 O(log2 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 "$b_m...b_3b_2b_1$"（ b_i 为二进制某位值，$i \in [1,m]$ ），则有：
  - **二进制转十进制：** $n = 1b_1 + 2b_2 + 4b_3 + ... + 2^{m-1}b_m$ *（即二进制转十进制公式）* ；
  - **幂的二进制展开：** $x^n = x^{1b_1+2b_2+4b_3+...+2^{m-1}b_m} = x^{1b_1}x^{2b_2}x^{4b_3}...x^{2^{m-1}b_m}$ ；
- 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 $x^1, x^2, x^4, ..., x^{2^{m-1}}$ 的值：** 循环赋值操作 $x = x^2$ 即可；
  - **获取二进制各位 $b_1, b_2, b_3, ..., b_m$ 的值：** 循环执行以下操作即可。
    1. $n\&1$ **（与操作）：** 判断 n 二进制最右一位是否为 1 ；
    2. $n >> 1$ **（移位操作）：** n 右移一位（可理解为删除最后一位）。

```
/**
 * 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
 * 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
 * @author 网友
 * @param x 底数
 * @param n 幂次
 * @return x ^ n
 * @time O(log2 N)
 * @space O(1)
 */
public static double myPow2(double x, int n) {
    if (x == 0) return x;
    double pow = 1.0;
    long newN = n;
    if (newN < 0) {
        newN = -newN;
        x = 1.0 / x;
    }
    while (newN != 0) {
        if ((newN & 1) == 1) pow *= x;
```

```
            x *= x;
            newN >>= 1;
        }
        return pow;
    }
```

3. **二分法快速幂 O(log2 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。

```
示例 1:

输入: n = 1
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。

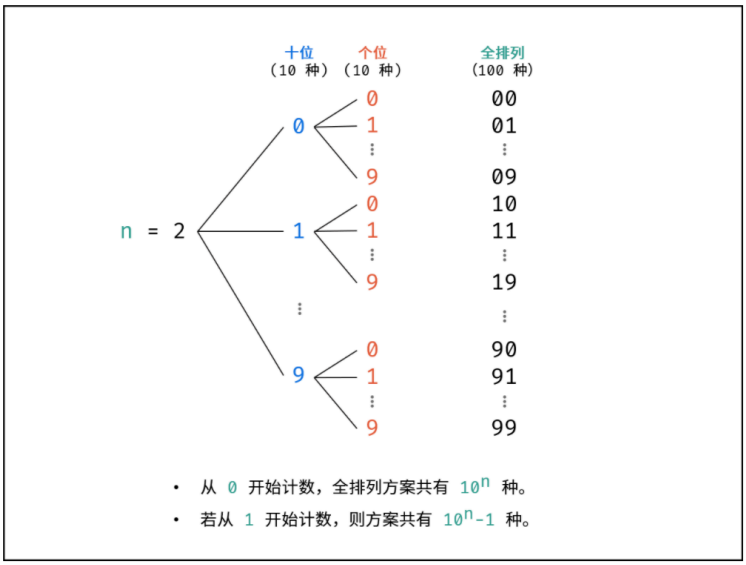
```
/**
 * 循环打印：直接构建数组循环输出
 * @author PAN
 * @param n 最大位数
 * @return 从 1 到最大的 n 位十进制数
 * @time O(10^N)
 * @space O(1)
 */
public static int[] printNumbers(int n) {
    int max = (int) Math.pow(10, n) - 1;
    int[] printArray = new int[max];
    for (int i = 0; i < max; i++) {
        printArray[i] = i + 1;
    }
    return printArray;
}
```

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。

```java
/**
 * 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。
 * @author 网友 & PAN，根据思路改了一些写法，
 * 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
 * @param n 最大位数
 * @return 从 1 到最大的 n 位十进制数
 * @time O(10^N)
 * @space O(10^N)
 */
public int[] printNumbers2(int n) {
    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。

```java
/**
 * 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。
 * @author PAN
 * @param head 头节点
 * @param val 待删除节点的值
 * @return 头节点
 * @time O(N)
 * @space O(1)
 */
public ListNode deleteNode(ListNode head, int val) {
    if (head.val == val) {
        head = head.next;
        return head;
    }
    ListNode point = head;
    while (point.next != null) {
        if (point.next.val == val) point.next = point.next.next;
        else point = point.next;
    }
    return head;
}
```

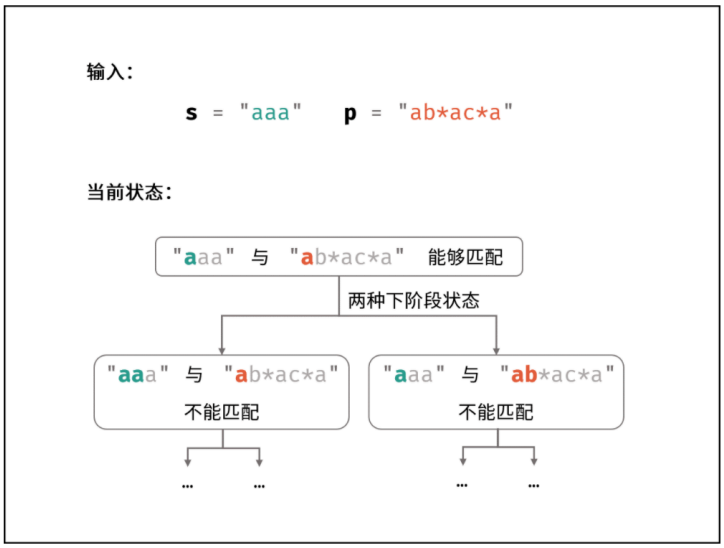
Q19. 正则表达式匹配：动态规划

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。

```
示例 1:

输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入:
s = "aa"
p = "a*"
输出: true
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。

示例 3:

```

```
输入：
s = "ab"
p = ".*"
输出：true
解释："." 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4：

输入：
s = "aab"
p = "c*a*b"
输出：true
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"。

示例 5：

输入：
s = "mississippi"
p = "mis*is*p*."
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。

```java
/**
 * 动态规划：每轮添加一个字符并判断是否能匹配，直至添加完整个字符串 s 和 p
 * @author 网友
 * @param s 字符串
 * @param p 正则表达式
 * @return 匹配成功与否
 * @time O(NM)
 * @space O(NM)
 */
public static boolean isMatch2(String s, String p) {
    boolean[][] dp = new boolean[s.length() + 1][p.length() + 1];
    dp[0][0] = true;
    for (int j = 2; j < p.length() + 1; j += 2) {
        dp[0][j] = dp[0][j - 2] && p.charAt(j - 1) == '*';
    }
    for (int i = 1; i < s.length() + 1; i++) {
        for (int j = 1; j < p.length() + 1; j++) {
            if (p.charAt(j - 1) == '*') {
                if (dp[i][j - 2]) dp[i][j] = true;
                else if (dp[i - 1][j] && s.charAt(i - 1) ==
p.charAt(j - 2)) dp[i][j] = true;
                else if (dp[i - 1][j] && p.charAt(j - 2) == '.')
dp[i][j] = true;
            } else {
                if (dp[i - 1][j - 1] && s.charAt(i - 1) ==
p.charAt(j - 1)) dp[i][j] = true;
```

```
                else if (dp[i - 1][j - 1] && p.charAt(j - 1) ==
'.') dp[i][j] = true;
            }
        }
    }
    return dp[s.length()][p.length()];
}
```

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；

```
public static boolean isNumber(String s) {
    boolean hasNum = false, hasDecimal = false, hasE = false; //
是否有数字、小数、e/E
    s = s.trim(); // 删除前后多余空格
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) >= '0' && s.charAt(i) <= '9') hasNum =
true;
        else if (s.charAt(i) == '.' && !hasDecimal && !hasE) {
            hasDecimal = true;
        } else if ((s.charAt(i) == 'e' || s.charAt(i) == 'E') &&
!hasE && hasNum){
            hasE = true;
            hasNum = false;
        } else if ((s.charAt(i) == '+' || s.charAt(i) == '-') &&
(i == 0 || s.charAt(i - 1) == 'e' || s.charAt(i - 1) == 'E')) {

        } else {
            return false;
        }
    }
    return hasNum;
}
```

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：

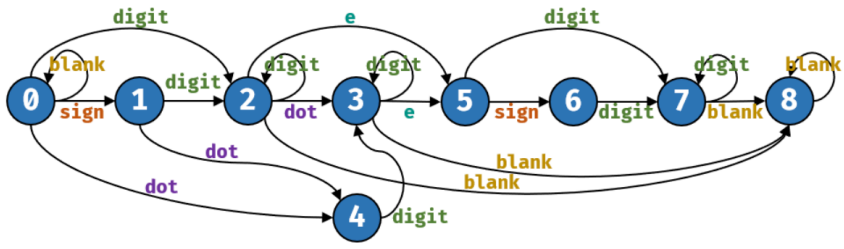
按照字符串从左到右的顺序，定义以下 9 种状态。

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 。

状态转移图



表示	缩写	含义	字符
blank		空格	' '
sign	s	正负号	'+','-'
digit	d	数字	'0'~'9'
dot	.	小数点	'.'
e	e	幂符号	'e','E'

状态	描述
0	起始的 blank
1	e 之前的 sign
2	dot 之前的 digit
3	dot 之后的 digit
4	当 dot 前为空时，dot 后的 digit
5	e
6	e 之后的 sign
7	e 之后的 digit
8	尾部的 blank

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。

```java
class Solution {
    public boolean isNumber(String s) {
        Map[] states = {
            new HashMap<>() {{ put(' ', 0); put('s', 1); put('d', 2); put('.', 4); }}, // 0.
            new HashMap<>() {{ put('d', 2); put('.', 4); }},                           // 1.
            new HashMap<>() {{ put('d', 2); put('.', 3); put('e', 5); put(' ', 8); }}, // 2.
            new HashMap<>() {{ put('d', 3); put('e', 5); put(' ', 8); }},              // 3.
            new HashMap<>() {{ put('d', 3); }},                                        // 4.
            new HashMap<>() {{ put('s', 6); put('d', 7); }},                           // 5.
            new HashMap<>() {{ put('d', 7); }},                                        // 6.
            new HashMap<>() {{ put('d', 7); put(' ', 8); }},                           // 7.
            new HashMap<>() {{ put(' ', 8); }}                                         // 8.
        };
        int p = 0;
        char t;
        for(char c : s.toCharArray()) {
            if(c >= '0' && c <= '9') t = 'd';
            else if(c == '+' || c == '-') t = 's';
            else if(c == 'e' || c == 'E') t = 'e';
            else if(c == '.' || c == ' ') t = c;
            else t = '?';
            if(!states[p].containsKey(t)) return false;
            p = (int)states[p].get(t);
        }
        return p == 2 || p == 3 || p == 7 || p == 8;
    }
}
```

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。

```
示例：

输入：nums = [1,2,3,4]
输出：[1,3,2,4]
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。

```java
/**
 * 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。
 * @author PAN
 * @param nums 待调整的数组
 * @return 按奇偶调整后的数组
 * @time O(N)
 * @space O(N)
 */
public static int[] exchange(int[] nums) {
    int i = 0, j = nums.length - 1;
    int[] exchangeNums = new int[nums.length];
    for (int k = 0; k < nums.length; k++) {
        if (nums[k] % 2 == 0) {
            exchangeNums[j] = nums[k];
            j--;
        }
        else {
            exchangeNums[i] = nums[k];
            i++;
        }
    }
    return exchangeNums;
}
```

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。

```java
/**
 * 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。
```

```java
     * @author PAN
     * @param nums 待调整的数组
     * @return 按奇偶调整后的数组
     * @time O(N)
     * @space O(1)
     */
    public static int[] exchange2(int[] nums) {
        int i = 0, j = nums.length - 1;
        int tmp;
        while (i < j) {
            boolean flagI = (nums[i] % 2 == 0);
            boolean flagJ = (nums[j] % 2 == 1);
            if (flagI && flagJ) {
                tmp = nums[i];
                nums[i] = nums[j];
                nums[j] = tmp;
                i++;
                j--;
            } else {
                if (!flagI) i++;
                if (!flagJ) j--;
            }
        }
        return nums;
    }

    /**
     * 双指针交换2
     * @author 网友
     * @param nums 待调整的数组
     * @return 按奇偶调整后的数组
     * @time O(N)
     * @space O(1)
     */
    public int[] exchange3(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            while (left < right && nums[left] % 2 != 0) {
                left++;
            }
            while (left < right && nums[right] % 2 == 0) {
                right--;
            }
            if (left < right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
            }
```

```
    }
    return nums;
}
```

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。

```
示例：

给定一个链表：1->2->3->4->5, 和 k = 2.

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；

```java
/**
 * 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
 * @author PAN
 * @param head 链表头节点
 * @param k 倒数第 k
 * @return 链表倒数第 k 个节点
 * @time O(N)
 * @space O(1)
 */
public ListNode getKthFromEnd(ListNode head, int k) {
    int len = 0;
    ListNode point = head;
    while (point != null) {
        len++;
        point = point.next;
    }
    if (k > len) return null;
    int i = 0;
    point = head;
    while (i != len - k) {
        point = point.next;
        i++;
    }
    return point;
}
```

2. **双指针 O(N) O(1)** - 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；

```java
/**
 * 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
 * @author 网友 & PAN，参考思路自己实现
 * @param head 链表头节点
 * @param k 倒数第 k
 * @return 链表倒数第 k 个节点
 * @time O(N)
 * @space O(1)
 */
public ListNode getKthFromEnd2(ListNode head, int k) {
    ListNode former = head, latter = head;
    for (int i = 0; i < k; i++) {
        if (latter == null) return null;
        latter = latter.next;
    }
    while (latter != null) {
        former = former.next;
        latter = latter.next;
    }
    return former;
}
```

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。

```
示例：

输入：1->2->3->4->5->NULL
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；

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

```java
public ListNode reverseList(ListNode head) {
    if (head == null) return null;
    Stack s = new Stack();
    while (head != null) {
        s.push(head);
        head = head.next;
    }
    ListNode newHead = (ListNode) s.pop();
    ListNode point = newHead;
    while (!s.isEmpty()) {
        point.next = (ListNode) s.pop();
        point = point.next;
    }
    point.next = null;
    return newHead;
}

/**
 * 数组：用数组按原先顺序存储，再按倒序反向操作
 * @author PAN
 * @param head 链表头节点
 * @return 反转后链表头
 * @time O(N)
 * @space O(N)
 */
public ListNode reverseList2(ListNode head){
    if (head == null) return null;
    ListNode[] array = new ListNode[5000];
    int i = 0, len = 0;
    while (head != null) {
        array[i] = head;
        i++;
        len++;
        head = head.next;
    }
    for (i = len - 1; i > 0; i--) {
        array[i].next = array[i - 1];
    }
    array[0].next = null;
    return array[len - 1];
}
```

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；

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

```
 * @time O(N)
 * @space O(1)
 */
public ListNode reverseList3(ListNode head) {
    if (head == null) return null;
    ListNode former = head, latter = head.next, tmp;
    if (latter == null) return head;
    former.next = null;
    while (latter != null) {
        tmp = latter.next;
        latter.next = former;
        former = latter;
        latter = tmp;
    }
    return former;
}

/**
 * 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, head.next 以省略一些多余的判断
 * @author 网友
 * @param head 链表头节点
 * @return 反转后链表头
 * @time O(N)
 * @space O(1)
 */
public ListNode reverseList4(ListNode head) {
    ListNode cur = head, pre = null;
    while(cur != null) {
        ListNode tmp = cur.next; // 暂存后继节点 cur.next
        cur.next = pre;          // 修改 next 引用指向
        pre = cur;               // pre 暂存 cur
        cur = tmp;               // cur 访问下一节点
    }
    return pre;
}
```

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。

```
/**
 * 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 next 引用指向。
 * @author 网友
 * @param head 链表头节点
 * @return 反转后链表头
 * @time O(N)
 * @space O(N)
 */
```

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