# 剑指offer

## 1-10

### 3、数组中重复的数字

找出数组中重复的数字。

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

**题解**

1、数组排序，一次遍历，遇到与前一个数相等的就输出

```
class Solution:
    def findRepeatNumber(self, nums: List[int]) -> int:
        nums.sort()
        for i in range(1,len(nums)):
            if nums[i-1]==nums[i]:
                return nums[i]
```

2、使用哈希表（Set）记录数组的各个数字，当查找到重复数字则直接返回。

```
class Solution:
    def findRepeatNumber(self, nums: [int]) -> int:
        dic = set()
        for num in nums:
            if num in dic: return num
            dic.add(num)
        return -1
```

3、原地交换

可遍历数组并通过交换操作，使元素的 索引 与 值 一一对应（即 nums[i] = inums[i]=i ）。因而，就能通过索引映射对应的值，起到与字典等价的作用

1. 若 nums[i] = i： 说明此数字已在对应索引位置，无需交换，因此跳过；
2. 若 nums[nums[i]] = nums[i]： 代表索引 nums[i] 处和索引 i 处的元素值都为 nums[i]，即找到一组重复值，返回此值 nums[i]；
3. 否则： 交换索引为 i 和 nums[i] 的元素值，将此数字交换至对应索引位置。

```python
class Solution:
    def findRepeatNumber(self, nums: [int]) -> int:
        i = 0
        while i < len(nums):
            if nums[i] == i:
                i += 1
                continue
            if nums[nums[i]] == nums[i]: return nums[i]
            nums[nums[i]], nums[i] = nums[i], nums[nums[i]]
        return -1
```

## 4、二位数组中的查找

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,   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`

**题解**

1、从左下角或者右上角开始走

```python
class Solution:
    def findNumberIn2DArray(self, matrix: List[List[int]], target: int) -> bool:
        i, j = len(matrix) - 1, 0
        while i >= 0 and j < len(matrix[0]):
            if matrix[i][j] > target: i -= 1
            elif matrix[i][j] < target: j += 1
            else: return True
        return False
```

2、二分搜索

首先在对角线上迭代，对于每一个对角线元素，对该元素所在的行和列使用二分搜索。

```python
class Solution:
    def binary_search(self, matrix, target, start, vertical):
        lo = start
        hi = len(matrix) - 1 if vertical else len(matrix[0]) - 1 # 垂直搜索：hi = 行数 - 1
        while lo <= hi:
```

```
            mid = (lo + hi) // 2
            if vertical:  # 垂直搜索
                if matrix[mid][start] < target:
                    lo = mid + 1
                elif matrix[mid][start] > target:
                    hi = mid - 1
                else:
                    return True
            else:   # 水平搜索
                if matrix[start][mid] < target:
                    lo = mid + 1
                elif matrix[start][mid] > target:
                    hi = mid - 1
                else:
                    return True

        return False

    def findNumberIn2DArray(self, matrix: List[List[int]], target: int) -> bool:
        if not matrix: return False   # 边界条件

        for i in range(min(len(matrix), len(matrix[0]))):
            vertical_found = self.binary_search(matrix, target, i, True) # 垂直方向是否找到
            horizontal_found = self.binary_search(matrix, target, i, False) # 水平是否找到
            if vertical_found or horizontal_found:  # 任一方向找到即可
                return True

        return False
```

3、递归

```
class Solution:
    def findNumberIn2DArray(self, matrix: List[List[int]], target: int) -> bool:
        if not matrix: return False
        def search_backtrack(left, up, right, down):
            if left > right or up > down:
                return False
            elif target < matrix[up][left] or target > matrix[down][right]:
                return False
            mid = left + (right - left) // 2

            row = up        #中间查找分开
            while row <= down and matrix[row][mid] <= target:
                if matrix[row][mid] == target:
                    return True
                row += 1

            return search_backtrack(left, row, mid - 1, down) or search_backtrack(mid + 1, up, right, down)

        return search_backtrack(0, 0, len(matrix[0]) - 1, len(matrix) - 1)
```

## 5、替换空格

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."

**题解**

1、python内置函数——replace

```
class Solution:
    def replaceSpace(self, s: str) -> str:
        return s.replace(" ","%20")
```

2、split之后join

```
class Solution:
    def replaceSpace(self, s: str) -> str:
        return '%20'.join(s.split(' '))
```

3、一次遍历，另存res字符串，遇到空格保存“%20”

```
class Solution:
    def replaceSpace(self, s: str) -> str:
        if not s:return s
        res=""
        for i in s:
            if i!=" ":
                res+=i
            else:
                res+="%20"

        return res
```

## 6、从头到尾打印链表

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]

**题解**

1、遍历，保存到res数组

```python
class Solution:
    def reversePrint(self, head: ListNode) -> List[int]:
        res=[]
        while head:
            res.append(head.val)
            head=head.next

        return res[::-1]
```

2、递归

```python
class Solution:
    def reversePrint(self, head: ListNode) -> List[int]:
        return self.reversePrint(head.next) + [head.val] if head else []
```

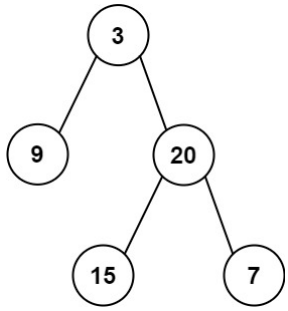
分开写

```python
class Solution:
    def reversePrint(self, head: ListNode) -> List[int]:
        def recur(node):
            if not node:
                return
            recur(node.next)
            res.append(node.val)
        res = []
        recur(head)
        return res
```

## 7、重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请构建该二叉树并返回其根节点。

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。



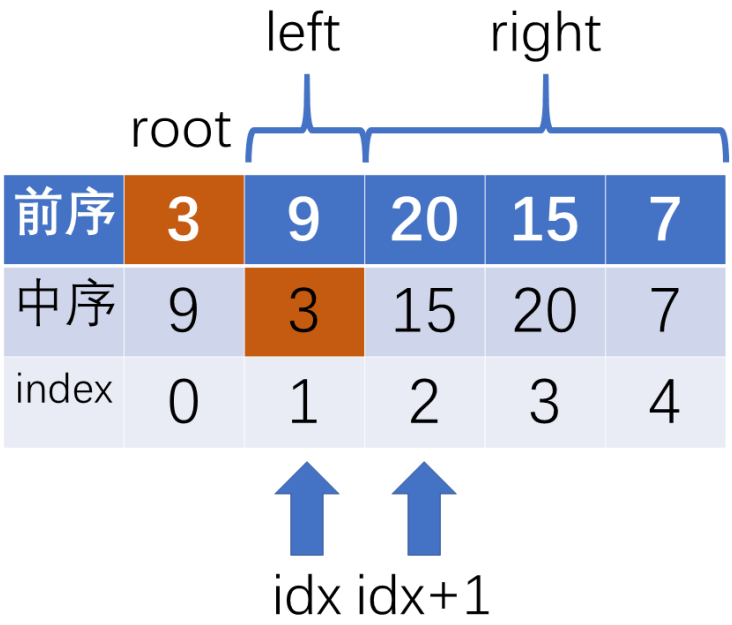

```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```

**题解**

1、先序找根，划分左右，再递归构造左右子树

- 通过先序遍历我们可以找到root，根据root我们可以再中序找到当前root对应的左右子树，再递归对当前root的左右子树进行构造
- 知道inorder中，当前root的左侧的所有点就是其左子树，root的右侧的所有点就是当前root的右子树，就把这左右两堆数字想成当前root的左右2个节点就好，然后扔到函数里进行下一层的递归。
- `inorder.index(preorder[0])` 这一步获取根的索引值，题目说树中的各个节点的值都不相同，也确保了这步得到的结果是唯一准确的。而且这个idx还能当长度用相当于 左+根 的长度，因为 左+根 和 根+左 是等长的。
-




```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) ->
TreeNode:
        if not preorder or not inorder:  # 递归终止条件
            return
        root = TreeNode(preorder[0])
        # 先序为“根左右”，所以根据preorder可以确定root

        idx = inorder.index(preorder[0])
        # 中序为“左根右”，根据root可以划分出左右子树

        # 下面递归对root的左右子树求解即可
        root.left = self.buildTree(preorder[1:1 + idx], inorder[:idx])
        root.right = self.buildTree(preorder[1 + idx:], inorder[idx + 1:])
        return root
```

## 9、用两个栈实现队列

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 )

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

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

---

**题解**

1、辅助栈

```
class CQueue:
    def __init__(self):
        self.A, self.B = [], []

    def appendTail(self, value: int) -> None:
        self.A.append(value)

    def deleteHead(self) -> int:
        if self.B: return self.B.pop()       #A保存数组，B实现A元素倒序
        if not self.A: return -1
        while self.A:
            self.B.append(self.A.pop())
        return self.B.pop()
```

## 10-Ⅰ、斐波那契数列

---

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。

```
输入：n = 2
输出：1
```

```
输入：n = 5
输出：5
```

---

**题解**

1、压缩的动态规划

```python
class Solution:
    def fib(self, n: int) -> int:
        x1,x2=0,1
        if n==0:return 0
        if n==1:return 1
        for i in range(2,n+1):
            x2,x1=x1+x2,x2
        if x2>1000000007:
            x2%=1000000007

        return x2
```

优化

```python
class Solution:
    def fib(self, n: int) -> int:
        a, b = 0, 1
        for _ in range(n):
            a, b = b, a + b
        return a % 1000000007
```

2、递归

```python
class Solution:
    def fib(self, n: int) -> int:
        demo = {}
        def dp(n):
            if n in demo:
                return demo[n]
            if n == 0:
                return 0
            if n == 1:
                return 1
            demo[n] = dp(n-1)+dp(n-2)
            return demo[n]

        return dp(n)%1000000007
```

## 10-II、青蛙跳台阶问题

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。

> 输入：n = 2
> 输出：2

> 输入：n = 7
> 输出：21

```
输入：n = 0
输出：1
```

---

**题解**

1、同上题，只不过初始条件变成1，动态规划，d(n)表示n阶台阶共有多少中跳法，d(n)=d(n-1)+d(n-2)

```python
class Solution:
    def numWays(self, n: int) -> int:
        a,b=1,1
        for _ in range(n):
            a,b=b,a+b

        return a%1000000007
```

# 11-20

## 11、旋转数组的最小数字

---

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。

```
输入：[3,4,5,1,2]
输出：1
```

```
输入：[2,2,2,0,1]
输出：0
```

---

**题解**

1、数组排序，输出第一个数

```python
class Solution:
    def minArray(self, numbers: List[int]) -> int:
        numbers.sort()
        return numbers[0]
```

2、一次遍历，遇到第一个比前一个数小的，就输出，没有就输出第一个数

```python
class Solution:
    def minArray(self, numbers: List[int]) -> int:
        for i in range(1,len(numbers)):
            if numbers[i-1]>numbers[i]:
                return numbers[i]

        return numbers[0]
```

3、二分法

1. 初始化： 声明 i, j 双指针分别指向 nums数组左右两端；
2. 循环二分： 设 m = (i + j) / 2 为每次二分的中点（ "/" 代表向下取整除法，因此恒有 i≤m<j ），可分为以下三种情况：
    1. 当 nums[m] > nums[j] 时： m 一定在 左排序数组 中，即旋转点 x 一定在 [m + 1, j] 闭区间内，因此执行 i = m + 1；
    2. 当 nums[m] < nums[j] 时： m 一定在 右排序数组 中，即旋转点 xx 一定在[i, m]闭区间内，因此执行 j = m；
    3. 当 nums[m] = nums[j] 时： 无法判断 m 在哪个排序数组中，即无法判断旋转点 x 在 [i, m]还是 [m + 1, j]区间中。解决方案： 执行 j = j - 1缩小判断范围，分析见下文。
3. 返回值： 当 i = j 时跳出二分循环，并返回 旋转点的值 nums[i] 即可。

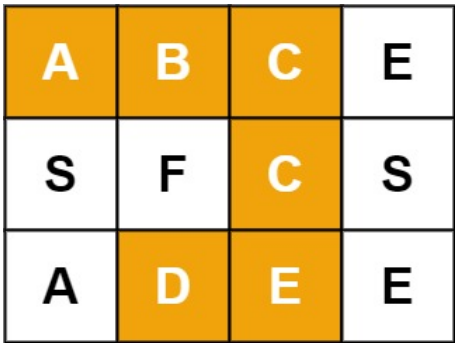
```
class Solution:
    def minArray(self, numbers: [int]) -> int:
        i, j = 0, len(numbers) - 1
        while i < j:
            m = (i + j) // 2
            if numbers[m] > numbers[j]: i = m + 1
            elif numbers[m] < numbers[j]: j = m
            else: j -= 1
        return numbers[i]
```

## 12、矩阵中的路径

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

例如，在下面的 3×4 的矩阵中包含单词 "ABCCED"（单词中的字母已标出）。




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

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

**题解**

1、DFS+剪枝

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(i, j, k):
            if not 0 <= i < len(board) or not 0 <= j < len(board[0]) or board[i]
[j] != word[k]: return False
            if k == len(word) - 1: return True
            board[i][j] = ''         #board[i][j]置空，表示已经访问过该字符
            res = dfs(i + 1, j, k + 1) or dfs(i - 1, j, k + 1) or dfs(i, j + 1,
k + 1) or dfs(i, j - 1, k + 1)
            board[i][j] = word[k]
            return res

        for i in range(len(board)):
            for j in range(len(board[0])):
                if dfs(i, j, 0): return True
        return False
```

## 13、机器人的运动范围

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？

> 输入：m = 2, n = 3, k = 1
> 输出：3

> 输入：m = 3, n = 1, k = 0
> 输出：1

**题解**

1、DFS，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。

1. 递归参数： 当前元素在矩阵中的行列索引 i 和 j ，两者的数位和 si, sj 。
2. 终止条件： 当 ① 行列索引越界 或 ② 数位和超出目标值 k 或 ③ 当前元素已访问过 时，返回 0 ，代表不计入可达解。
3. 递推工作：
    1. 标记当前单元格 ： 将索引 (i, j) 存入 Set visited 中，代表此单元格已被访问过。
    2. 搜索下一单元格： 计算当前元素的 下、右 两个方向元素的数位和，并开启下层递归 。
4. 回溯返回值： 返回 1 + 右方搜索的可达解总数 + 下方搜索的可达解总数，代表从本单元格递归搜索的可达解总数。

```python
class Solution:
    def movingCount(self, m: int, n: int, k: int) -> int:
        def dfs(i, j, si, sj):
            if i >= m or j >= n or k < si + sj or (i, j) in visited: return 0
            visited.add((i,j))
            return 1 + dfs(i + 1, j, si + 1 if (i + 1) % 10 else si - 8, sj) +
dfs(i, j + 1, si, sj + 1 if (j + 1) % 10 else sj - 8)

        visited = set()
        return dfs(0, 0, 0, 0)
```

2、BFS，DFS 是朝一个方向走到底，再回退，以此类推；BFS 则是按照“平推”的方式向前搜索。

1. 初始化： 将机器人初始点 (0, 0) 加入队列 queue ；
2. 迭代终止条件： queue 为空。代表已遍历完所有可达解。
3. 迭代工作：
    1. 单元格出队： 将队首单元格的 索引、数位和 弹出，作为当前搜索单元格。
    2. 判断是否跳过： 若 ① 行列索引越界 或 ② 数位和超出目标值 k 或 ③ 当前元素已访问过 时，执行 continue 。
    3. 标记当前单元格 ： 将单元格索引 (i, j) 存入 Set visited 中，代表此单元格 已被访问过 。
    4. 单元格入队： 将当前元素的 下方、右方 单元格的 索引、数位和 加入 queue 。
4. 返回值： Set visited 的长度 len(visited) ，即可达解的数量。

```python
class Solution:
    def movingCount(self, m: int, n: int, k: int) -> int:
        queue, visited = [(0, 0, 0, 0)], set()
        while queue:
            i, j, si, sj = queue.pop(0)
            if i >= m or j >= n or k < si + sj or (i, j) in visited: continue
            visited.add((i,j))
            queue.append((i + 1, j, si + 1 if (i + 1) % 10 else si - 8, sj))
            queue.append((i, j + 1, si, sj + 1 if (j + 1) % 10 else sj - 8))
        return len(visited)
```

## 14-Ⅰ、剪绳子

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。

> 输入: 2
> 输出: 1
> 解释: 2 = 1 + 1, 1 × 1 = 1

> 输入: 10
> 输出: 36
> 解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36

**题解**

1、数学推导，把绳子分成尽可能多的3的段，乘积最大

- 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 2×2>3×1。

```
class Solution:
    def cuttingRope(self, n: int) -> int:
        if n <= 3: return n - 1
        a, b = n // 3, n % 3
        if b == 0: return int(math.pow(3, a))
        if b == 1: return int(math.pow(3, a - 1) * 4)
        return int(math.pow(3, a) * 2)
```

2、动态规划

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

```
class Solution:
    def cuttingRope(self, n: int) -> int:
        dp = [0] * (n + 1)
        dp[2] = 1
        for i in range(3, n + 1):
            for j in range(2, i):
                dp[i] = max(dp[i], max(j * (i - j), j * dp[i - j]))
        return dp[n]
```

## 14-II、剪绳子II

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。

> 输入: 2
> 输出: 1
> 解释: 2 = 1 + 1, 1 × 1 = 1

```
输入：10
输出：36
解释：10 = 3 + 3 + 4, 3 × 3 × 4 = 36
```

---

**题解**

1、贪心解法

- 为什么没用`math.pow`，大数情况下 `math.pow()` 浮点取幂不精确~
- 可以使用 `3**a` 代替 `math.pow(3,a)`

```python
class Solution:
    def cuttingRope(self, n: int) -> int:
        if n < 4:
            return n - 1
        res = 1
        while n > 4:
            res = res * 3
            n -= 3
        return res * n % 1000000007
```

2、动态规划，最后加上 %1000000007

```python
class Solution:
    def cuttingRope(self, n: int) -> int:
        dp = [0] * (n + 1)
        dp[2] = 1
        for i in range(3, n + 1):
            for j in range(2, i):
                dp[i] = max(dp[i], max(j * (i - j), j * dp[i - j]))
        return dp[n] % 1000000007
```

## 15、二进制中1的个数

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为 汉明重量).）。

```
输入：n = 11 (控制台输入 00000000000000000000000000001011)
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

```
输入：n = 128 (控制台输入 00000000000000000000000010000000)
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。
```

---

**题解**

1、转成二进制，遍历一次计算1的个数

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        res=0
        for i in str(bin(n)):
            if i =="1":
                res+=1

        return res
```

2、循环，向右移，比较最后一位是否为1

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        res=0
        for i in range(32):
            res+=n&1
            n>>=1

        return res
```

3、库函数

```
class Solution(object):
    def hammingWeight(self, n):
        return bin(n).count("1")
```

4、 `n & (n - 1)` ，这个代码可以把 n 的二进制中，最后一个出现的 1 改写成 0。

```
class Solution(object):
    def hammingWeight(self, n):
        res = 0
        while n:
            res += 1
            n &= n - 1
        return res
```

## 16、数值的整数次方

实现 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。

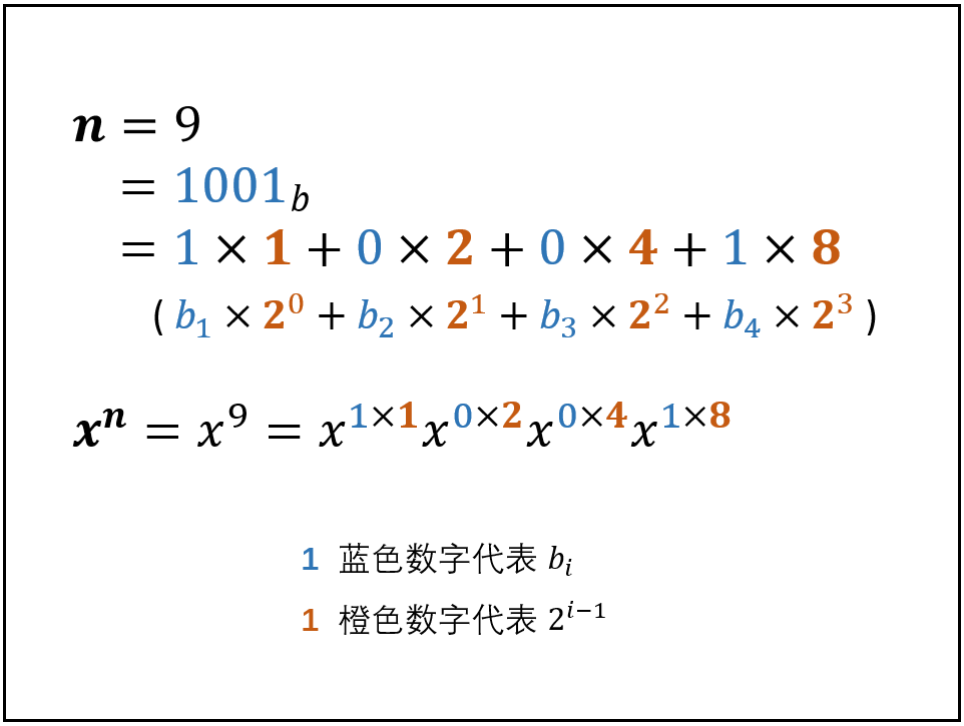
> 输入：x = 2.00000, n = 10
> 输出：1024.00000
>
> 输入：x = 2.10000, n = 3
> 输出：9.26100
>
> 输入：x = 2.00000, n = -2
> 输出：0.25000
> 解释：2-2 = 1/22 = 1/4 = 0.25

**题解**

1、Python自带写法

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        return x**n
```

2、快速幂




1. 当 x = 0 时：直接返回 0 （避免后续 x = 1 / x 操作报错）。
2. 初始化 res = 1；
3. 当 n < 0时：把问题转化至 n≥0 的范围内，即执行 x=1/x ， n=−n ；
4. 循环计算：当 n = 0时跳出；
    1. 当 n & 1 = 1时：将当前 x 乘入 res （即 res *= x ）；
    2. 执行 x = x^2 （即 x *= xx∗=x ）；
    3. 执行 n右移一位（即 n >>= 1）。
5. 返回 res 。

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if x == 0: return 0
        res = 1
        if n < 0: x, n = 1 / x, -n
        while n:
            if n & 1: res *= x
            x *= x
            n >>= 1
        return res
```

3、递归

1. 如果n == 0，返回1
2. 如果n < 0，最终结果为 1/x^−n

3. 如果n为奇数，最终结果为x*x ^n−1
4. 如果n为偶数，最终结果为 x ^2*(n/2)

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        elif n < 0:
            return 1/self.myPow(x, -n)
        elif n & 1:
            return x * self.myPow(x, n - 1)
        else:
            return self.myPow(x*x, n // 2)
```

## 17、打印从1到最大的n位数

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。

> 输入: n = 1
> 输出: [1,2,3,4,5,6,7,8,9]

---

**题解**

1、遍历写入数据

```
class Solution:
    def printNumbers(self, n: int) -> List[int]:
        res=[]
        dic={}
        for i in range(1,10):
            dic[i]=10**i-1
        for i in range(dic[n]):
            res.append(i+1)
        return res
```

优化

```
class Solution:
    def printNumbers(self, n: int) -> List[int]:
        return list(range(1, 10 ** n))
```

2、大数打印解法

```
class Solution:
    def printNumbers(self, n: int) -> [int]:
        def dfs(x):
            if x == n:  # 终止条件：已固定完所有位
                s = ''.join(num[self.start:])
                if s != '0': res.append(int(s))     # 拼接 num 并添加至 res 尾部
                if n - self.start == self.nine: self.start -= 1
                return
```

```
            for i in range(10):     # 遍历 0 - 9
                if i == 9: self.nine += 1
                num[x] = str(i)     # 固定第 x 位为 i
                dfs(x + 1)      # 开启固定第 x + 1 位
            self.nine -= 1

        num, res = ['0'] * n, []    # 起始数字定义为 n 个 0 组成的字符列表
        self.nine = 0
        self.start = n - 1
        dfs(0)      # 开启全排列递归
        return res
```

多个for循环，迭代写法

```
class Solution:
    def printNumbers(self, n: int) -> List[int]:
        nums = []
        for i in range(n):
            curnums = []
            for j in range(1, 10):
                curnums.append(str(j) + '0' * i)
                for num in nums:
                    curnums.append(str(j) + '0' * (i - len(num)) + num)
            nums.extend(curnums)
        return list(map(int, nums))
```

## 18、删除链表中的节点

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。

> ```
> 输入: head = [4,5,1,9], val = 5
> 输出: [4,1,9]
> 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.
> ```
>
> ```
> 输入: head = [4,5,1,9], val = 1
> 输出: [4,5,9]
> 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.
> ```

**题解**

1、判断下一个节点是否等于val，等于的话就跳过这个节点

```python
class Solution:
    def deleteNode(self, head: ListNode, val: int) -> ListNode:
        if head.val==val:return head.next
        cur=head
        while cur:
            if cur.next.val==val:
                cur.next=cur.next.next
                break
            else:
                cur=cur.next
        return head
```

2、递归解法

```python
class Solution:
    def deleteNode(self, head: ListNode, val: int) -> ListNode:
        if not head:
            return head
        if head.val == val:
            return head.next
        head.next = self.deleteNode(head.next, val)
        return head

//  用三目运算符
class Solution:
    def deleteNode(self, head: ListNode, val: int) -> ListNode:
        if not head:
            return head
        head.next = self.deleteNode(head.next, val)

        return head.next if head.val == val else head
```

## 19、正则表达式匹配

给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。

- `'.'` 匹配任意单个字符
- `'*'` 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 **整个** 字符串 `s`的，而不是部分字符串。

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

> 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。

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

```
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"。
```

---

**题解**

1、回溯

- 首先，我们考虑只有 `'.'` 的情况。这种情况会很简单：我们只需要从左到右依次判断 `s[i]` 和 `p[i]` 是否匹配。

```python
def isMatch(self,s:str, p:str) -> bool:
    if not p: return not s # 边界条件

    first_match = s and p[0] in {s[0],'.'} # 比较第一个字符是否匹配

    return first_match and self.isMatch(s[1:], p[1:])
```

- 如果有星号，它会出现在 p[1] 的位置，这时有两种情况：
  - 星号代表匹配 0 个前面的元素。如 `'##'` 和 `a*##`，这时我们直接忽略 p 的 a*，比较 ## 和 ##；
  - 星号代表匹配一个或多个前面的元素。如 `aaab` 和 `a*b`，这时我们将忽略 s 的第一个元素，比较 `aab` 和 `a*b`。
- 以上任一情况忽略掉元素进行比较时，剩下的如果匹配，我们认为 s 和 p 是匹配的。

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        if not p: return not s
        # 第一个字母是否匹配
        first_match = bool(s and p[0] in {s[0],'.'})
        # 如果 p 第二个字母是 *
        if len(p) >= 2 and p[1] == "*":
            return self.isMatch(s, p[2:]) or \
            first_match and self.isMatch(s[1:], p)
            #多个and和or，先运算所有and部分，在运算所有or
        else:
            return first_match and self.isMatch(s[1:], p[1:])
```

2、动态规划

`dp[i][j]` 表示的状态是 s 的前 i 项和 p 的前 j 项是否匹配。

现在如果已知了 `dp[i-1][j-1]` 的状态，我们该如何确定 `dp[i][j]` 的状态呢？我们可以分三种情况讨论，其中，前两种情况考虑了所有能匹配的情况，剩下的就是不能匹配的情况了：

1. `s[i] == p[j] or p[j] == '.'`：比如 abb 和 abb，或者 abb 和 ab.，很容易得到 `dp[i][j] = dp[i-1][j-1] = True`。因为 ab 和 ab 是匹配的，如果后面分别加一个 b，或者 s 加一个 b 而 p 加一个 .，仍然是匹配的。

2. p[j] == '*'：当 p[j] 为星号时，由于星号与前面的字符相关，因此我们比较星号前面的字符 p[j-1] 和 s[i] 的关系。根据星号前面的字符与 s[i] 是否相等，又可分为以下两种情况：
    - p[j-1] != s[i]：如果星号前一个字符匹配不上，星号匹配了 0 次，应忽略这两个字符，看 p[j-2] 和 s[i] 是否匹配。 这时 `dp[i][j] = dp[i][j-2]`。
    - p[j-1] == s[i] or p[j-1] == '.':星号前面的字符可以与 s[i] 匹配，这种情况下，星号可能匹配了前面的字符的 0 个，也可能匹配了前面字符的多个，当匹配 0 个时，如 ab 和 abb，*或者 ab 和 ab.*，这时我们需要去掉 p 中的 b* 或 .* 后进行比较，即 `dp[i][j] = dp[i][j-2]`；当匹配多个时，如 abbb 和 ab*，或者 abbb 和 `a.*`，我们需要将 s[i] 前面的与 p 重新比较，即 `dp[i][j] = dp[i-1][j]`
3. 其他情况：以上两种情况把能匹配的都考虑全面了，所以其他情况为不匹配，即 `dp[i][j] = False`

$$dp(i)(j) = \begin{cases} dp(i-1)(j-1), & \text{s(i)} = \text{p(j)} \text{ or } \text{p(j)} = . \\ dp(i)(j-2), & \text{p(j)} = *, \text{p(j-1)} != \text{s(i)} \\ dp(i-1)(j) or dp(i)(j-2), & \text{p(j)} = *, \text{p(j-1)} = \text{s(i)} \text{ or } \text{p(j-1)} = . \\ False & else \end{cases}$$

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        # 边界条件，考虑 s 或 p 分别为空的情况
        if not p: return not s
        if not s and len(p) == 1: return False

        m, n = len(s) + 1, len(p) + 1
        dp = [[False for _ in range(n)] for _ in range(m)]
        # 初始状态
        dp[0][0] = True
        dp[0][1] = False

        for c in range(2, n):
            j = c - 1
            if p[j] == '*':
                dp[0][c] = dp[0][c - 2]

        for r in range(1,m):
            i = r - 1
            for c in range(1, n):
                j = c - 1
                if s[i] == p[j] or p[j] == '.':
                    dp[r][c] = dp[r - 1][c - 1]
                elif p[j] == '*':       # ‘*’前面的字符匹配s[i] 或者为'.'
                    if p[j - 1] == s[i] or p[j - 1] == '.':
                        dp[r][c] = dp[r - 1][c] or dp[r][c - 2]
                    else:                       # ‘*’匹配了0次前面的字符
                        dp[r][c] = dp[r][c - 2]
                else:
                    dp[r][c] = False
        return dp[m - 1][n - 1]
```

# 20、表示数值的字符串

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）。

数值（按顺序）可以分成以下几个部分：

1. 若干空格
2. 一个 小数 或者 整数
3. （可选）一个 'e' 或 'E' ，后面跟着一个 整数
4. 若干空格

小数（按顺序）可以分成以下几个部分：

1. （可选）一个符号字符（'+' 或 '-'）
2. 下述格式之一：
    1. 至少一位数字，后面跟着一个点 '.'
    2. 至少一位数字，后面跟着一个点 '.' ，后面再跟着至少一位数字
    3. 一个点 '.' ，后面跟着至少一位数字

整数（按顺序）可以分成以下几个部分：

- （可选）一个符号字符（'+' 或 '-'）
- 至少一位数字

部分数值列举如下：

- ["+100", "5e2", "-123", "3.1416", "-1E-16", "0123"]

部分非数值列举如下：

- ["12e", "1a3.14", "1.2.3", "+-5", "12e+5.4"]

> ```
> 输入：s = "0"
> 输出：true
> ```
>
> ```
> 输入：s = "    .1  "
> 输出：true
> ```

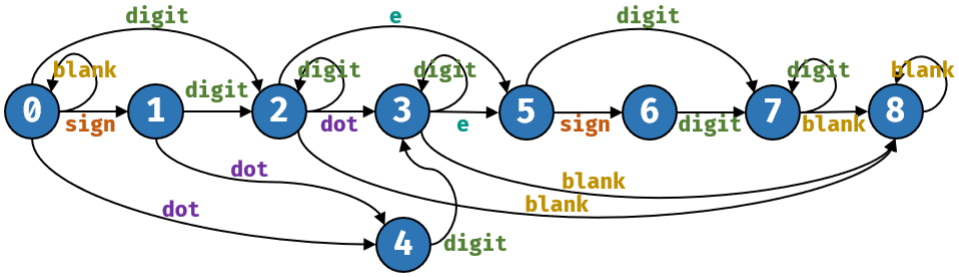
---

**题解**

1、有限状态自动机

- 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
- 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 |

- 状态转移循环： 遍历字符串 ss 的每个字符 cc 。
  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 tt 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]` 。

```
class Solution:
    def isNumber(self, s: str) -> bool:
        states = [
            { ' ': 0, 's': 1, 'd': 2, '.': 4 }, # 0. start with 'blank'
            { 'd': 2, '.': 4 } ,                # 1. 'sign' before 'e'
            { 'd': 2, '.': 3, 'e': 5, ' ': 8 }, # 2. 'digit' before 'dot'
            { 'd': 3, 'e': 5, ' ': 8 },         # 3. 'digit' after 'dot'
            { 'd': 3 },                         # 4. 'digit' after 'dot'
('blank' before 'dot')
            { 's': 6, 'd': 7 },                 # 5. 'e'
            { 'd': 7 },                         # 6. 'sign' after 'e'
            { 'd': 7, ' ': 8 },                 # 7. 'digit' after 'e'
            { ' ': 8 }                          # 8. end with 'blank'
        ]
        p = 0                           # start with state 0
        for c in s:
            if '0' <= c <= '9': t = 'd' # digit
            elif c in "+-": t = 's'     # sign
            elif c in "eE": t = 'e'     # e or E
            elif c in ". ": t = c       # dot, blank
            else: t = '?'               # unknown
```

```
            if t not in states[p]: return False
            p = states[p][t]
        return p in (2, 3, 7, 8)
```

2、循环遍历

1. 首先定义了四个flag，对应四种字符
    - 是否有数字：hasNum
    - 是否有e：hasE
    - 是否有正负符号：hasSign
    - 是否有点：hasDot
2. 其余还定义了字符串长度n以及字符串索引index
3. 先处理一下开头的空格，index相应的后移
4. 然后进入循环，遍历字符串
    - 如果当前字符c是数字：将hasNum置为true，index往后移动一直到非数字或遍历到末尾位置；如果已遍历到末尾(index == n)，结束循环
    - 如果当前字符c是'e'或'E'：如果e已经出现或者当前e之前没有出现过数字，返回fasle；否则令hasE = true，并且将其他3个flag全部置为false，因为要开始遍历e后面的新数字了
    - 如果当前字符c是+或-：如果已经出现过+或-或者已经出现过数字或者已经出现过'.'，返回flase；否则令hasSign = true
    - 如果当前字符c是'.'：如果已经出现过'.'或者已经出现过'e'或'E'，返回false；否则令hasDot = true
    - 如果当前字符c是' '：结束循环，因为可能是末尾的空格了，但也有可能是字符串中间的空格，在循环外继续处理
    - 如果当前字符c是除了上面5种情况以外的其他字符，直接返回false
5. 处理空格，index相应的后移
6. 如果当前索引index与字符串长度相等，说明遍历到了末尾，但是还要满足hasNum为true才可以最终返回true，因为如果字符串里全是符号没有数字的话是不行的，而且e后面没有数字也是不行的，但是没有符号是可以的，所以4个flag里只要判断一下hasNum就行；所以最后返回的是hasNum && index == n
7. 如果字符串中间有空格，按以上思路是无法遍历到末尾的，index不会与n相等，返回的就是false

```
class Solution:
    def isNumber(self, s: str) -> bool:
        n = len(s)
        index = 0
        has_num = has_e = has_sign = has_dot = False
        while index < n and s[index] == ' ':
            index += 1
        while index < n:
            while index < n and '0' <= s[index] <= '9':
                index += 1
                has_num = True
            if index == n:
                break
            if s[index] == 'e' or s[index] == 'E':
                if has_e or not has_num:
                    return False
                has_e = True
                has_num = has_sign = has_dot = False
            elif s[index] == '+' or s[index] == '-':
                if has_sign or has_num or has_dot:
```

```
                    return False
                has_sign = True
            elif s[index] == '.':
                if has_dot or has_e:
                    return False
                has_dot = True
            elif s[index] == ' ':
                break
            else:
                return False
            index += 1
        while index < n and s[index] == ' ':
            index += 1
        return has_num and index == n
```

3、逻辑判断

使用3个标志位met_dot, met_e, met_digit来分别标记是否遇到了“.”,“e/E”和任何0-9的数字。

```
class Solution:
    def isNumber(self, s: str) -> bool:
        s = s.strip()
        met_dot = met_e = met_digit = False
        for i, char in enumerate(s):
            if char in ('+', '-'):
                if i > 0 and s[i-1] != 'e' and s[i-1] != 'E':
                    return False
            elif char == '.':
                if met_dot or met_e:
                    return False
                met_dot = True
            elif char == 'e' or char == 'E':
                if met_e or not met_digit:
                    return False
                met_e, met_digit = True, False # e后必须接，所以这时重置met_digit为
False,以免e为最后一个char
            elif char.isdigit():
                met_digit = True
            else:
                return False
        return met_digit
```

4、正则表达式

```
import re
class Solution:
    p = re.compile(r'^[+-]?(\.\d+|\d+\.?\d*)([eE][+-]?\d+)?$')
    def isNumber(self, s: str) -> bool:
        return bool(self.p.match(s.strip()))
```

5、try/except法

```
class Solution(object):
    def isNumber(self, s):
        try:
            float(s)
            return True
        except :
            return False
```

# 21-30

## 21、调整数组顺序使奇数位于偶数前面

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。

> 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。

**题解**

1、双指针，前后指针

```
class Solution:
    def exchange(self, nums: List[int]) -> List[int]:
        left,right=0,len(nums)-1
        while left<right:
            while nums[left]%2==1 and left<len(nums)-1: #找到从左往右的偶数
                left+=1
            while nums[right]%2==0 and right>0:         #找到从右往左的奇数
                right-=1
            if left<right:                              #如果偶数在奇数左边，交换
                nums[left],nums[right]=nums[right],nums[left]
                left+=1
                right-=1
            else:                                       #否则直接退出循环
                break

        return nums
```

2、辅助数组

```python
class Solution:
    def exchange(self, nums: List[int]) -> List[int]:
        odd = []
        even = []

        for num in nums:
            if num % 2 == 1:
                odd.append(num)
            else:
                even.append(num)
        return odd + even
```

3、快慢指针

```python
class Solution:
    def exchange(self, nums: List[int]) -> List[int]:
        slow = fast = 0
        while fast < len(nums):
            if nums[fast] % 2 == 1:
                nums[slow], nums[fast] = nums[fast], nums[slow]
                slow += 1
            fast += 1
        return nums
```

## 22、链表中倒数第K个节点

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。

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

**题解**

1、快慢指针

- 先让快指针走k步
- 然后慢指针和快指针一起走到头

```python
class Solution:
    def getKthFromEnd(self, head: ListNode, k: int) -> ListNode:
        slow,fist=head,head
        for _ in range(k):
            fist=fist.next
        while fist:
            slow=slow.next
            fist=fist.next

        return slow
```

2、使用数组，保存每个节点

```python
class Solution:
    def getKthFromEnd(self, head: ListNode, k: int) -> ListNode:
        res = []
        while head:
            res.append(head)        #数组保存的就是每个节点，链表类
            head = head.next
        return res[-k]
```

## 24、反转链表

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。

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

**题解**

1、链表转数组，数组转链表

```python
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        res=[]
        while head:
            res.append(head.val)
            head=head.next

        reve=ListNode(None)
        cur=reve
        for i in res[::-1]:
            cur.next=ListNode(i)
            cur=cur.next

        return reve.next
```

2、迭代，通过三个指针

```python
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre, cur = None, head
        while cur:
            nxt = cur.next
            cur.next = pre
            pre = cur
            cur = nxt
        return pre
```

3、递归

1. 终止条件：当 cur 为空，则返回尾节点 pre （即反转链表的头节点）；
2. 递归后继节点，记录返回值（即反转链表的头节点）为 res；
3. 修改当前节点 cur 引用指向前驱节点 pre；
4. 返回反转链表的头节点 res；

```python
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        def recur(cur, pre):
            if not cur: return pre     # 终止条件
            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

        return recur(head, None)       # 调用递归并返回
```

## 25、合并两个排序的链表

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。

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

**题解**

1、递归

- 终止条件：当两个链表都为空时，表示我们对链表已合并完成。
- 如何递归：我们判断 l1 和 l2 头结点哪个更小，然后较小结点的 next 指针指向其余结点的合并结果。（调用递归）

```python
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        if not l1: return l2  # 终止条件，直到两个链表都空
        if not l2: return l1
        if l1.val <= l2.val:  # 递归调用
            l1.next = self.mergeTwoLists(l1.next,l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1,l2.next)
            return l2
```

2、提前截止运算

- and：如果 and 前面的表达式已经为 False，那么 and 之后的表达式将被 跳过，返回左表达式结果
- or：如果 or 前面的表达式已经为 True，那么 or 之后的表达式将被跳过，直接返回左表达式的结果
- 代码流程
    - 判断 l1 或 l2 中是否有一个节点为空，如果存在，那么我们只需要把不为空的节点接到链表后面即可
    - 对 l1 和 l2 重新赋值，使得 l1 指向比较小的那个节点对象
    - 修改 l1 的 next 属性为递归函数返回值
    - 返回 l1，注意：如果 l1 和 l2 同时为 None，此时递归停止返回 None

```python
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        if l1 and l2:
            if l1.val > l2.val: l1, l2 = l2, l1
            l1.next = self.mergeTwoLists(l1.next, l2)
        return l1 or l2
```

3、迭代

```python
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode(0)
        cur = dummy
        while l1 and l2:
            if l1.val <= l2.val:
                cur.next = l1
                l1 = l1.next
            else:
                cur.next = l2
                l2 = l2.next
            cur = cur.next

        cur.next = l1 if l1 else l2
        return dummy.next
```

# 26、树的子结构

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。

> 给定的树 A:
>
> ```
>    3
>   / \
>  4   5
> / \
> 1   2
> ```
>
> 给定的树 B：
>
> ```
> 4
> /
> 1
> ```
>
> 返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。
>
> ```
> 输入：A = [1,2,3], B = [3,1]
> 输出：false
> ```

---

**题解**

1、先序遍历

- recur(A, B) 函数：
    1. 终止条件：
        1. 当节点 B 为空：说明树 B 已匹配完成（越过叶子节点），因此返回 true；
        2. 当节点 A 为空：说明已经越过树 A 叶子节点，即匹配失败，返回 false；
        3. 当节点 A 和 B 的值不同：说明匹配失败，返回 false；
    2. 返回值：
        1. 判断 A 和 B 的左子节点是否相等，即 recur(A.left, B.left)；
        2. 判断 A 和 B 的右子节点是否相等，即 recur(A.right, B.right)；
- isSubStructure(A, B) 函数：
    1. 特例处理： 当 树 A 为空 或 树 B 为空 时，直接返回 false；
    2. 返回值： 若树 B 是树 A 的子结构，则必满足以下三种情况之一，因此用或 || 连接；
        1. 以 节点 A 为根节点的子树 包含树 B ，对应 recur(A, B)；
        2. 树 B 是 树 A 左子树 的子结构，对应 isSubStructure(A.left, B)；
        3. 树 B 是 树 A 右子树 的子结构，对应 isSubStructure(A.right, B)；

```
class Solution:
    def isSubStructure(self, A: TreeNode, B: TreeNode) -> bool:
        def recur(A, B):
            if not B: return True
            if not A or A.val != B.val: return False
            return recur(A.left, B.left) and recur(A.right, B.right)

        return bool(A and B) and (recur(A, B) or self.isSubStructure(A.left, B)
or self.isSubStructure(A.right, B))
```

2、迭代（BFS）

- 思路
  - 先遍历树A，如果遍历到和B节点值相同的节点，进入helper方法判断接下来的节点是否都相同
  - 节点都相同返回True；不相同返回False，并且继续遍历树A找下一个相同的节点
  - 如果遍历完了A还没有返回过True，说明B不是A的子结构，返回False
- **helper方法**：用于判断从A的子树是否有和B相同的部分
  - 正常BFS步骤，用队列存储树A和B相对应的节点nodeA, nodeB
  - 因为入队的条件是只要树B节点存在就入队，如果A已经没有了相应节点返回False：if not A
  - 如果A和B对应节点值不相同也返回False：if nodeA.val != nodeB.val
  - 如果遍历完了B也没有返回过False，说明B是A的子结构，返回True

```
class Solution:
    def isSubStructure(self, A: TreeNode, B: TreeNode) -> bool:
        def helper(A, B):
            queue = [(A, B)]
            while queue:
                nodeA, nodeB = queue.pop(0)
                if not nodeA or nodeA.val != nodeB.val:
                    return False
                if nodeB.left:
                    queue.append((nodeA.left, nodeB.left))
                if nodeB.right:
                    queue.append((nodeA.right, nodeB.right))
            return True

        if not B: return False
        queue = collections.deque([A])
        while queue:
            node = queue.popleft()
            if node.val == B.val:
                if helper(node, B):
                    return True
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        return False
```

## 27、二叉树的镜像

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

> 输入
>
>      4
> /   \
> 2     7
> / \   / \
> 1   3 6   9
> 镜像输出：
>   4
> /   \
> 7     2
> / \   / \
> 9   6 3   1
> 输入：root = [4,2,7,1,3,6,9]
> 输出：[4,7,2,9,6,3,1]

**题解**

1、递归

```
class Solution:
    def mirrorTree(self, root: TreeNode) -> TreeNode:
        def dfs(root):
            if not root:return
            root.left,root.right=dfs(root.right),dfs(root.left)
            return root

        return dfs(root)
```

2、迭代

```
class Solution:
    def mirrorTree(self, root: TreeNode) -> TreeNode:
        if not root: return
        stack = [root]
        while stack:
            node = stack.pop()
            if node.left: stack.append(node.left)
            if node.right: stack.append(node.right)
            node.left, node.right = node.right, node.left
        return root
```

## 28、对称的二叉树

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。

```
输入：root = [1,2,2,3,4,4,3]
输出：true
```

```
输入：root = [1,2,2,null,3,null,3]
输出：false
```

---

**题解**

1、递归：根据镜像二叉树的特点，即左右子树是对称的

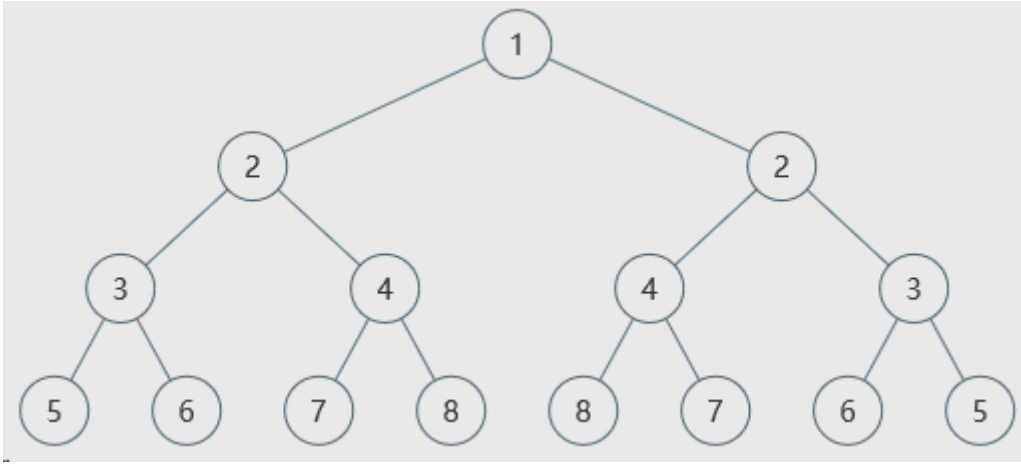
```
class Solution(object):
    def isSymmetric(self, root):
        if not root:
            return True

        def isSym(root1, root2):
            if not root1 and not root2:
                return True
            if root1 and root2 and root1.val == root2.val:
                    return isSym(root1.left, root2.right) and isSym(root1.right,
root2.left)
            return False

        return isSym(root.left, root.right)
```

2、迭代

- 在队列中同时取出两个节点left, right，判断这两个节点的值是否相等，然后把他们的孩子中按照(left.left, right.right) 一组，(left.right, right.left)一组放入队列中。
- BFS做法需要把所有的节点都检查完才能确定返回结果True，除非提前遇到不同的节点值而终止返回False。

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if not root:return True
        stack=[(root.left,root.right)]
        while stack:
            left,right=stack.pop()
            if not left and not right: continue
            if not left or not right:return False
            if left.val!=right.val:return False
            stack.append((left.right,right.left))
            stack.append((left.left,right.right))

        return True
```

3、前序遍历+后序遍历




- 首先我们对这棵树根节点的左子树进行前序遍历: pre_order = [2,3,5,6,4,7,8]
- 接着我们对这棵树根节点的右子树进行后序遍历：post_order = [8,7,4,6,5,3,2]
- 根据两次遍历我们不难发现 post_order 就是 pre_order的逆序，其实这也是对称二叉树的一个性质，

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        bli = []      # 用来存左子树的前序遍历
        fli = []      # 用来存右子树的后序遍历
        if root == None:    # 无根节点
            return True
        if root and root.left == None and root.right == None:  # 只有根节点
            return True

        if root and root.left and root.right:
            self.pre_order(root.left, bli)
            self.post_order(root.right, fli)
            fli.reverse()             # 将后序遍历的列表倒序
            if bli == fli:
                return True
            else:
                return False

    def pre_order(self,root,li):     # 二叉树的前序遍历
        if root:
            li.append(root.val)
            self.pre_order(root.left,li)
            self.pre_order(root.right,li)
        elif root == None:
            li.append(None)

    def post_order(self,root,li):    # 二叉树的后序遍历
        if root:
            self.post_order(root.left,li)
            self.post_order(root.right,li)
            li.append(root.val)
        elif root == None:
            li.append(None)
```

# 29、顺时针打印矩阵

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。

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

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

**题解**

1、判断边界条件

- up, down, left, right 分别表示四个方向的边界。
- x, y 表示当前位置。
- dirs 分别表示移动方向是 右、下、左、上 。
- cur_d 表示当前的移动方向的下标，dirs[cur_d] 就是下一个方向需要怎么修改 x, y。
- cur_d == 0 and y == right 表示当前的移动方向是向右，并且到达了右边界，此时将移动方向更改为向下，并且上边界 up 向下移动一格。
- 结束条件是结果数组 res 的元素个数能与 matrix 中的元素个数。

```
class Solution(object):
    def spiralOrder(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        if not matrix or not matrix[0]: return []
        M, N = len(matrix), len(matrix[0])
        left, right, up, down = 0, N - 1, 0, M - 1
        res = []
        x, y = 0, 0
        dirs = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        cur_d = 0
        while len(res) != M * N:
            res.append(matrix[x][y])
            if cur_d == 0 and y == right:
                cur_d += 1
                up += 1
            elif cur_d == 1 and x == down:
                cur_d += 1
                right -= 1
            elif cur_d == 2 and y == left:
                cur_d += 1
                down -= 1
            elif cur_d == 3 and x == up:
                cur_d += 1
                left += 1
            cur_d %= 4
            x += dirs[cur_d][0]
            y += dirs[cur_d][1]
        return res
```

2、递归，每次剥掉最外面一圈数字，如果遇到0，或者一行，就是出口

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        def circle(i,j,m,n):
            if i > j or m > n:
                return []
            elif i == j:
                return matrix[i][m:n+1]
            elif m == n:
                res = []
                for x in range(i,j+1):
                    res.append(matrix[x][m])
                return res
            else:
                res = []
                res.extend(matrix[i][m:n+1])
                for x in range(i+1, j):
                    res.append(matrix[x][n])
                res.extend(reversed(matrix[j][m:n+1]))
                for x in range(j-1, i, -1):
                    res.append(matrix[x][m])
                return res + circle(i+1,j-1,m+1,n-1)

        return circle(0,len(matrix)-1,0,len(matrix[0])-1)
```

## 30、包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.min();   --> 返回 -2.
```

**题解**

1、一个栈保存两个数，一个x，一个当前最小值

```
class MinStack:
    def __init__(self):

        self.stack=[]

    def push(self, x: int) -> None:
        if not self.stack:
            self.stack.append((x,x))
```

```
        else:
            if x<=self.stack[-1][-1]:
                self.stack.append((x,x))
            else:
                self.stack.append((x,self.stack[-1][-1]))

    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0]

    def min(self) -> int:
        return self.stack[-1][-1]
```

2、辅助栈

```
class MinStack:
    def __init__(self):
        self.A = []
        self.B = []

    def push(self, x: int) -> None:
        self.A.append(x)
        if not self.B or x <= self.B[-1]:
            self.B.append(x)

    def pop(self) -> None:
        a = self.A.pop()
        if a == self.B[-1]:
            self.B.pop()

    def top(self) -> int:
        return self.A[-1]

    def min(self) -> int:
        return self.B[-1]
```

# 31-40

## 31、栈的压入、弹出序列

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为该栈的弹出顺序。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。

> ```
> 输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
> 输出：true
> 解释：我们可以按以下顺序执行：
> push(1), push(2), push(3), push(4), pop() -> 4,
> push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
> ```

```
输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。
```

---

**题解**

1、模拟入栈、出栈

1. 初始化： 辅助栈 stackstack ， 弹出序列的索引 ii ；
2. 遍历压栈序列： 各元素记为 numnum ；
    1. 元素 numnum 入栈；
    2. 循环出栈：若 stackstack 的栈顶元素 == 弹出序列元素 popped[i]popped[i] ， 则执行出栈与 i++i++ ；
3. 返回值： 若 stackstack 为空，则此弹出序列合法。

```
class Solution:
    def validateStackSequences(self, pushed: List[int], popped: List[int]) ->
bool:
        stack, i = [], 0
        for num in pushed:
            stack.append(num) # num 入栈
            while stack and stack[-1] == popped[i]: # 循环判断与出栈
                stack.pop()
                i += 1
        return not stack
```

## 32-Ⅰ、从上到下打印二叉树

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。

```
  3
 / \
9  20
  /  \
 15   7

[3,9,20,15,7]
```

---

**题解**

1、BFS

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        if not root:return[]
        cur,res=[root],[]
        while cur:
            node=cur.pop(0)
            res.append(node.val)
            if node.left:cur.append(node.left)
            if node.right:cur.append(node.right)

        return res
```

2、DFS

- DFS 不是按照层次遍历的。为了让递归的过程中同一层的节点放到同一个列表中，在递归时要记录每个节点的深度 level。递归到新节点要把该节点放入 level 对应列表的末尾。
    - 当遍历到一个新的深度 level，而最终结果 res 中还没有创建 level 对应的列表时，应该在 res 中新建一个列表用来保存该 level 的所有节点。

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        res=[]
        self.level(root,0,res)
        return [k for i in res for k in i]

    def level(self,root,l,res):
        if not root:return
        if len(res)==l:res.append([])
        res[l].append(root.val)
        if root.left:self.level(root.left,l+1,res)
        if root.right:self.level(root.right,l+1,res)
```

## 32-II、从上到下打印二叉树II

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。

```
  3
/ \
9  20
  /  \
15   7

[
[3],
[9,20],
[15,7]
]
```

**题解**

1、同上题DFS

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        res=[]
        self.dfs(root,0,res)
        return res

    def dfs(self,root,l,res):
        if not root:return
        if len(res)==l:res.append([])
        res[l].append(root.val)
        if root.left:self.dfs(root.left,l+1,res)
        if root.right:self.dfs(root.right,l+1,res)
```

2、BFS，增加一个判断层的for循环

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:return []
        stack,res=[root],[]
        i=0
        while stack:
            res.append([])
            for _ in range(len(stack)):
                node=stack.pop(0)
                res[i].append(node.val)
                if node.left:stack.append(node.left)
                if node.right:stack.append(node.right)
            i+=1

        return res
```

## 32-Ⅲ、从上到下打印二叉树Ⅲ

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。

```
 3
/ \
9  20
 /  \
15   7

[
[3],
[20,9],
[15,7]
]
```

**题解**

1、同上题，最后的输出，根据层数反转即可

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        res=[]
        self.dfs(root,0,res)
        x=[]
        for i,temp in enumerate(res):
            x.append(temp[::-1] if i%2==1 else temp)          #增加这一步即可，偶数层的反转
        return x

    def dfs(self,root,l,res):
        if not root:return
        if len(res)==l:res.append([])
        res[l].append(root.val)
        if root.left:self.dfs(root.left,l+1,res)
        if root.right:self.dfs(root.right,l+1,res)
```

2、层序遍历 + 双端队列

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        res, deque = [], collections.deque([root])
        while deque:
            tmp = collections.deque()
            for _ in range(len(deque)):
                node = deque.popleft()
                if len(res) % 2: tmp.appendleft(node.val) # 偶数层 -> 队列头部
                else: tmp.append(node.val) # 奇数层 -> 队列尾部
                if node.left: deque.append(node.left)
                if node.right: deque.append(node.right)
            res.append(list(tmp))
        return res
```

3、层序遍历 + 双端队列（奇偶层逻辑分离）

- while循环，一次两层，中间为空提前跳出

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        res, deque = [], collections.deque()
        deque.append(root)
        while deque:
            tmp = []
            # 打印奇数层
            for _ in range(len(deque)):
                # 从左向右打印
                node = deque.popleft()
                tmp.append(node.val)
                # 先左后右加入下层节点
                if node.left: deque.append(node.left)
                if node.right: deque.append(node.right)
            res.append(tmp)
            if not deque: break # 若为空则提前跳出
            # 打印偶数层
```

```
            tmp = []
            for _ in range(len(deque)):
                # 从右向左打印
                node = deque.pop()
                tmp.append(node.val)
                # 先右后左加入下层节点
                if node.right: deque.appendleft(node.right)
                if node.left: deque.appendleft(node.left)
            res.append(tmp)
        return res
```

## 33、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

```
  5
 / \
2   6
/ \
1   3
```

```
输入: [1,6,3,2,5]
输出: false
```

```
输入: [1,3,2,6,5]
输出: true
```

**题解**

1、递归分治

- 终止条件： 当 i≥j ，说明此子树节点数量 ≤1 ，无需判别正确性，因此直接返回true；
- 递推工作：
    1. 划分左右子树： 遍历后序遍历的 [i, j]区间元素，寻找 第一个大于根节点 的节点，索引记为 m 。此时，可划分出左子树区间 [i,m-1]、右子树区间 [m, j - 1]、根节点索引 j 。
    2. 判断是否为二叉搜索树：
        - 左子树区间 [i, m - 1] 内的所有节点都应 <postorder[j] 。而第 1.划分左右子树 步骤已经保证左子树区间的正确性，因此只需要判断右子树区间即可。
        - 右子树区间 [m, j-1] 内的所有节点都应 >postorder[j] 。实现方式为遍历，当遇到 ≤postorder[j] 的节点则跳出；则可通过 p = j 判断是否为二叉搜索树。
- 返回值： 所有子树都需正确才可判定正确，因此使用 与逻辑符 && 连接。
    1. p = j ： 判断 此树 是否正确。
    2. recur(i, m - 1)： 判断 此树的左子树 是否正确。
    3. recur(m, j - 1)： 判断 此树的右子树 是否正确。

```python
class Solution:
    def verifyPostorder(self, postorder: [int]) -> bool:
        def recur(i, j):
            if i >= j: return True
            p = i
            while postorder[p] < postorder[j]: p += 1
            m = p
            while postorder[p] > postorder[j]: p += 1
            return p == j and recur(i, m - 1) and recur(m, j - 1)

        return recur(0, len(postorder) - 1)
```

2、辅助单调栈

1. 初始化： 单调栈 stack ， 父节点值 root=+∞ （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 ri;
    1. 判断： 若 ri>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 ri<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 ri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。

```python
class Solution:
    def verifyPostorder(self, postorder: [int]) -> bool:
        stack, root = [], float("+inf")
        for i in range(len(postorder) - 1, -1, -1):
            if postorder[i] > root: return False
            while(stack and postorder[i] < stack[-1]):
                root = stack.pop()
            stack.append(postorder[i])
        return True
```

## 34、二叉树中和为某一值的路径

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

```
            5
           / \
          4   8
         /   / \
        11  13  4
       /  \    / \
      7    2  5   1
target = 22
```

```
[
[5,4,11,2],
[5,8,4,5]
]
```

---

**题解**

1、回溯

- 递推参数： 当前节点 root ， 当前目标值 tar 。
- 终止条件： 若节点 root 为空，则直接返回。
- 递推工作：
    1. 路径更新： 将当前节点值 root.val 加入路径 path ；
    2. 目标值更新： tar = tar - root.val（即目标值 tar 从 sum 减至 00 ）；
    3. 路径记录： 当 ① root 为叶节点 且 ② 路径和等于目标值 ， 则将此路径 path 加入 res 。
    4. 先序遍历： 递归左 / 右子节点。
    5. 路径恢复： 向上回溯前，需要将当前节点从路径 path 中删除，即执行 path.pop() 。

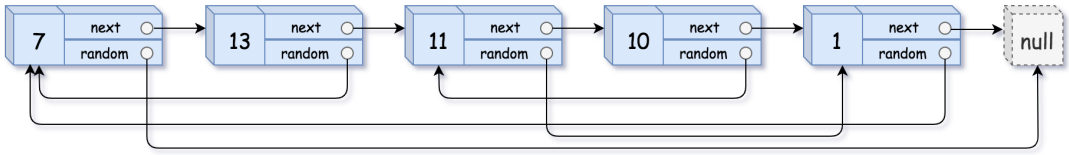
```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> List[List[int]]:
        res, path = [], []
        def recur(root, tar):
            if not root: return
            path.append(root.val)
            tar -= root.val
            if tar == 0 and not root.left and not root.right:
                res.append(list(path))
            recur(root.left, tar)
            recur(root.right, tar)
            path.pop()

        recur(root, sum)
        return res
```

2、BFS

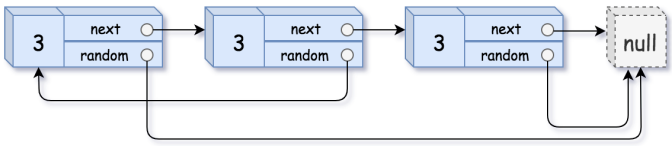
```
#itrative BFS
        if not root: return []
        queue = [(root, root.val, [root.val])]
        res = []
        while queue:
            node, val, temp = queue.pop(0)
            if val == sum and not node.left and not node.right: res.append(temp)
            if node.left:
                queue.append((node.left, val + node.left.val, temp+
[node.left.val]))
            if node.right:
                queue.append((node.right, val + node.right.val , temp+
[node.right.val]))
        return res
```

# 35、复杂链表的复制

请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。




```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```




```
输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]
```

**题解**

1、哈希表

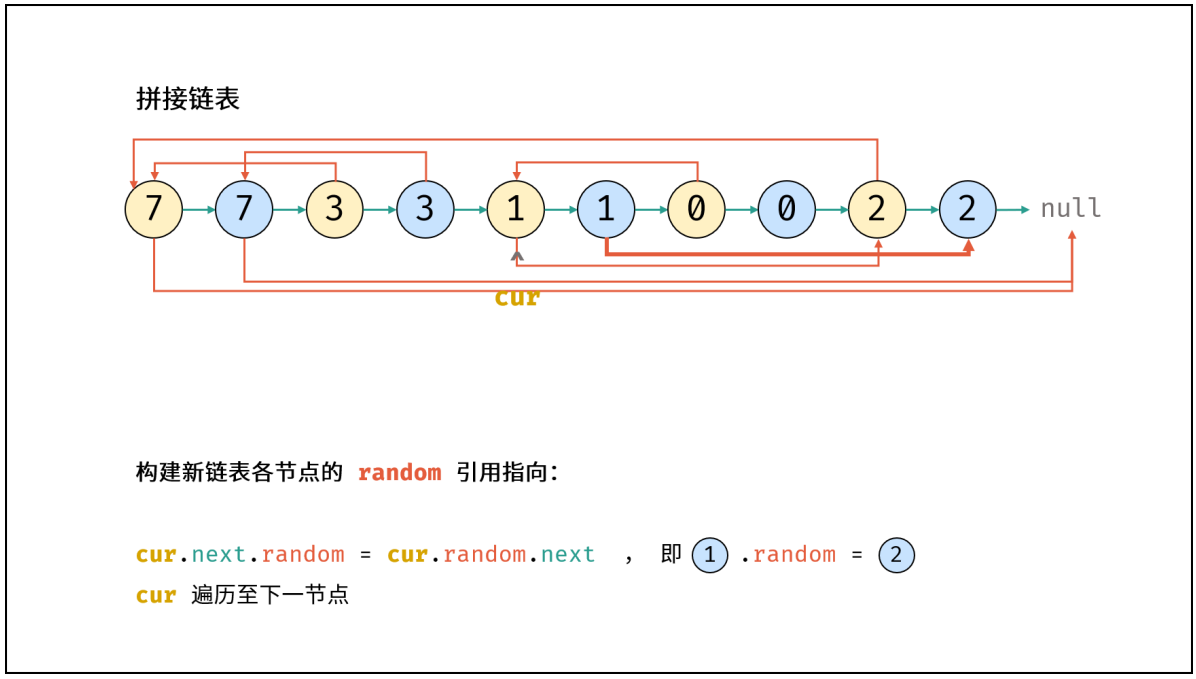
1. 若头节点 head 为空节点，直接返回 null ；
2. 初始化： 哈希表 dic ， 节点 cur 指向头节点；
3. 复制链表：
    1. 建立新节点，并向 dic 添加键值对 (原 cur 节点, 新 cur 节点) ；
    2. cur 遍历至原链表下一节点；
4. 构建新链表的引用指向：
    1. 构建新节点的 next 和 random 引用指向；
    2. cur 遍历至原链表下一节点；
5. 返回值： 新链表的头节点 dic[cur] ；

```python
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head: return
        dic = {}
        # 3. 复制各节点，并建立 “原节点 -> 新节点” 的 Map 映射
        cur = head
        while cur:
            dic[cur] = Node(cur.val)
            cur = cur.next
        cur = head
        # 4. 构建新节点的 next 和 random 指向
        while cur:
            dic[cur].next = dic.get(cur.next)
```

```
            dic[cur].random = dic.get(cur.random)
            cur = cur.next
        # 5. 返回新链表的头节点
        return dic[head]
```

2、拼接 + 拆分

1. 复制各节点，构建拼接链表:
    - 设原链表为 node1→node2→⋯ ，构建的拼接链表如下所示:
      node1→node1new→node2→node2 new→⋯
2. 构建新链表各节点的 random 指向:
    - 当访问原节点 cur 的随机指向节点 cur.random 时，对应新节点 cur.next 的随机指向节点为 cur.random.next 。
3. 拆分原 / 新链表:
    - 设置 pre / cur 分别指向原 / 新链表头节点，遍历执行 pre.next = pre.next.next 和 cur.next = cur.next.next 将两链表拆分开。
4. 返回新链表的头节点 res 即可。

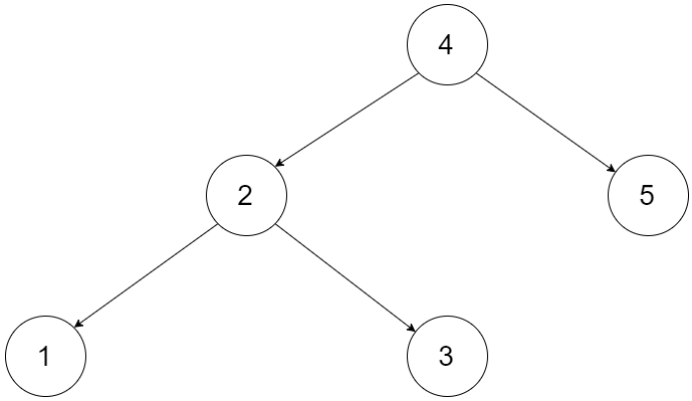



```
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head: return
        cur = head
        # 1. 复制各节点，并构建拼接链表
        while cur:
            tmp = Node(cur.val)
            tmp.next = cur.next
            cur.next = tmp
            cur = tmp.next
        # 2. 构建各新节点的 random 指向
        cur = head
        while cur:
            if cur.random:
                cur.next.random = cur.random.next
            cur = cur.next.next
        # 3. 拆分两链表
        cur = res = head.next
```

```
        pre = head
        while cur.next:
            pre.next = pre.next.next
            cur.next = cur.next.next
            pre = pre.next
            cur = cur.next
        pre.next = None # 单独处理原链表尾节点
        return res      # 返回新链表头节点
```

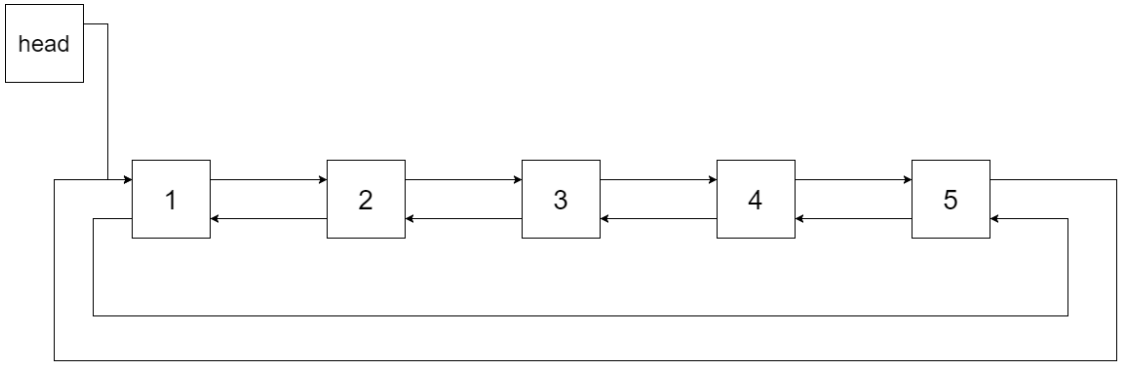
## 36、二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。




我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

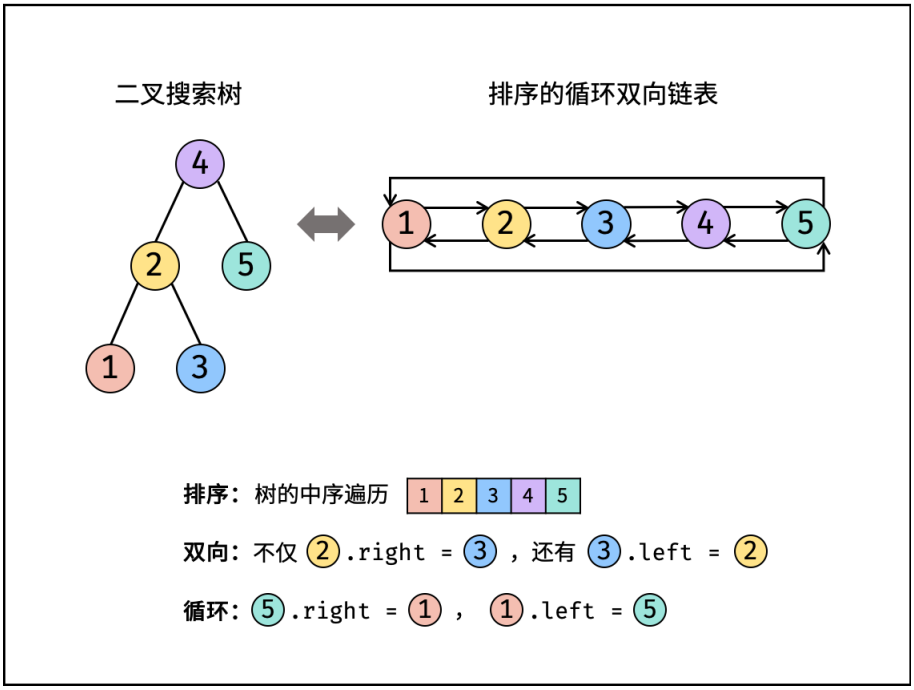
下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。




特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

**题解**

1、中序遍历




`dfs(cur)`：递归法中序遍历；

1. 终止条件： 当节点 cur 为空，代表越过叶节点，直接返回；
2. 递归左子树，即 dfs(cur.left) ；
3. 构建链表：
    1. 当 pre 为空时： 代表正在访问链表头节点，记为 head ；
    2. 当 pre 不为空时： 修改双向节点引用，即 pre.right = cur ， cur.left = pre ；
    3. 保存 cur ： 更新 pre = cur ，即节点 cur 是后继节点的 pre ；
4. 递归右子树，即 dfs(cur.right) ；

treeToDoublyList(root)：

1. 特例处理： 若节点 root 为空，则直接返回；
2. 初始化： 空节点 pre ；
3. 转化为双向链表： 调用 dfs(root) ；
4. 构建循环链表： 中序遍历完成后，head 指向头节点， pre 指向尾节点，因此修改 head 和 pre 的双向节点引用即可；
5. 返回值： 返回链表的头节点 head 即可；

```
class Solution:
    def treeToDoublyList(self, root: 'Node') -> 'Node':
        def dfs(cur):
            if not cur: return
            dfs(cur.left) # 递归左子树
            if self.pre: # 修改节点引用
                self.pre.right, cur.left = cur, self.pre
            else: # 记录头节点
                self.head = cur
            self.pre = cur # 保存 cur
            dfs(cur.right) # 递归右子树

        if not root: return
        self.pre = None
        dfs(root)
        self.head.left, self.pre.right = self.pre, self.head
        return self.head
```

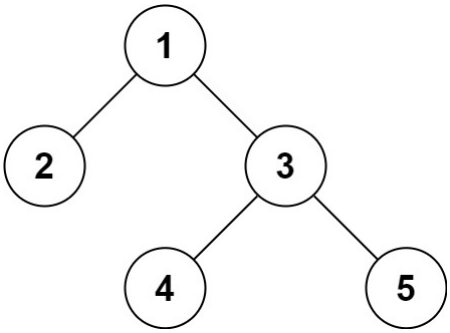
2、迭代，根据中序遍历模板改

```python
class Solution:
    def treeToDoublyList(self, root: TreeNode) -> TreeNode:
        if not root:
            return None
        # pre 记录上一个遍历结点，s作为遍历栈
        pre, head, s = root, None, []

        # 常规二叉树的迭代中序遍历
        while pre:
            s.append(pre)
            pre = pre.left
        head = s[-1]
        while s:
            cur = s.pop()
            if pre:
                pre.right = cur
            cur.left = pre
            pre = cur
            cur = cur.right
            while cur:
                s.append(cur)
                cur = cur.left

        # 衔接首尾结点形成循环链表
        head.left, pre.right = pre, head
        return head
```

## 37、序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树。

你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。




```
输入：root = [1,2,3,null,null,4,5]
输出：[1,2,3,null,null,4,5]
```

**题解**

1、BFS

- 序列化
    1. 用BFS遍历树，与一般遍历的不同点是不管node的左右子节点是否存在，统统加到队列中
    2. 在节点出队时，如果节点不存在，在返回值res中加入一个”null“；如果节点存在，则加入节点值的字符串形式
- 反序列化
    1. 同样使用BFS方法，利用队列新建二叉树
    2. 首先要将data转换成列表，然后遍历，只要不为null将节点按顺序加入二叉树中；同时还要将节点入队
    3. 队列为空时遍历完毕，返回根节点

```
class Codec:
    def serialize(self, root):
        if not root:
            return ""
        queue = collections.deque([root])
        res = []
        while queue:
            node = queue.popleft()
            if node:
                res.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else:
                res.append('None')
        return '[' + ','.join(res) + ']'

    def deserialize(self, data):
        if not data:
            return []
        dataList = data[1:-1].split(',')
        root = TreeNode(int(dataList[0]))
        queue = collections.deque([root])
        i = 1
        while queue:
            node = queue.popleft()
            if dataList[i] != 'None':
                node.left = TreeNode(int(dataList[i]))
                queue.append(node.left)
            i += 1
            if dataList[i] != 'None':
                node.right = TreeNode(int(dataList[i]))
                queue.append(node.right)
            i += 1
        return root
```

2、DFS

- 序列化
    1. 递归的第一步都是特例的处理，因为这是递归的中止条件：如果根节点为空，返回”null“
    2. 序列化的结果为：根节点值 + "," + 左子节点值(进入递归) + "," + 右子节点值(进入递归)
    3. 递归就是不断将“根节点”值加到结果中的过程
- 反序列化
    1. 先将字符串转换成队列（python转换成列表即可）
    2. 接下来就进入了递归

1. i. 弹出左侧元素，即队列出队
2. ii. 如果元素为“null”，返回null（python返回None）
3. iii. 否则，新建一个值为弹出元素的新节点
4. iv. 其左子节点为队列的下一个元素，进入递归；右子节点为队列的下下个元素，也进入递归
5. v. 递归就是不断将子树的根节点连接到父节点的过程

```
class Codec:
    def serialize(self, root):
        if not root:
            return 'None'
        return str(root.val) + ',' + str(self.serialize(root.left)) + ',' +
str(self.serialize(root.right))

    def deserialize(self, data):
        def dfs(dataList):
            val = dataList.pop(0)
            if val == 'None':
                return None
            root = TreeNode(int(val))
            root.left = dfs(dataList)
            root.right = dfs(dataList)
            return root

        dataList = data.split(',')
        return dfs(dataList)
```

## 38、字符串的排列

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

> 输入：s = "abc"
> 输出：["abc","acb","bac","bca","cab","cba"]

**题解**

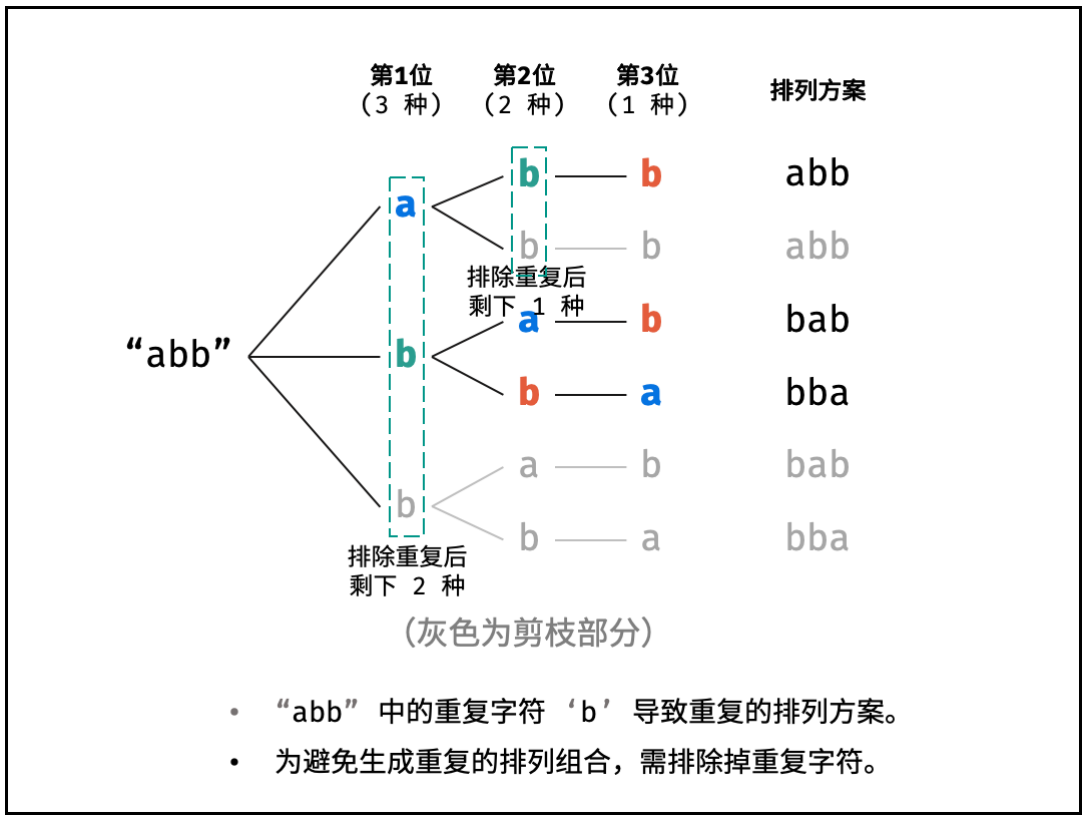
1、递归——不重复一样的列表元素

- 递归思想
  - 如果字符无重复，那么“abc”--->"a"+"bc"、“b"+"ac"、"c"+"ab",其中“bc”--->"b"+"c"会遍历两次，....依次类推，所以2+2+2=6。
  - 如果字符有重复，例如”aab“--->"a"+"ab"、"b"+"aa",其中”ab“会遍历两次，”aa“会遍历一次，所以2+1=3。
  - 所以使用set(s)，保证每次遍历不一样的列表元素即可去重
- 递归流程
  1. 递归终止条件：s为空，保存序列
  2. 遍历set(s),不重复一样的列表元素
     1. new_s=s

2. 删除new_s中的i
3. 继续递归

```
class Solution:
    def permutation(self, s: str) -> List[str]:
        res=[]
        def curer(ans,s):
            if not s:return res.append(ans)
            for i in set(s):
                new_s=s
                new_s=new_s.replace(i,'',1)
                curer(ans+i,new_s)
        curer("",s)
        return res
```

2、回溯+剪枝




1. 终止条件： 当 x = len(c) - 1 时，代表所有位已固定（最后一位只有 11 种情况），则将当前组合 c 转化为字符串并加入 res ， 并返回；
2. 递推参数： 当前固定位 x ；
3. 递推工作： 初始化一个 Set ， 用于排除重复的字符；将第 x 位字符与 i \in∈ [x, len(c)] 字符分别交换，并进入下层递归；
    1. 剪枝： 若 c[i] 在 Set 中，代表其是重复字符，因此 “剪枝” ；
    2. 将 c[i] 加入 Set ， 以便之后遇到重复字符时剪枝；
    3. 固定字符： 将字符 c[i] 和 c[x] 交换，即固定 c[i] 为当前位字符；
    4. 开启下层递归： 调用 dfs(x + 1) ，即开始固定第 x + 1 个字符；
    5. 还原交换： 将字符 c[i] 和 c[x] 交换（还原之前的交换）；

```
class Solution:
    def permutation(self, s: str) -> List[str]:
        c, res = list(s), []
        def dfs(x):
```

```
            if x == len(c) - 1:
                res.append(''.join(c))   # 添加排列方案
                return
            dic = set()
            for i in range(x, len(c)):
                if c[i] in dic: continue # 重复，因此剪枝
                dic.add(c[i])
                c[i], c[x] = c[x], c[i]  # 交换，将 c[i] 固定在第 x 位
                dfs(x + 1)               # 开启固定第 x + 1 位字符
                c[i], c[x] = c[x], c[i]  # 恢复交换
        dfs(0)
        return res
```

## 39、数组中出现次数超过一半的数字

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

> 输入: [1, 2, 3, 2, 2, 2, 5, 4, 2]
> 输出: 2

**题解**

1、一行代码

- 排序之后，数组中间的数一定是答案

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        return sorted(nums)[len(nums)//2]
```

2、摩尔投票法

- 核心理念为 **票数正负抵消** 。此方法时间和空间复杂度分别为 O(N)*O*(*N*) 和 O(1)*O*(1)

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        votes = 0
        for num in nums:
            if votes == 0: x = num
            votes += 1 if num == x else -1
        return x
```

3、字典计数

```
class Solution(object):
    def majorityElement(self, nums):
        if not nums:
            return None
        if len(nums) == 1:
            return nums[0]
```

```
        write = {}
        for i in nums:
            if i not in write:
                write[i] = 1
            else:
                write[i] += 1
                if write[i] > (len(nums) / 2):
                    return i
        return None
```

## 40、最小的K个数

输入整数数组 `arr` ，找出其中最小的 `k` 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

> 输入：arr = [3,2,1], k = 2
> 输出：[1,2] 或者 [2,1]

> 输入：arr = [0,1,2,1], k = 1
> 输出：[0]

**题解**

1、排序输出

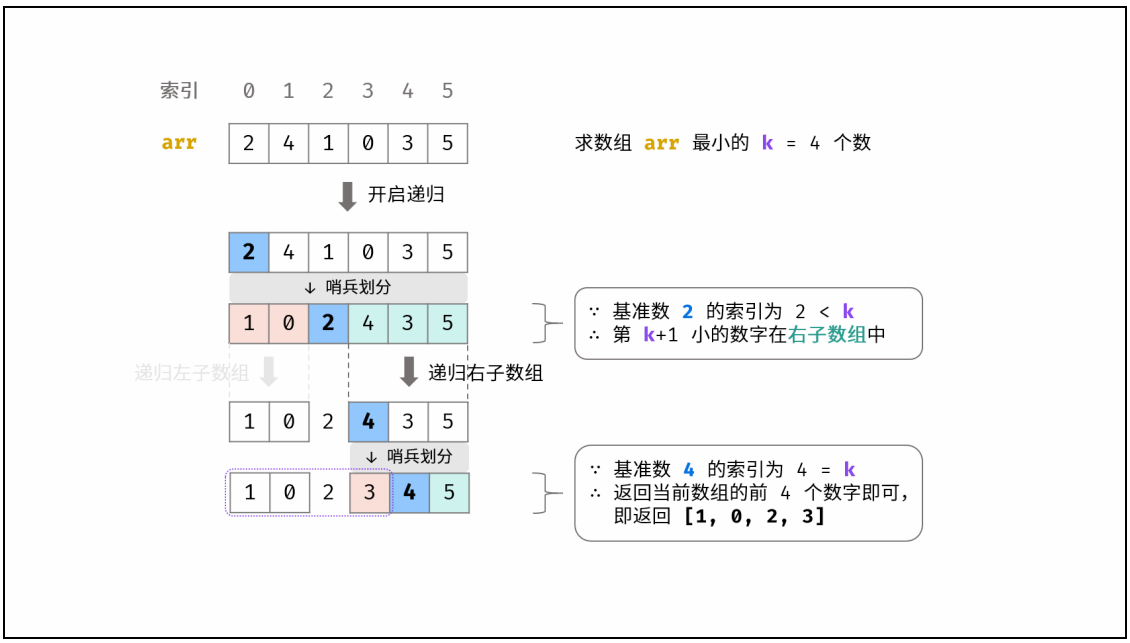
```
class Solution:
    def getLeastNumbers(self, arr: List[int], k: int) -> List[int]:
        return sorted(arr)[:k]
```

2、两次遍历，

- 第一遍循环对每个值计数
- 第二遍从小到大找齐k个元素

```
class Solution:
    def getLeastNumbers(self, arr: List[int], k: int) -> List[int]:
        rec,res = [0]*10001, []
        for ar in arr: rec[ar]+=1
        for i,r in enumerate(rec):
            if r>0:
                if k>r:
                    res += [i]*r
                    k-=r
                else:
                    res += [i]*k
                    break
        return res
```

3、快速排序




```python
class Solution:
    def getLeastNumbers(self, arr: List[int], k: int) -> List[int]:
        if k >= len(arr): return arr
        def quick_sort(l, r):
            i, j = l, r
            while i < j:
                while i < j and arr[j] >= arr[l]: j -= 1
                while i < j and arr[i] <= arr[l]: i += 1
                arr[i], arr[j] = arr[j], arr[i]
            arr[l], arr[i] = arr[i], arr[l]
            if k < i: return quick_sort(l, i - 1)
            if k > i: return quick_sort(i + 1, r)
            return arr[:k]

        return quick_sort(0, len(arr) - 1)
```

# 41-50

## 41、数据流中的中位数

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

> 输入：
> ["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]
> [[],[1],[2],[],[3],[]]
> 输出：[null,null,null,1.50000,null,2.00000]

```
输入：
["MedianFinder","addNum","findMedian","addNum","findMedian"]
[[],[2],[],[3],[]]
输出：[null,null,2.00000,null,2.50000]
```

---

**题解**

1、添加数后就排序，输出中位数

```
class MedianFinder:
    def __init__(self):
        self.nums=[]

    def addNum(self, num: int) -> None:
        self.nums.append(num)
        self.nums.sort()

    def findMedian(self) -> float:
        if len(self.nums)%2:
            return self.nums[len(self.nums)//2]
        else:
            return (self.nums[len(self.nums)//2]+self.nums[len(self.nums)//2-
1])/2
```

2、两个堆

- 建立一个 **小顶堆** *A* 和 **大顶堆** *B* ，各保存列表的一半元素，且规定：
  - A保存较大的一半，长度为N/2（N为偶数），或（N+1）/2（N为奇数）
  - B保存较小的一半，长度为N/2（N为偶数），或（N+1）/2（N为奇数）
- addNum(num) 函数：
  1. 当 m = n（即 N 为 偶数）：需向 A 添加一个元素。实现方法：将新元素 num 插入至 B ，再将 B 堆顶元素插入至 A ；
  2. 当 m \n（即 N 为 奇数）：需向 B 添加一个元素。实现方法：将新元素 num 插入至 A ，再将 A 堆顶元素插入至 B ；
  3. 假设插入数字 num 遇到情况 1. 。由于 num 可能属于 “较小的一半” （即属于 B ），因此不能将 nums 直接插入至 A 。而应先将 num 插入至 B ，再将 B 堆顶元素插入至 A 。这样就可以始终保持 A 保存较大一半、 B 保存较小一半。
- findMedian() 函数：
  1. 当 m = n （N 为 偶数）：则中位数为 (( A 的堆顶元素 + B 的堆顶元素 )/2)。
  2. 当 m \n （ N 为 奇数）：则中位数为 A 的堆顶元素。

```
from heapq import *
class MedianFinder:
    def __init__(self):
        self.A = [] # 小顶堆，保存较大的一半
        self.B = [] # 大顶堆，保存较小的一半

    def addNum(self, num: int) -> None:
        if len(self.A) != len(self.B):
            heappush(self.A, num)
            heappush(self.B, -heappop(self.A))
        else:
            heappush(self.B, -num)
```

```
            heappush(self.A, -heappop(self.B))

    def findMedian(self) -> float:
        return self.A[0] if len(self.A) != len(self.B) else (self.A[0] -
self.B[0]) / 2.0
```

## 42、连续子数组的最大和

输入一个整型数组，数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值。

要求时间复杂度为O(n)。

> 输入: nums = [-2,1,-3,4,-1,2,1,-5,4]
> 输出: 6
> 解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

**题解**

1、动态规划：基本思路就是遍历一遍，用两个变量，一个记录最大的和，一个记录当前的和。

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        tmp=nums[0]
        res=tmp
        n=len(nums)
        for i in range(1,n):
            if tmp>0:
                res=max(res,tmp+nums[i])
                tmp+=nums[i]
            else:
                res=max(res,tmp,nums[i])
                tmp=nums[i]

        return res
```

2、分治法：其实就是它的最大子序和要么在左半边，要么在右半边，要么是穿过中间，对于左右边的序列，情况也是一样，因此可以用递归处理。中间部分的则可以直接计算出来，时间复杂度应该是O(nlogn)。

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        #递归终止条件
        if n == 1:
            return nums[0]
        else:
            #递归计算左半边最大子序和
            max_left = self.maxSubArray(nums[0:len(nums) // 2])
            #递归计算右半边最大子序和
            max_right = self.maxSubArray(nums[len(nums) // 2:len(nums)])
```

```
        #计算中间的最大子序和，从右到左计算左边的最大子序和，从左到右计算右边的最大子序和，再相加
        max_l = nums[len(nums) // 2 - 1]
        tmp = 0
        for i in range(len(nums) // 2 - 1, -1, -1):
            tmp += nums[i]
            max_l = max(tmp, max_l)
        max_r = nums[len(nums) // 2]
        tmp = 0
        for i in range(len(nums) // 2, len(nums)):
            tmp += nums[i]
            max_r = max(tmp, max_r)
        #返回三个中的最大值
        return max(max_right,max_left,max_l+max_r)
```

## 43、1~n整数中1出现的次数

输入一个整数 n ， 求1～n这n个整数的十进制表示中1出现的次数。

例如，输入12，1～12这些整数中包含1 的数字有1、10、11和12，1一共出现了5次。

> 输入：n = 12
> 输出：5

> 输入：n = 13
> 输出：6

**题解**

1、总结规律

- 首先算出位数的规律，得到位数字典
  - 1-9——1
  - 1-99——10+1*9+1=20
  - 1-999——100+20*9+20=300
  - 1-9999——1000+300*9+300=4000

```
        dic={}
        num=0
        for i in range(20):
            dic[i]=num
            num=10**i+num*10
        print(dic)
dic={0: 0, 1: 1, 2: 20, 3: 300, 4: 4000, 5: 50000, 6: 600000, 7: 7000000, 8: 80000000, 9: 900000000, 10: 10000000000, 11: 110000000000, 12: 1200000000000, 13: 13000000000000, 14: 140000000000000, 15: 1500000000000000, 16: 16000000000000000, 17: 170000000000000000, 18: 1800000000000000000, 19: 19000000000000000000}
```

- 将数字n转换为字符串s，按位计算，几个特例举例说明
  - `584——（100+20*5）+（10+1*8）+（1）`

        - 因为5大于1，所以100-200的百分位上有100个1，0-500共有五个1-99区间，所以+5*20
        - 接下来计算84中有多少1，同理，8大于1，所以10-20十分位上有10个1，0-80有八个1-9区间，所以8*1
        - 最后判断4，大于1，有一个1
    - `112--（12+1+20）+（2+1+1）+1`
        - 因为1等于1，所以在100-112在百分位上有12+1个1，同时只有一个1-99区间，所以+20
        - 接下来计算12中有多少1，同理，1等于1，所以10-12有2+1个1，同时有一个1-9区间，所以要+1
        - 因为2大于1，结果+1
    - 101——（1+1+20）+（1）
        - 百分位1等于1，100-101共有1+1个，同时只有一个1-99区间
        - 遇0跳过
        - 判断最后1位是否大于1

```
class Solution:
    def countDigitOne(self, n: int) -> int:
        dic={}
        num=0
        for i in range(20):
            dic[i]=num
            num=10**i+num*10
        print(dic)
        res=0
        s=str(n)
        l=len(s)-1
        for i in range(len(s)-1):
            if int(s[i])>1:
                res+=(10**l+int(s[i])*dic[l])
                l-=1
            elif s[i]=='1':
                res+=(int(s[i+1:])+dic[l]+1)
                l-=1
            else:
                l-=1

        return res if s[-1]=='0' else res+1
```

2、从后向前算

1. 设数字n是个x位数，记n的第i位为ni，则可将n写为nxnx-1...n2n1：
    - 称ni为当前位，记为cur
    - 将ni-1ni-2...n2n1称为低位，记为low
    - 将nxnx-1...ni2ni+1称为高位，记为high
    - 将10^i称为位因子，记为digit
2. 根据当前位 cur值的不同，分为以下三种情况：
    1. 当 cur = 0 时： 此位 1 的出现次数只由高位 high 决定，计算公式为：

       high*digit
    2. 当 cur = 1时： 此位 1 的出现次数由高位 high和低位 low决定，计算公式为：

       high*digit+low+1
    3. 当 cur =2,3,⋯,9 时： 此位 1 的出现次数只由高位 high决定，计算公式为：

       (high+1)*digit

```python
class Solution:
    def countDigitOne(self, n: int) -> int:
        digit, res = 1, 0
        high, cur, low = n // 10, n % 10, 0
        while high != 0 or cur != 0:
            if cur == 0: res += high * digit
            elif cur == 1: res += high * digit + low + 1
            else: res += (high + 1) * digit
            low += cur * digit
            cur = high % 10
            high //= 10
            digit *= 10
        return res
```

3、余数规则

1. 0 ~ 9 只有一个1，那么如果某数是10的a倍，那么个位上就有a个1;
2. 10 ~ 99 十位上有10个1，即10 ~ 19，那么如果某数是 100 的a倍，是10的b倍，那么十位有a * 10个1，个位有b个1，共 (a * 10 + b * 1) 个1;
3. 100 ~ 999 百位上有100个，即100 ~ 199，那么如果某数是 1000 的a倍， 100 的b倍，是10的c倍，那么百位有a * 100个1，十位有b * 10个1，个位有c个1，共 (a * 100 + b * 10 + c) 个1;
4. 案例分析
    1. a = 1, b = 10, 根据x, y = divmod(n, b)可知，x = 201， y = 2, 因为2 >= 2 * a，那么个位共有201 + 1个1, one_count = 202
    2. a = 10, b = 100, 此时x = 20， y = 12， 因为 a <= 12 < 2 * a，那么十位共有 (12 + 1 + (20 -1) * 10)个1, one_count = 202 + 203 = 405
    3. a = 100, b = 1000, 此时x = 2， y = 12， 因为 12 < a，那么百位共有 (2 * 100)个1, one_count = 405 + 200 = 605
    4. a = 1000, b = 10000, 此时x = 0， y = 2012， 因为 2012 >= 2 * a，那么千位共有1000个1, one_count = 605 + 1000 = 1605

```python
class Solution:
    def countDigitOne(self, n: int) ->  int:
        a, b, one_count = 1, 10, 0
        while n >= a:
            x, y = divmod(n, b)
            if y >= a * 2:
                one_count += (x + 1) * a
            elif y >= a:
                one_count += y + 1 + (x - 1) * a
            else:
                one_count += x * a
            a, b = b, b*10

        return one_count
```

# 44、数字序列中某一位的数字

数字以0123456789101112131415...的格式序列化到一个字符序列中。在这个序列中，第5位（从下标0开始计数）是5，第13位是1，第19位是4，等等。

请写一个函数，求任意第n位对应的数字。

> 输入：n = 3
> 输出：3

> 输入：n = 11
> 输出：0

1、找规律

规律：

1. 将 101112··· 中的每一位称为 数位 ， 记为 n ；
2. 将 10,11,12,··· 称为 数字 ， 记为 num ；
3. 数字 10 是一个两位数，称此数字的 位数 为 2 ， 记为 digit ；
4. 每 digit 位数的起始数字（即：1,10,100,···） ， 记为start 。

| 数字范围 | 位数 | 数字数量 | 数位数量 |
| --- | --- | --- | --- |
| $1 \sim 9$ | $1$ | $9$ | $9$ |
| $10 \sim 99$ | $2$ | $90$ | $180$ |
| $100 \sim 999$ | $3$ | $900$ | $2700$ |
| ... | ... | ... | ... |
| $start \sim end$ | $digit$ | $9 \times start$ | $9 \times start \times digit$ |

位数递推公式 $digit = digit + 1$

起始数字递推公式 $start = start \times 10$

数位数量计算公式 $count = 9 \times start \times digit$

求解分三步

1. 确定 n所在 数字 的 位数 ， 记为 digit；

```
digit, start, count = 1, 1, 9
while n > count:
    n -= count
    start *= 10 # 1, 10, 100, ...
    digit += 1  # 1,  2,  3, ...
    count = 9 * start * digit # 9, 180, 2700, ...
```

2. 确定 n所在的 数字 ， 记为 num ；

```
num = start + (n - 1) // digit
```

3. 确定 n 是 num中的哪一数位，并返回结果。

```
s = str(num) # 转化为 string
res = int(s[(n - 1) % digit]) # 获得 num 的 第 (n - 1) % digit 个数位，并转化为 int
```

```
class Solution:
    def findNthDigit(self, n: int) -> int:
        digit, start, count = 1, 1, 9
        while n > count: # 1.
            n -= count
            start *= 10
            digit += 1
            count = 9 * start * digit
        num = start + (n - 1) // digit # 2.
        return int(str(num)[(n - 1) % digit]) # 3.
```

## 45、把数组排成最小的数

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

> 输入: [10,2]
> 输出: "102"

> 输入: [3,30,34,5,9]
> 输出: "3033459"

**题解**

1、自定义排序

- 若拼接字符串 x + y > y + x ， 则 x “大于” y ；
- 反之，若 x + y < y + x ， 则 x “小于” y ；
- 使用快速排序，使用该规则排序

```
class Solution:
    def minNumber(self, nums: List[int]) -> str:
```

```
        def quick_sort(l , r):
            if l >= r: return
            i, j = l, r
            while i < j:
                while strs[j] + strs[l] >= strs[l] + strs[j] and i < j: j -= 1
                while strs[i] + strs[l] <= strs[l] + strs[i] and i < j: i += 1
                strs[i], strs[j] = strs[j], strs[i]
            strs[i], strs[l] = strs[l], strs[i]
            quick_sort(l, i - 1)
            quick_sort(i + 1, r)

        strs = [str(num) for num in nums]
        quick_sort(0, len(strs) - 1)
        return ''.join(strs)
```

2、同上，另一种排序写法

```
class cmpSmaller(str):
    def __lt__(self, y):
        return self + y < y + self  # 字符串拼接比较(两两比较)
    # 按由小到大来排列

class Solution:
    def minNumber(self, nums: List[int]) -> str:
        res=sorted(map(str, nums),key=cmpSmaller)
        smallest = ''.join(res)
        return smallest
```

3、把数字转换成小数在排序

```
class Solution:
    def minNumber(self, nums: List[int]) -> str:
        def cyclic(x):
            if x ==0 : return 0
            power = 0
            k = x
            while x:
                x //= 10
                power += 1
            return k/(10**power-1)
        nums.sort(key=cyclic)
        ans = list(map(str,nums))
        return ''.join(ans)
```

## 46、把数字翻译成字符串

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a” ，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

> 输入: 12258
> 输出: 5
> 解释: 12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"

---

**题解**

1、动态规划

$$num = x_1x_2\dots x_{i-2}x_{i-1}x_i\dots x_{n-1}x_n$$

（例如： $12258 = x_1x_2x_3x_4x_5$ ）

⬇

设 $x_1x_2\dots x_{i-2}$ 的翻译方案数量为 $f(i-2)$

设 $x_1x_2\dots x_{i-2}x_{i-1}$ 的翻译方案数量为 $f(i-1)$

⬇

当整体翻译 $x_{i-1}x_i$ 时，$x_1x_2\dots x_{i-2}x_{i-1}x_i$ 的方案数为 $f(i-2)$

当单独翻译 $x_i$ 时，$x_1x_2\dots x_{i-2}x_{i-1}x_i$ 的方案数为 $f(i-1)$

⬇

方案数的递推关系：

$$f(i) = \begin{cases} f(i-2) + f(i-1) & ，若数字\ x_{i-1}x_i\ 可被翻译 \\ f(i-1) & ，若数字\ x_{i-1}x_i\ 不可被翻译 \end{cases}$$

```python
class Solution:
    def translateNum(self, num: int) -> int:
        s = str(num)
        a = b = 1
        for i in range(2, len(s) + 1):
            a, b = (a + b if "10" <= s[i - 2:i] <= "25" else a), a
        return a
```

优化，不使用s存储字符串

```python
class Solution:
    def translateNum(self, num: int) -> int:
        a = b = 1
        y = num % 10
        while num != 0:
            num //= 10
            x = num % 10
            a, b = (a + b if 10 <= 10 * x + y <= 25 else a), a
            y = x
        return a
```

# 47、礼物的最大价值

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

> 输入:
> [
> [1,3,1],
> [1,5,1],
> [4,2,1]
> ]
> 输出: 12
> 解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物

**题解**

1、动态规划，创建dp数组，与grid同尺寸，每格表示到达该格的最大值

- `dp[i][j]=max(dp[i-1][j],dp[i][j-1])+grid[i][j]`

```
class Solution:
    def maxValue(self, grid: List[List[int]]) -> int:
        dp=[[grid[0][0]]*len(grid[0]) for _ in range(len(grid))]
        if len(grid)>1:     #先最左列
            for i in range(1,len(grid)):
                dp[i][0]=dp[i-1][0]+grid[i][0]
        for i in range(1,len(grid[0])):     #再第一行
            dp[0][i]=dp[0][i-1]+grid[0][i]

        for i in range(1,len(grid)):        #再遍历其他格子
            for j in range(1,len(grid[0])):
                dp[i][j]=max(dp[i-1][j],dp[i][j-1])+grid[i][j]

        return dp[-1][-1]
```

优化，增加左边和上边界

```
class Solution:
    def maxValue(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        dp = [[0] * (n+1) for _ in range(m+1)]
        for i in range(1, m+1):
            for j in range(1, n+1):
                dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]) + grid[i-1][j-1]
        return dp[-1][-1]
```

# 48、最长不含重复字符的子字符串

请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

> 输入: "abcabcbb"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

> 输入: "bbbbb"
> 输出: 1
> 解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

> 输入: "pwwkew"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
> 请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

**题解**

1、一次遍历，不断保存最长字符串，如果遇到重复字符，删除字符串中重复字符索引前的所有数

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        if len(s)==1:
            return 1
        a=[]
        x=0
        for i in s:
            if i in a:
                x=max(len(a),x)
                a=a[a.index(i)+1:]
                a.append(i)
            else:
                a.append(i)

        return max(len(a),x)
```

2、滑动窗口：从每一个字符开始的，不包含重复字符的最长子串，那么其中最长的那个字符串即为答案

- 我们使用两个指针表示字符串中的某个子串（的左右边界）。其中左指针代表着上文中「枚举子串的起始位置」，而右指针即为上文中的 r_k
- 在每一步的操作中，我们会将左指针向右移动一格，表示 我们开始枚举下一个字符作为起始位置，然后我们可以不断地向右移动右指针，但需要保证这两个指针对应的子串中没有重复的字符。在移动结束后，这个子串就对应着 以左指针开始的，不包含重复字符的最长子串。我们记录下这个子串的长度；
- 在枚举结束后，我们找到的最长的子串的长度即为答案。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # 哈希集合，记录每个字符是否出现过
        occ = set()
        n = len(s)
```

```
        # 右指针，初始值为 -1，相当于我们在字符串的左边界的左侧，还没有开始移动
        rk, ans = -1, 0
        for i in range(n):
            if i != 0:
                # 左指针向右移动一格，移除一个字符
                occ.remove(s[i - 1])
            while rk + 1 < n and s[rk + 1] not in occ:
                # 不断地移动右指针
                occ.add(s[rk + 1])
                rk += 1
            # 第 i 到 rk 个字符是一个极长的无重复字符子串
            ans = max(ans, rk - i + 1)
        return ans
```

3、字典记录每个字符最后出现的位置

1. 定义 longest, end分别记录最长子串的结果和上一次出现重复字符的最后一个位置
2. 如果新的字符在字典中，那么 end 需要更新为 max{end, dic[c]}。这是由于如果 dic[c] < end那就没必要更新。比如字符串 pwwp，在第二个 p 处， end = 1，而 dic[p] = 0， 如果 end 更新为 0，就会代表出现重复的 w，因此 end 取最大值更新。
3. 每次更新完 endend 和 longestlongest 值以后，再把当前字符的结尾位置更新到字典中

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        n = len(s)
        longest = 0
        end = -1
        dic = {}
        for i, c in enumerate(s):
            if c in dic:
                end = max(end, dic[c])
            longest = max(longest, i - end)
            dic[c] = i
        return longest
```

## 49、丑数

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

> 输入: n = 10
> 输出: 12
> 解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。

**题解**

1、动态规划

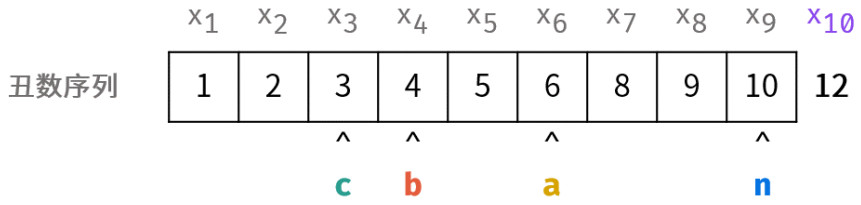
递推公式：$x_{n+1} = \min(x_a \times 2, x_b \times 3, x_c \times 5)$

三个索引满足：$\begin{cases} x_a \times 2 > x_n \geq x_{a-1} \times 2 \\ x_b \times 3 > x_n \geq x_{b-1} \times 3 \\ x_c \times 5 > x_n \geq x_{c-1} \times 5 \end{cases}$

以求 $x_{10}$ 为例：

此时三个索引应为 a = 6 , b = 4 , c = 3

则 $x_{10} = \min(x_6 \times 2, x_4 \times 3, x_3 \times 5) = 12$




```
class Solution:
    def nthUglyNumber(self, n: int) -> int:
        dp, a, b, c = [1] * n, 0, 0, 0
        for i in range(1, n):
            n2, n3, n5 = dp[a] * 2, dp[b] * 3, dp[c] * 5
            dp[i] = min(n2, n3, n5)
            if dp[i] == n2: a += 1
            if dp[i] == n3: b += 1
            if dp[i] == n5: c += 1
        return dp[-1]
```

2、小根堆

- 质数的线性筛算法的精髓在于保存下每一个合数的最小质因子, 只能乘以不超过最小质因子的质数, 去生成新的合数, 从而可以避免重复生成相同的合数.

```
class Solution:
    def nthUglyNumber(self, n: int) -> int:
        if n == 1:
            return 1
        H = [(2, 2), (3, 3), (5, 5)]
        heapify(H)
        for i in range(n - 1):
            num, p = heappop(H)
            heappush(H, (num * 2, 2))
            if p >= 3:
                heappush(H, (num * 3, 3))
                if p >= 5:
                    heappush(H, (num * 5, 5))
        #print(len(H)) #n = 1670时, H长度为162
        return num
```

# 50、第一个只出现一次的字符

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

> ```
> s = "abaccdeff"
> 返回 "b"
>
> s = ""
> 返回 " "
> ```

**题解**

1、字典保存遍历过的字符，然后字符和后面字符判断，一次遍历

```
class Solution:
    def firstUniqChar(self, s: str) -> str:
        if not s:return ' '
        dic={}
        for i in range(len(s)-1):
            if s[i] not in dic:
                dic[s[i]]=1
            else:
                continue
            if s[i] not in s[i+1:]:
                return s[i]

        return s[-1] if s[-1] not in dic else ' '
```

2、两次遍历，一次存字典，一次看是否为1

```
class Solution:
    def firstUniqChar(self, s: str) -> str:
        dic = {}
        for c in s:
            dic[c] = not c in dic
        for c in s:
            if dic[c]: return c
        return ' '
```

优化，遍历保存的字典

```
class Solution:
    def firstUniqChar(self, s: str) -> str:
        dic = {}
        for c in s:
            dic[c] = not c in dic
        for k, v in dic.items():
            if v: return k
        return ' '
```

# 51-60

## 51、数组中的逆序对

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

> 输入: [7,5,6,4]
> 输出: 5

**题解**

1、分治思想，归并排序

- 前面「分」的时候什么都不做，「合」的过程中计算「逆序对」的个数；
- 所有的「逆序对」来源于 3 个部分：
  1. 左边区间的逆序对；
  2. 右边区间的逆序对；
  3. 横跨两个区间的逆序对。
- 下面提供两种写法
  1. 在第 2 个子区间元素归并回去的时候，计算逆序对的个数

```
# 后有序数组中元素出列的时候，计算逆序个数
from typing import List
class Solution:
    def reversePairs(self, nums: List[int]) -> int:
        size = len(nums)
        if size < 2:
            return 0

        # 用于归并的辅助数组
        temp = [0 for _ in range(size)]
        return self.count_reverse_pairs(nums, 0, size - 1, temp)

    def count_reverse_pairs(self, nums, left, right, temp):
        # 在数组 nums 的区间 [left, right] 统计逆序对
        if left == right:
            return 0
        mid = (left + right) >> 1
        left_pairs = self.count_reverse_pairs(nums, left, mid, temp)
        right_pairs = self.count_reverse_pairs(nums, mid + 1, right, temp)

        reverse_pairs = left_pairs + right_pairs
        # 代码走到这里的时候，[left, mid] 和 [mid + 1, right] 已经完成了排序并且计算
好逆序对

        if nums[mid] <= nums[mid + 1]:
            # 此时不用计算横跨两个区间的逆序对，直接返回 reverse_pairs
            return reverse_pairs

        reverse_cross_pairs = self.merge_and_count(nums, left, mid, right,
temp)
        return reverse_pairs + reverse_cross_pairs
```

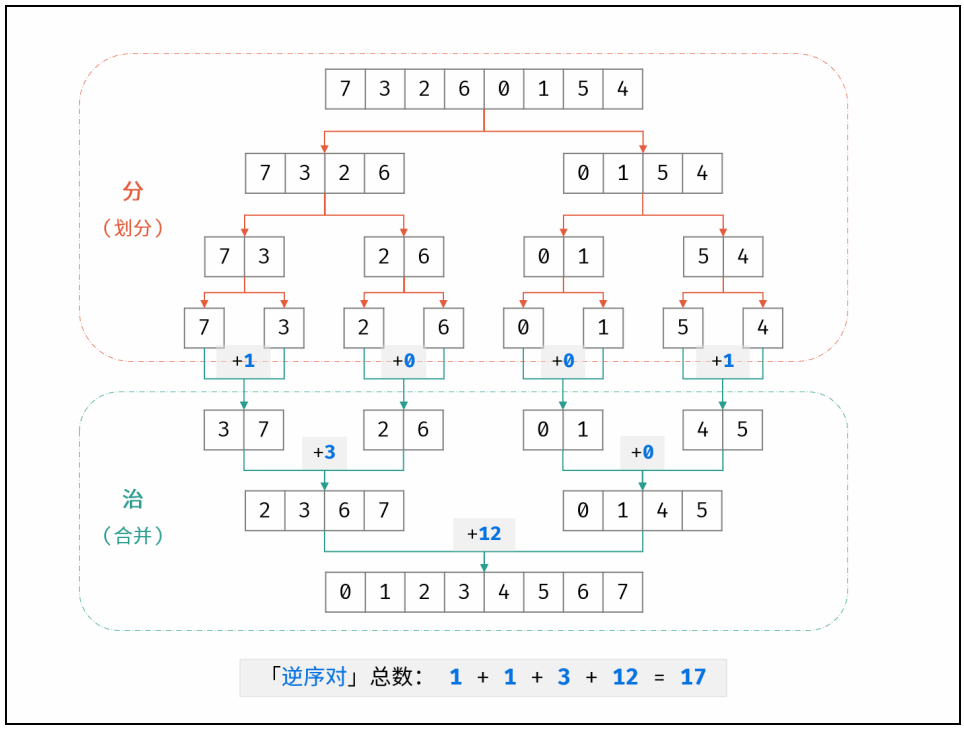
```
    def merge_and_count(self, nums, left, mid, right, temp):
        """
        [left, mid] 有序，[mid + 1, right] 有序
        前：[2, 3, 5, 8]，后：[4, 6, 7, 12]
        只在后面数组元素出列的时候，数一数前面这个数组还剩下多少个数字，
        由于"前"数组和"后"数组都有序，
        此时"前"数组剩下的元素个数 mid - i + 1 就是与"后"数组元素出列的这个元素构成的逆序对个数
        """
        for i in range(left, right + 1):
            temp[i] = nums[i]

        i = left
        j = mid + 1
        res = 0
        for k in range(left, right + 1):
            if i > mid:
                nums[k] = temp[j]
                j += 1
            elif j > right:
                nums[k] = temp[i]
                i += 1
            elif temp[i] <= temp[j]:
                # 此时前数组元素出列，不统计逆序对
                nums[k] = temp[i]
                i += 1
            else:
                # assert temp[i] > temp[j]
                # 此时后数组元素出列，统计逆序对，快就快在这里，一次可以统计出一个区间的个数的逆序对
                nums[k] = temp[j]
                j += 1
                # 例：[7, 8, 9][4, 6, 9]，4 与 7 以及 7 后面所有的数都构成逆序对
                res += (mid - i + 1)
        return res
```

2. 在第 1 个子区间元素归并回去的时候，计算逆序对的个数

```
  def merge_and_count(self, nums, left, mid, right, temp):
        """
        [left, mid] 有序，[mid + 1, right] 有序
        前：[2, 3, 5, 8]，后：[4, 6, 7, 12]
        我们只需要在后面数组元素出列的时候，数一数前面这个数组还剩下多少个数字，
        因为"前"数组和"后"数组都有序，
        因此，"前"数组剩下的元素个数 mid - i + 1 就是与"后"数组元素出列的这个元素构成的逆序对个数
        """
        for i in range(left, right + 1):
            temp[i] = nums[i]

        i = left
        j = mid + 1

        res = 0
        for k in range(left, right + 1):
            if i > mid:
```

```
                nums[k] = temp[j]
                j += 1
            elif j > right:
                nums[k] = temp[i]
                i += 1

                res += (right - mid)
            elif temp[i] <= temp[j]:
                nums[k] = temp[i]
                i += 1

                res += (j - mid - 1)
            else:
                assert temp[i] > temp[j]
                nums[k] = temp[j]
                j += 1
        return res
```

2、归并排序




1. 终止条件： 当l≥r 时，代表子数组长度为 1 ， 此时终止划分；
2. 递归划分： 计算数组中点 m ， 递归划分左子数组 merge_sort(l, m) 和右子数组 merge_sort(m + 1, r) ；
3. 合并与逆序对统计：
    1. 暂存数组 nums 闭区间 [i, r]内的元素至辅助数组 tmp ；
    2. 循环合并： 设置双指针 i , j 分别指向左 / 右子数组的首元素；
        - 当 i = m + 1 时： 代表左子数组已合并完，因此添加右子数组当前元素 tmp[j] ， 并执行 j = j + 1 ；
        - 否则，当 j = r + 1 时： 代表右子数组已合并完，因此添加左子数组当前元素 tmp[i] ， 并执行 i = i + 1
        - 否则，当 tmp[i]≤tmp[j] 时： 添加左子数组当前元素 tmp[i]，并执行 i = i + 1 ；
        - 否则（即 tmp[i] > tmp[j]）时： 添加右子数组当前元素 tmp[j] ， 并执行 j = j + 1 ； 此时构成 m - i + 1 个「逆序对」， 统计添加至 res ；
4. 返回值： 返回直至目前的逆序对总数res ；

```python
class Solution:
    def reversePairs(self, nums: List[int]) -> int:
        def merge_sort(l, r):
            # 终止条件
            if l >= r: return 0
            # 递归划分
            m = (l + r) // 2
            res = merge_sort(l, m) + merge_sort(m + 1, r)
            # 合并阶段
            i, j = l, m + 1
            tmp[l:r + 1] = nums[l:r + 1]
            for k in range(l, r + 1):
                if i == m + 1:
                    nums[k] = tmp[j]
                    j += 1
                elif j == r + 1 or tmp[i] <= tmp[j]:
                    nums[k] = tmp[i]
                    i += 1
                else:
                    nums[k] = tmp[j]
                    j += 1
                    res += m - i + 1 # 统计逆序对
            return res

        tmp = [0] * len(nums)
        return merge_sort(0, len(nums) - 1)
```

3、树状数组

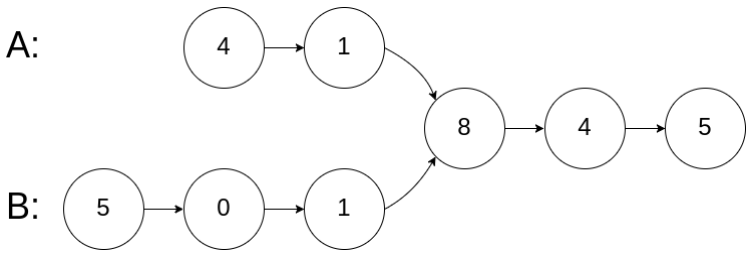
- 先离散化，将所有的数组元素映射到 0、1、2、3... ，这是为了节约树状数组的空间；
- 从后向前扫描，边统计边往树状数组里面添加元素，这个过程是「动态的」，需要动手计算才能明白思想。

```python
from typing import List
class Solution:
    def reversePairs(self, nums: List[int]) -> int:
        class FenwickTree:
            def __init__(self, n):
                self.size = n
                self.tree = [0 for _ in range(n + 1)]

            def __lowbit(self, index):
                return index & (-index)

            # 单点更新：从下到上，最多到 len，可以取等
            def update(self, index, delta):
                while index <= self.size:
                    self.tree[index] += delta
                    index += self.__lowbit(index)

            # 区间查询：从上到下，最少到 1，可以取等
            def query(self, index):
                res = 0
                while index > 0:
                    res += self.tree[index]
                    index -= self.__lowbit(index)
                return res
```

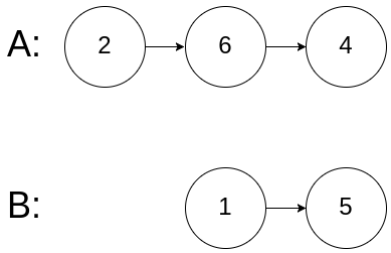
```
        # 特判
        size = len(nums)
        if size < 2:
            return 0

        # 原始数组去除重复以后从小到大排序，这一步叫做离散化
        s = list(set(nums))

        # 构建最小堆，因为从小到大一个一个拿出来，用堆比较合适
        import heapq
        heapq.heapify(s)

        # 由数字查排名
        rank_map = dict()
        rank = 1
        # 不重复数字的个数
        rank_map_size = len(s)
        for _ in range(rank_map_size):
            num = heapq.heappop(s)
            rank_map[num] = rank
            rank += 1

        res = 0
        # 树状数组只要不重复数字个数这么多空间就够了
        ft = FenwickTree(rank_map_size)

        # 从后向前看，拿出一个数字来，就更新一下，然后向前查询比它小的个数
        for i in range(size - 1, -1, -1):
            rank = rank_map[nums[i]]
            ft.update(rank, 1)
            res += ft.query(rank - 1)
        return res
```

## 52、两个链表的第一个公共节点

输入两个链表，找出它们的第一个公共节点。




```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```




```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。
```

---

**题解**

1、我们通常做这种题的思路是设定两个指针分别指向两个链表头部，一起向前走直到其中一个到达末端，另一个与末端距离则是两链表的 长度差。再通过长链表指针先走的方式消除长度差，最终两链表即可同时走到相交点

```
class Solution(object):
    def getIntersectionNode(self, headA, headB):
        ha, hb = headA, headB       #ha，hb代表指针
        while ha != hb:
            ha = ha.next if ha else headB
            hb = hb.next if hb else headA
        return ha
```

2、存链表节点进行查询

1. 遍历链表A，并将其节点存入集合
2. 遍历B的每个节点并在集合中进行查询，一旦遍历过程中查询到相同节点，即说明有交点，否则无

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        A = set()
        cur1 = headA
        cur2 = headB
        while cur1:
            A.add(cur1)
            cur1 = cur1.next
        while cur2:
            if cur2 in A:
                return cur2
            cur2 = cur2.next
        return None
```

3、算出链表距离差挨个对比，分别遍历两个链表，并记录两链表的长度差n，将出现两种情况，

1. 让长链表先走n步
2. 再同时开始走，并对比两个链表的当前节点，节点相等时即为交点
3. 若没有交点，则在最后的Null处相交

```
class Solution:
```

```
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) ->
ListNode:
        cur1 = headA
        cur2 = headB
        n = 0
        while cur1:
            n += 1
            cur1 = cur1.next
        while cur2:
            n -= 1
            cur2 = cur2.next
        if cur1 != cur2:
            return None
        cur1 = headA if n > 0 else headB
        cur2 = headB if cur1 == headA else headA
        n = abs(n)
        while n:
            n -= 1
            cur1 = cur1.next
        while cur1 != cur2:
            cur1 = cur1.next
            cur2 = cur2.next
        return cur1
```

## 53-Ⅰ、在排序数组中查找数字

统计一个数字在排序数组中出现的次数。

> 输入: nums = [5,7,7,8,8,10], target = 8
> 输出: 2

> 输入: nums = [5,7,7,8,8,10], target = 6
> 输出: 0

**题解**

1、一次遍历，相等+1

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        res=0
        for i in nums:
            if i==target:
                res+=1

        return res
```

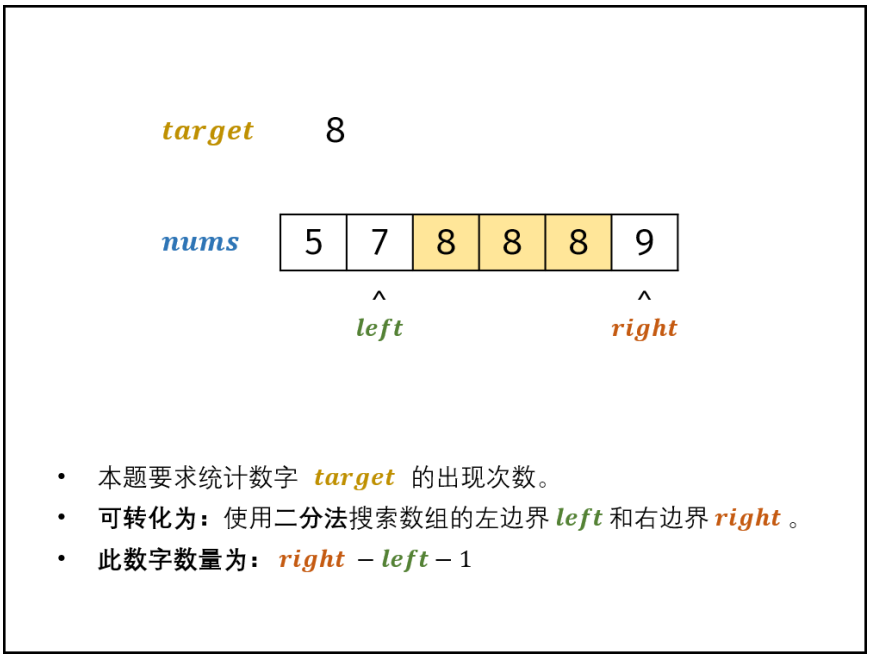
2、创建计数字典

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        dic=collections.Counter(nums)
        return dic[target]
```

3、获取前后索引，使用try检测异常

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left,right=-1,-1
        try:
            left=nums.index(target)
            right=len(nums)-nums[::-1].index(target)-1
        except:
            return 0

        return right-left+1
```

4、二分查找




1. 初始化： 左边界 i = 0，右边界 j = len(nums) - 1 。
2. 循环二分： 当闭区间 [i, j][i,j] 无元素时跳出；
    1. 计算中点 m = (i + j) / 2 （向下取整）；
    2. 若 nums[m] < target ， 则 target 在闭区间 [m + 1, j] 中，因此执行 i = m + 1；
    3. 若 nums[m] > target ， 则 target 在闭区间 [i, m - 1]中，因此执行 j = m - 1；
    4. 若 nums[m] = target，则右边界 right 在闭区间 [m+1, j] 中；左边界 left 在闭区间 [i, m-1] 中。因此分为以下两种情况：
        1. 若查找 右边界 right ， 则执行 i = m + 1 ； （跳出时 i 指向右边界）
        2. 若查找 左边界 left ， 则执行 j = m - 1 ； （跳出时 j 指向左边界）
3. 返回值： 应用两次二分，分别查找 right 和 left ， 最终返回 right - left - 1 即可。

```python
class Solution:
    def search(self, nums: [int], target: int) -> int:
        # 搜索右边界 right
```

```
        i, j = 0, len(nums) - 1
        while i <= j:
            m = (i + j) // 2
            if nums[m] <= target: i = m + 1
            else: j = m - 1
        right = i
        # 若数组中无 target ，则提前返回
        if j >= 0 and nums[j] != target: return 0
        # 搜索左边界 left
        i = 0
        while i <= j:
            m = (i + j) // 2
            if nums[m] < target: i = m + 1
            else: j = m - 1
        left = j
        return right - left - 1
```

优化：可将二分查找右边界right的代码封装

```
class Solution:
    def search(self, nums: [int], target: int) -> int:
        def helper(tar):
            i, j = 0, len(nums) - 1
            while i <= j:
                m = (i + j) // 2
                if nums[m] <= tar: i = m + 1
                else: j = m - 1
            return i
        return helper(target) - helper(target - 1)
```

## 53-II、0~n-1中缺失的数字

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

> 输入: [0,1,3]
> 输出: 2

> 输入: [0,1,2,3,4,5,6,7,9]
> 输出: 8

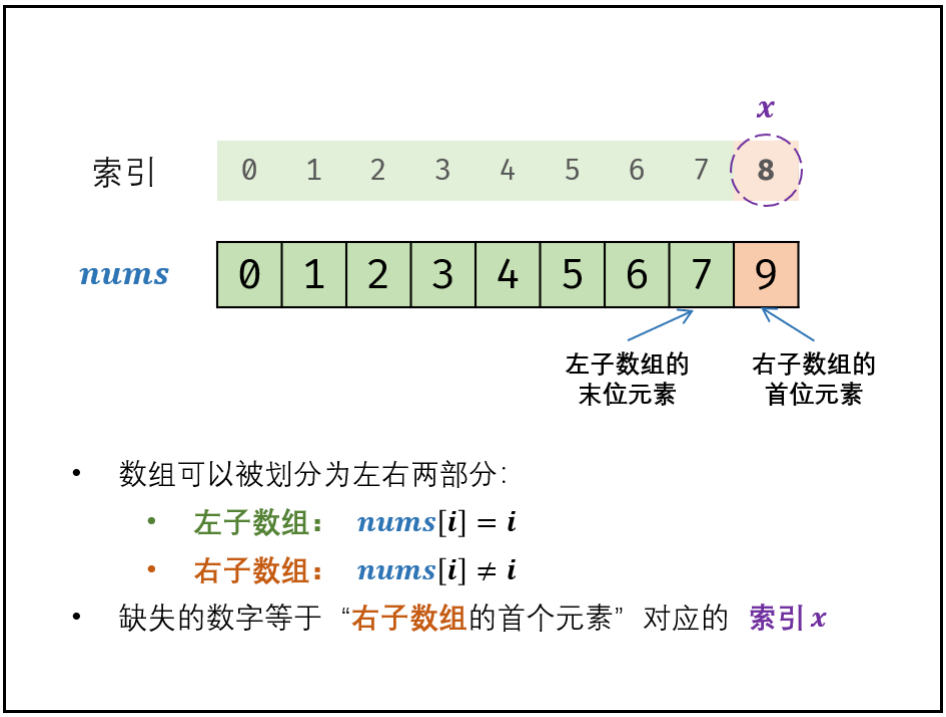
**题解**

1、数学公式，

- 高斯求和，求出1-len(nums)的和
- 减去nums数组的和，

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        return (1+len(nums))*len(nums)//2-sum(nums)
```

2、二分查找




- 数组可以被划分为左右两部分：
  - 左子数组： $nums[i] = i$
  - 右子数组： $nums[i] \neq i$
- 缺失的数字等于 “右子数组的首个元素” 对应的 索引 $x$

1. 初始化： 左边界 i = 0 ， 右边界 j = len(nums) - 1 ； 代表闭区间 [i, j]。
2. 循环二分： 当 i≤j 时循环 （即当闭区间 [i, j] 为空时跳出） ；
   1. 计算中点 m = (i + j) // 2 ， 其中 "//" 为向下取整除法；
   2. 若 nums[m] = m ， 则 “右子数组的首位元素” 一定在闭区间 [m + 1, j]中，因此执行 i = m + 1；
   3. 若 nums[m] /=m，则 “左子数组的末位元素” 一定在闭区间 [i, m - 1] 中，因此执行 j = m - 1；
3. 返回值： 跳出时，变量 i 和 j 分别指向 “右子数组的首位元素” 和 “左子数组的末位元素” 。因此返回 i 即可。

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        i, j = 0, len(nums) - 1
        while i <= j:
            m = (i + j) // 2
            if nums[m] == m: i = m + 1
            else: j = m - 1
        return i
```

## 54、二叉搜索树的第K大的节点

给定一棵二叉搜索树，请找出其中第k大的节点。

```
输入: root = [3,1,4,null,2], k = 1
3
/ \
1   4
\
2
输出: 4
```

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
    5
   / \
  3   6
 / \
2   4
/
1
输出: 4
```

---

**题解**

1、中序遍历，保存到数组，输出倒数第K个节点

```
class Solution:
    def kthLargest(self, root: TreeNode, k: int) -> int:
        res=[]
        def dfs(root):
            if not root:return
            if root.left:dfs(root.left)
            res.append(root.val)
            if root.right:dfs(root.right)

        dfs(root)
        return res[-k]
```

优化:

```
class Solution:
    def kthLargest(self, root: TreeNode, k: int) -> int:
        def helper(root):
            return helper(root.left) + [root.val] + helper(root.right) if root
else []
        return helper(root)[-k]
```

2、中序遍历倒序，提前结束

1. 终止条件： 当节点 root 为空（越过叶节点），则直接返回；
2. 递归右子树： 即 dfs(root.right)；
3. 三项工作：
    1. 提前返回： 若 k = 0，代表已找到目标节点，无需继续遍历，因此直接返回；
    2. 统计序号： 执行 k = k - 1 （即从 k 减至 0 ）；
    3. 记录结果： 若 k = 0，代表当前节点为第 k 大的节点，因此记录 res = root.val；
       递归左子树： 即 dfs(root.left)；

```python
class Solution:
    def kthLargest(self, root: TreeNode, k: int) -> int:
        def dfs(root):
            if not root: return
            dfs(root.right)
            if self.k == 0: return
            self.k -= 1
            if self.k == 0: self.res = root.val
            dfs(root.left)

        self.k = k
        dfs(root)
        return self.res
```

## 55-Ⅰ、二叉树的深度

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

> ```
>  3
> / \
> 9  20
>  /  \
> 15   7
> 返回它的最大深度 3 。
> ```

**题解**

1、dfs，保存一个最深深度

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        self.res=0
        def dfs(root,deep):
            if not root:return
            self.res=max(self.res,deep)
            dfs(root.left,deep+1)
            dfs(root.right,deep+1)

        dfs(root,1)
        return self.res
```

优化：根节点的最大深度=子节点的最大深度+1，不断嵌套，

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root:
            return 0
        max_left=self.maxDepth(root.left)
        max_right=self.maxDepth(root.right)
        return max(max_left,max_right)+1
```

2、BFS，一层一层搜索，先搜索距离近的，从上到下遍历一次

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        # BFS
        if root is None:
            return 0
        queue = [(1, root)]
        while queue:
            depth, node = queue.pop(0)      #先进先出，一层一层遍历
            if node.left:
                queue.append((depth+1,node.left))
            if node.right:
                queue.append((depth+1,node.right))
        return depth
```

## 55-II、平衡二叉树

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

```
 3
/ \
9  20
 /  \
15   7
True
```

```
    1
   / \
  2   2
 / \
3   3
/ \
4   4
False
```

**题解**

1、同上，得到左右子树的深度，比较深度差，大于1改变标志位

```python
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        self.flag=True
        def dfs(root):
            if not root:return 0
            left=dfs(root.left)
            right=dfs(root.right)
            if left-right<-1 or left-right>1:
                self.flag=False
            return max(left,right)+1

        dfs(root)
        return self.flag
```

优化：少一个标志位，如果不是平衡树，返回深度-1

```python
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def recur(root):
            if not root: return 0
            left = recur(root.left)
            if left == -1: return -1
            right = recur(root.right)
            if right == -1: return -1
            return max(left, right) + 1 if abs(left - right) <= 1 else -1

        return recur(root) != -1
```

2、先序遍历，判断深度

- 判断当前子树是否是平衡树
- 判断当前子树左子树是平衡树
- 判断当前子树右子树是平衡树

```python
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        if not root: return True
        return abs(self.depth(root.left) - self.depth(root.right)) <= 1 and \
            self.isBalanced(root.left) and self.isBalanced(root.right)

    def depth(self, root):
        if not root: return 0
        return max(self.depth(root.left), self.depth(root.right)) + 1
```

## 56-Ⅰ、数组中数字出现的次数

个整型数组 `nums` 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

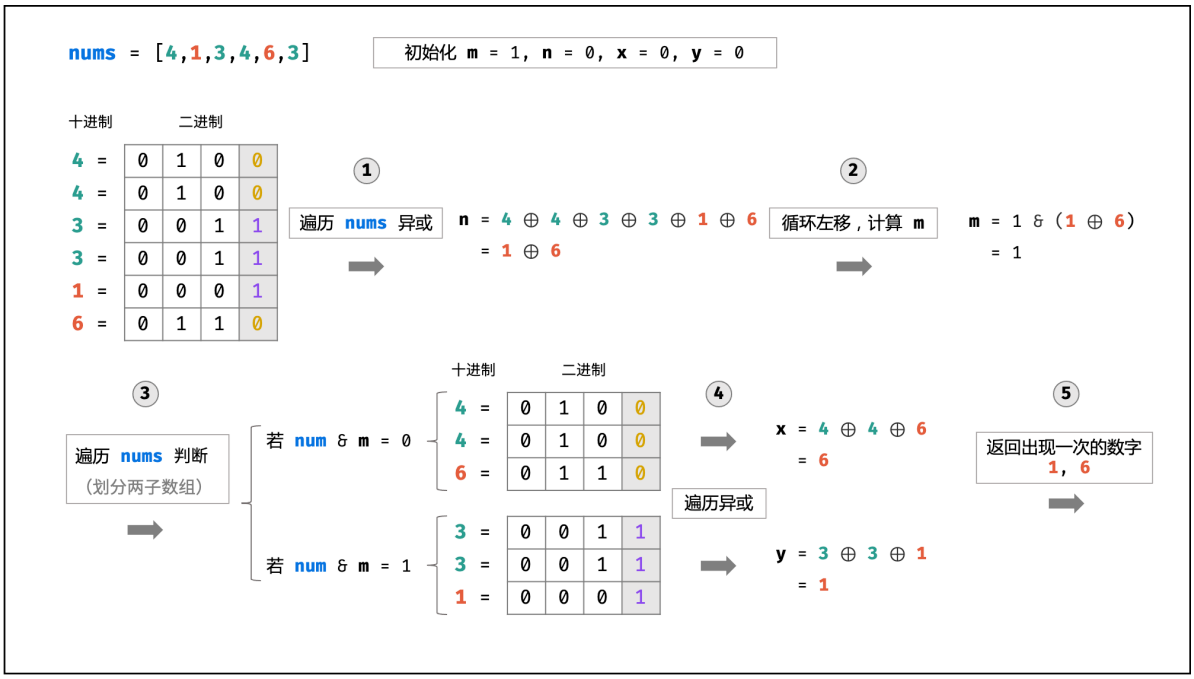
> ```
> 输入：nums = [4,1,4,6]
> 输出：[1,6] 或 [6,1]
> ```

```
输入：nums = [1,2,10,4,1,4,3,3]
输出：[2,10] 或 [10,2]
```

---

**题解**

1、位运算




- 第一次遍历，全员异或，得到答案的两个数的异或值
- 循环左移计算m，获取整数x⊕y 首位 1 ，记录于 m 中
- 通过遍历判断 nums 中各数字和 m做与运算的结果，可将数组拆分为两个子数组，并分别对两个子数组遍历求异或，则可得到两个只出现一次的数字，

```python
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        x, y, n, m = 0, 0, 0, 1
        for num in nums:          # 1. 遍历异或
            n ^= num
        while n & m == 0:         # 2. 循环左移，计算 m
            m <<= 1
        for num in nums:          # 3. 遍历 nums 分组
            if num & m: x ^= num # 4. 当 num & m != 0
            else: y ^= num        # 4. 当 num & m == 0
        return x, y               # 5. 返回出现一次的数字
```

## 56-II、数组中数字出现的次数II

---

在一个数组 `nums` 中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

```
输入：nums = [3,4,3,3]
输出：4
```

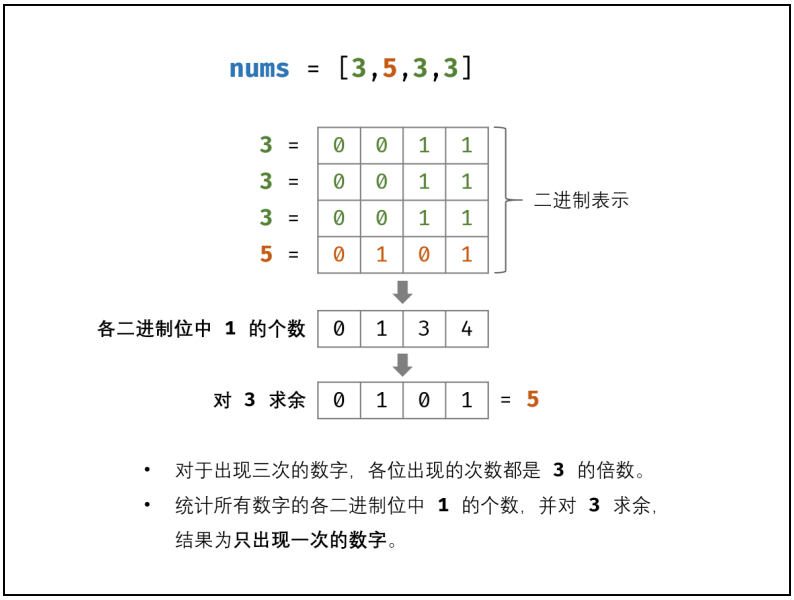
> 输入：nums = [9,1,7,9,7,9,7]
> 输出：1

---

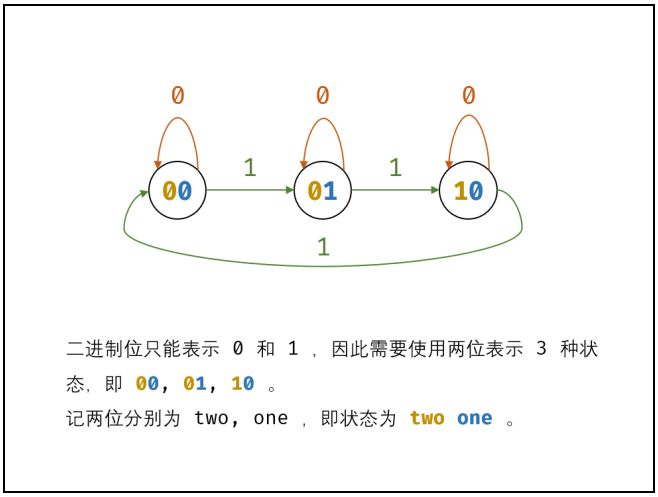
**题解**

1、位运算

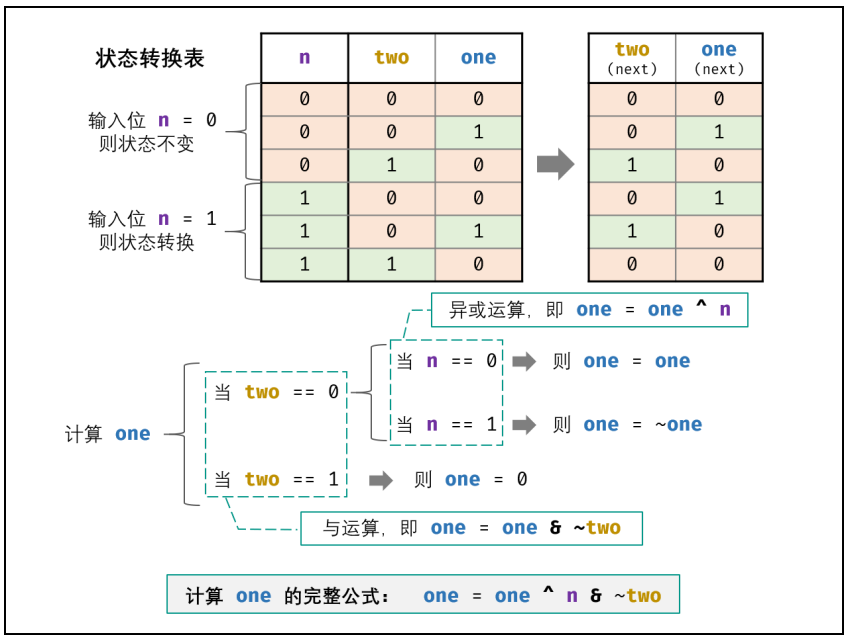
- 考虑数字的二进制形式，对于出现三次的数字，各 二进制位 出现的次数都是 33 的倍数。因此，统计所有数字的各二进制位中 11 的出现次数，并对 33 求余，结果则为只出现一次的数字。




状态转移




- 计算one




```
if two == 0:
  if n == 0:
    one = one
  if n == 1:
    one = ~one
if two == 1:
    one = 0
```

优化：引入异或运算

```
if two == 0:
    one = one ^ n
if two == 1:
    one = 0
```

优化：引入与运算

```
one = one ^ n & ~two
```

- 计算two

```
two = two ^ n & ~one
```

- 遍历完所有数字后，各二进制位都处于状态 00 和状态 01 （取决于 “只出现一次的数字” 的各二进制位是 1 还是 0 ），而此两状态是由 one 来记录的（此两状态下 twos 恒为 0 ），因此返回 ones 即可

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        ones, twos = 0, 0
        for num in nums:
            ones = ones ^ num & ~twos
            twos = twos ^ num & ~ones
        return ones
```

2、字典循环

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        dic = {}
        for num in nums:
            if num not in dic:
                dic[num] = 1
            else:
                dic[num] += 1
        for key, value in dic.items():#以列表返回可遍历的(键, 值) 元组数组
            if value == 1:
                return key
```

3、数学方法

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        return (3*sum(list(set(nums)))-sum(nums))//2
```

## 57、和为s的两个数字

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

> 输入：nums = [2,7,11,15], target = 9
> 输出：[2,7] 或者 [7,2]

> 输入：nums = [10,26,30,31,47,60], target = 40
> 输出：[10,30] 或者 [30,10]

**题解**

1、双指针，向内收缩

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        left,right=0,len(nums)-1
        while left<right:
            if nums[left]+nums[right]==target:
                return [nums[left],nums[right]]
            elif nums[left]+nums[right]<target:
                left+=1
            else:
                right-=1
```

# 57-II、和为s的连续正数序列

输入一个正整数 target ，输出所有和为 target 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

> 输入：target = 9
> 输出：[[2,3,4],[4,5]]

> 输入：target = 15
> 输出：[[1,2,3,4,5],[4,5,6],[7,8]]

**题解**

1、滑动窗口

- i表示窗口长度，从2开始，到target//2
- 使用高斯求和，如果等于，保存到res
- 如果大于则窗口左滑，小于则跳出循环

超时间

```
class Solution:
    def findContinuousSequence(self, target: int) -> List[List[int]]:
        res=[]
        for i in range(target//2,1,-1):
            n=target//2+1
            while n-i>=0:
                if (n*2-i+1)*i/2==target:
                    res.append([j for j in range(n-i+1,n+1)])
                    break
                elif (n*2-i+1)*i/2>target:
                    n-=1
                else:
                    break

        return res
```

2、求和公式

- target=(i+j)*(j-i+1)/2
- 当确定 元素和 target 与 左边界 i时，可通过 解一元二次方程 ，直接计算出右边界 j
- `j=(-1+√1+4(2*target+i^2-i))/2`
- 通过从小到大遍历左边界 i*i* 来计算 以 i为起始数字的连续正整数序列

```
class Solution:
    def findContinuousSequence(self, target: int):
        i, j, res = 1, 2, []
        while i < j:
            j = (-1 + (1 + 4 * (2 * target + i * i - i)) ** 0.5) / 2
            if i < j and j == int(j):
                res.append(list(range(i, int(j) + 1)))
            i += 1
        return res
```

2、滑动窗口，双指针

1. 初始化： 左边界 i = 1 ，右边界 j = 2，元素和 s = 3，结果列表 res ；
2. 循环： 当 i≥j 时跳出；
    1. 当 s > target 时： 向右移动左边界 i = i + 1，并更新元素和 s ；
    2. 当 s < target 时： 向右移动右边界 j = j + 1 ，并更新元素和 s ；
    3. 当 s = target 时： 记录连续整数序列，并向右移动左边界 i = i + 1 ；
3. 返回值： 返回结果列表 res ；

```
class Solution:
    def findContinuousSequence(self, target: int) -> List[List[int]]:
        i, j, s, res = 1, 2, 3, []
        while i < j:
            if s == target:
                res.append(list(range(i, j + 1)))
            if s >= target:
                s -= i
                i += 1
            else:
                j += 1
                s += j
        return res
```

3、间隔法

- target=(x+y)*(y-x+1)/2
- 引入间隔，这个间隔就是末项y减去首项x，也就是首项和末项隔开多远的意思，可令i=间隔
- `x=(t-i*(i+1)/2)/(t_1)`:两个条件
    - i*(i+1)/2要小于t，x不能出现负数
    - x必须为整数，

```
class Solution:
    def findContinuousSequence(self, target: int) -> List[List[int]]:
        # 我们的间隔从1开始
        i, res = 1, []

        # 根据上面的条件1，限定i的大小，即间隔的范围
        while i*(i+1)/2 < target:
            # 根据条件2，如果x不为整数则扩大间隔
            if not (target - i*(i+1)/2) % (i+1):
                # 如果两个条件都满足，代入公式求出x即可，地板除//会把数改成float形式，用
int()改回来
                x = int((target - i*(i+1)/2) // (i+1))
                # 反推出y，将列表填入输出列表即可
```

```
                res.append(list(range(x,x+i+1)))
            # 当前间隔判断完毕，检查下一个间隔
            i += 1

        # 由于间隔是从小到大，意味着[x,y]列表是从大到小的顺序放入输出列表res的，所以反转之
        return res[::-1]
```

## 58-Ⅰ、翻转单词顺序

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. "，则输出"student. a am I"。

> 输入: "the sky is blue"
> 输出: "blue is sky the"
>
> 输入: "  hello world!  "
> 输出: "world! hello"
> 解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
>
> 输入: "a good   example"
> 输出: "example good a"
> 解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

**题解**

1、split()函数，将s转成单词列表

```
class Solution:
    def reverseWords(self, s: str) -> str:
        res=s.split()
        return ' '.join(res[::-1])
```

2、双指针，遍历s，保存单词

```
class Solution:
    def reverseWords(self, s: str) -> str:
        s = s.strip() # 删除首尾空格
        i = j = len(s) - 1
        res = []
        while i >= 0:
            while i >= 0 and s[i] != ' ': i -= 1 # 搜索首个空格
            res.append(s[i + 1: j + 1]) # 添加单词
            while s[i] == ' ': i -= 1 # 跳过单词间空格
            j = i # j 指向下个单词的尾字符
        return ' '.join(res) # 拼接并返回
```

## 58-II、左旋转字符串

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

> ```
> 输入: s = "abcdefg", k = 2
> 输出: "cdefgab"
> ```
>
> ```
> 输入: s = "lrloseumgh", k = 6
> 输出: "umghlrlose"
> ```

**题解**

1、切片操作

```
class Solution:
    def reverseLeftWords(self, s: str, n: int) -> str:
        return s[n:]+s[:n]
```

2、字符串拼接

```
class Solution:
    def reverseLeftWords(self, s: str, n: int) -> str:
        res = []
        for i in range(n, len(s)):
            res.append(s[i])
        for i in range(n):
            res.append(s[i])
        return ''.join(res)
```

## 59-Ⅰ、滑动窗口的最大值

给定一个数组 `nums` 和滑动窗口的大小 `k`，请找出所有滑动窗口里的最大值。

> ```
> 输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
> 输出: [3,3,5,5,6,7]
> 解释:
>
> 滑动窗口的位置                最大值
> ---------------               -----
> [1  3  -1] -3  5  3  6  7       3
> 1 [3  -1  -3] 5  3  6  7       3
> 1  3 [-1  -3  5] 3  6  7       5
> 1  3  -1 [-3  5  3] 6  7       5
> 1  3  -1  -3 [5  3  6] 7       6
> 1  3  -1  -3  5 [3  6  7]      7
> ```

**题解**

1、遍历，max函数

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums:return []
        res=[]
        for i in range(len(nums)-k+1):
            res.append(max(nums[i:i+k]))

        return res
```

2、单调队列


```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums or k == 0: return []
        deque = collections.deque()
        # 未形成窗口
        for i in range(k):
            while deque and deque[-1] < nums[i]:
                deque.pop()
            deque.append(nums[i])
        res = [deque[0]]
        # 形成窗口后
        for i in range(k, len(nums)):
            if deque[0] == nums[i - k]:
                deque.popleft()
            while deque and deque[-1] < nums[i]:
                deque.pop()
            deque.append(nums[i])
            res.append(deque[0])
        return res
```

# 59-II、队列的最大值

请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

> 输入:
> ["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
> [[],[1],[2],[],[],[]]
> 输出: [null,null,null,2,1,2]

> 输入:
> ["MaxQueue","pop_front","max_value"]
> [[],[],[]]
> 输出: [null,-1,-1]

**题解**

1、单调双向队列

- 最大值 max_value()：
  - 当双向队列 deque 为空，则返回 −1 ；
  - 否则，返回 deque 首元素；
- 入队 push_back()：
  - 将元素 value 入队 queue ；
  - 将双向队列中队尾 所有 小于 value 的元素弹出（以保持 deque 非单调递减），并将元素 value 入队 deque ；
- 出队 pop_front()：
  - 若队列 queue 为空，则直接返回 −1 ；
  - 否则，将 queue 首元素出队；
  - 若 deque 首元素和 queue 首元素 相等 ， 则将 deque 首元素出队（以保持两队列 元素一致 ） ；

```python
import queue

class MaxQueue:
    def __init__(self):
        self.queue = queue.Queue()
        self.deque = queue.deque()

    def max_value(self) -> int:
        return self.deque[0] if self.deque else -1

    def push_back(self, value: int) -> None:
        self.queue.put(value)
        while self.deque and self.deque[-1] < value:
            self.deque.pop()
        self.deque.append(value)

    def pop_front(self) -> int:
        if self.queue.empty(): return -1
```

```
        val = self.queue.get()
        if val == self.deque[0]:
            self.deque.popleft()
        return val
```

## 60、n个骰子的点数

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

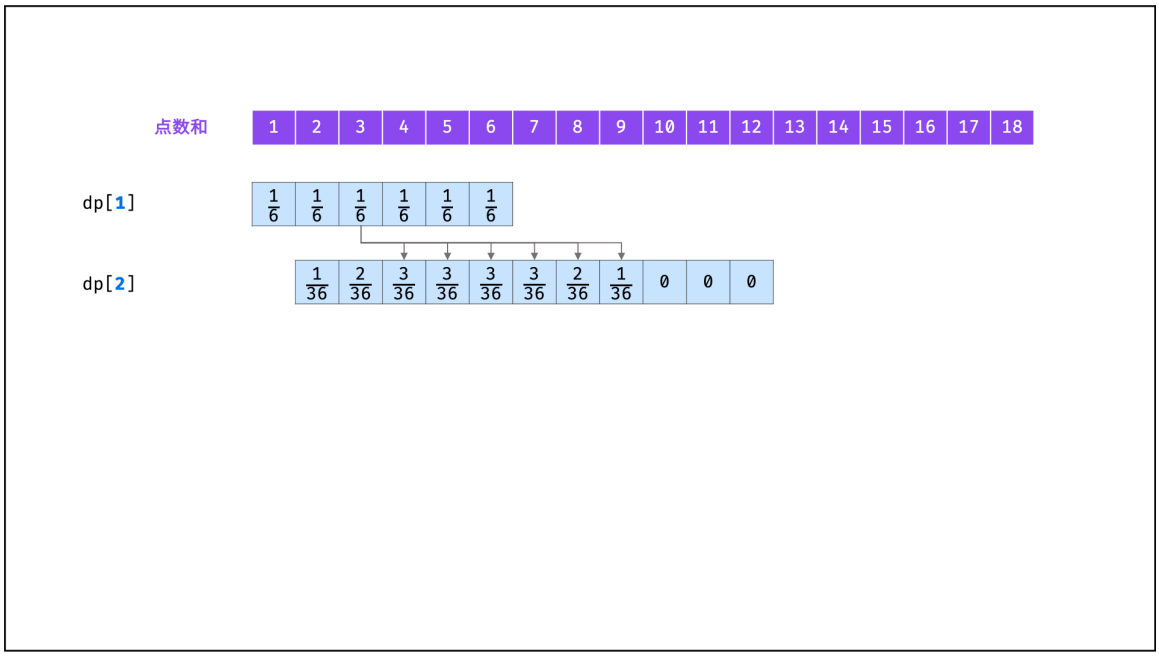
你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

> 输入: 1
> 输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]

> 输入: 2
> 输出:
> [0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.02778]

**题解**

1、动态规划



```python
class Solution:
    def dicesProbability(self, n: int) -> List[float]:
        dp = [1 / 6] * 6
        for i in range(2, n + 1):
            tmp = [0] * (5 * i + 1)
            for j in range(len(dp)):
                for k in range(6):
                    tmp[j + k] += dp[j] / 6
            dp = tmp
        return dp
```

# 61-68

## 61、扑克牌中的顺子

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A 不能视为 14。

> 输入: [1,2,3,4,5]
> 输出: True

> 输入: [0,0,1,2,5]
> 输出: True

**题解**

1、0是标志位，遍历一次，判断相差是否大于0的个数

```python
class Solution:
    def isStraight(self, nums: List[int]) -> bool:
        zero=0
        nums.sort()
        for i in range(len(nums)-1):
            if nums[i]==0:
                zero+=1
                continue
            if nums[i]==nums[i+1]:
                return False
            if (nums[i]+1) != nums[i+1]:
                k=nums[i+1]-nums[i]-1
                zero-=k

        return True if zero>=0 else False
```

优化:

```python
class Solution:
    def isStraight(self, nums: List[int]) -> bool:
        joker = 0
        nums.sort() # 数组排序
        for i in range(4):
            if nums[i] == 0: joker += 1 # 统计大小王数量
            elif nums[i] == nums[i + 1]: return False # 若有重复，提前返回 false
        return nums[4] - nums[joker] < 5 # 最大牌 - 最小牌 < 5 则可构成顺子
```

2、集合set，True的充分条件是

- 除大小王外，所有牌 无重复；
- 设此 55 张牌中最大的牌为 max，最小的牌为 min（大小王除外），则需满足：max-min<5

```python
class Solution:
    def isStraight(self, nums: List[int]) -> bool:
        repeat = set()
        ma, mi = 0, 14
        for num in nums:
            if num == 0: continue # 跳过大小王
            ma = max(ma, num) # 最大牌
            mi = min(mi, num) # 最小牌
            if num in repeat: return False # 若有重复，提前返回 false
            repeat.add(num) # 添加牌至 Set
        return ma - mi < 5 # 最大牌 - 最小牌 < 5 则可构成顺子
```

优化：

```python
class Solution:
    def isStraight(self, nums: List[int]) -> bool:
        nums = [num for num in nums if num] #筛选非0值
        return len(set(nums)) == len(nums) and max(nums)-min(nums) < 5 #判断非0值是否有重复及跨度区间
```

## 62、圆圈中最后剩下的数字

0,1,···,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

> 输入: n = 5, m = 3
> 输出: 3

> 输入: n = 10, m = 17
> 输出: 2

**题解**

1、主要在于找到索引，每次循环删除索引处数据，直到数组长度为1——超时间

```python
class Solution:
    def lastRemaining(self, n: int, m: int) -> int:
        nums=list(range(n))
        index=0
        while len(nums)>1:
            index=(index+m-1)%len(nums)
            nums.pop(index)

        return nums[0]
```

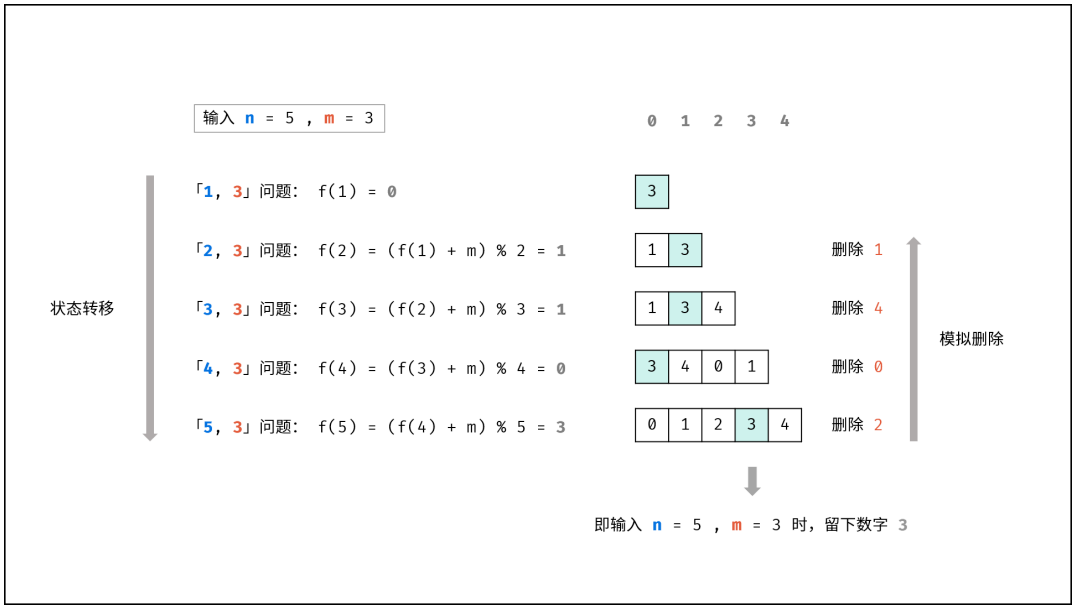
2、动态规划：约瑟夫环问题，输入n，m，设解（即最后留下的数字）为 f(n)，则有

- 「n,m 问题」：数字环为 0, 1, 2, ..., n - 1，解为 f(n)；
- 「n-1, m 问题」：数字环为 0, 1, 2, ..., n - 20,1,2,...,n−2，解为 f(n-1)；
- 以此类推.....
- 由于有可能 m > n，因此删除的数字为 (m−1)%n，删除后的数字环从下个数字（即 m % n）开始，设 t=m%n，可得数字环：
- ```
  [n-1,m]问题    ->      [n,m]问题删除后
      0        ->          t+0
      1        ->          t+1
      ...      ->          ...
      n-2      ->          t-2
  ```

  - $f(n)=(f(n-1)+t)\%n$

    $=(f(n-1)+m\%n)\%n$

    $=(f(n-1)+m)\%n$




```python
class Solution:
    def lastRemaining(self, n: int, m: int) -> int:
        x = 0
        for i in range(2, n + 1):
            x = (x + m) % i
        return x
```

3、迭代写法
```

```
sys.setrecursionlimit(100000)

class Solution:
    def lastRemaining(self, n: int, m: int) -> int:
        return self.f(n, m)

    def f(self, n, m):
        if n == 0:
            return 0
        x = self.f(n - 1, m)
        return (m + x) % n
```

## 63、股票的最大利润

假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少?

> 输入: [7,1,5,3,6,4]
> 输出: 5
> 解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。

> 输入: [7,6,4,3,1]
> 输出: 0
> 解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

**题解**

1、一次遍历；最小值更新；最大收益更新；

```
class Solution:
    def maxProfit(self, prices):
        res = 0
        minValue = float("inf")
        for i in range(len(prices)):
            if prices[i] < minValue:        #更新最小值
                minValue = prices[i]
            if prices[i] - minValue > res:  #更新最大收益
                res = prices[i] - minValue
        return res
```

另一种写法

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        minprice = float('inf')
        maxprofit = 0
        for price in prices:
            minprice = min(minprice, price)
            maxprofit = max(maxprofit, price - minprice)
        return maxprofit
```

2、动态规划

- dp[i] 表示前 i 天的最大利润，因为我们始终要使利润最大化，则：
- `dp[i] = max(dp[i-1], prices[i]-minprice)`

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        if n == 0: return 0 # 边界条件
        dp = [0] * n
        minprice = prices[0]

        for i in range(1, n):
            minprice = min(minprice, prices[i])
            dp[i] = max(dp[i - 1], prices[i] - minprice)

        return dp[-1]
```

## 64、求1+2+...+n

---

求 `1+2+...+n` ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

> 输入: n = 3
> 输出: 6

> 输入: n = 9
> 输出: 45

---

**题解**

1、递归，使用逻辑符短路

```python
class Solution:
    def __init__(self):
        self.res = 0
    def sumNums(self, n: int) -> int:
        n > 1 and self.sumNums(n - 1)
        self.res += n
        return self.res
```

优化

```
class Solution:
    def sumNums(self, n: int) -> int:
        return n and n + self.sumNums(n-1)
```

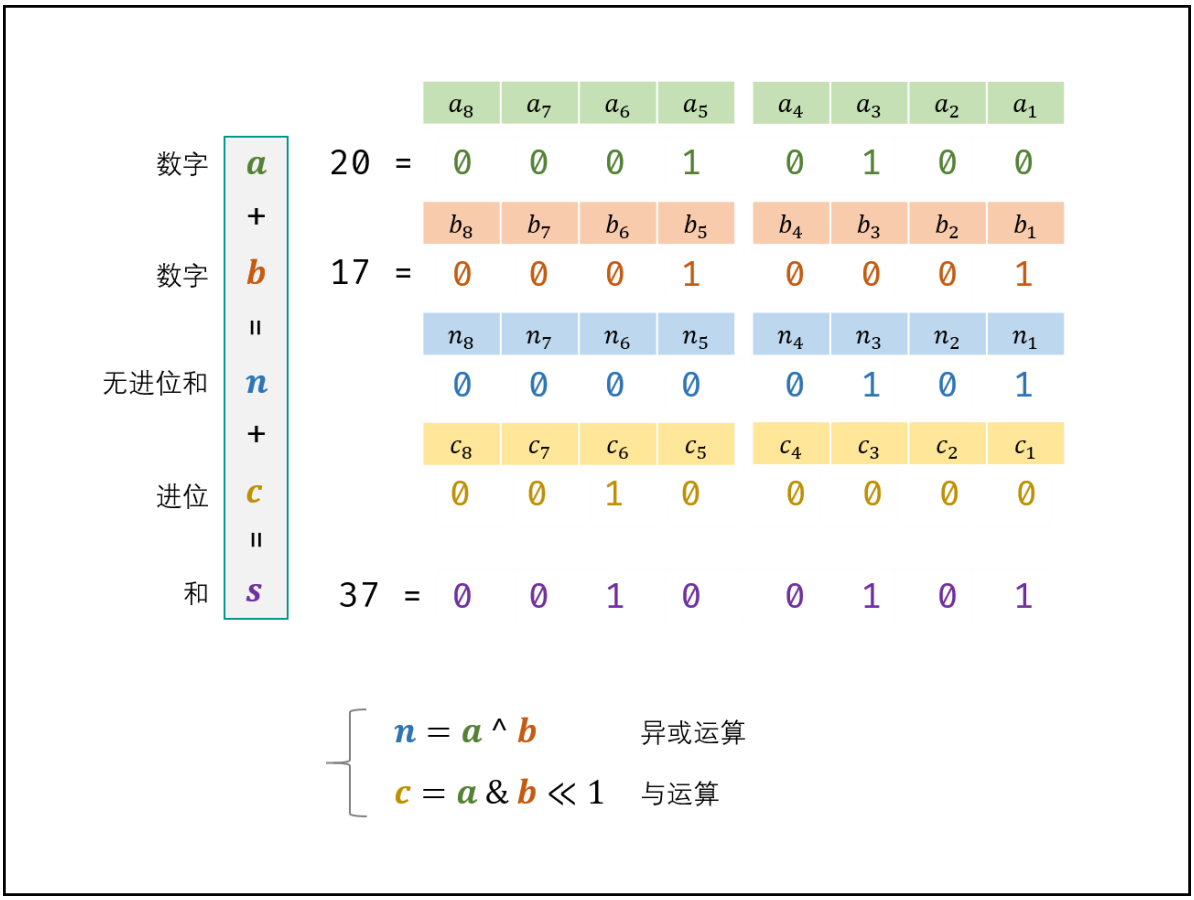
## 65、不用加减整除做加法

写一个函数，求两个整数之和，要求在函数体内不得使用 “+”、“-”、“*”、“/” 四则运算符号。

> 输入: a = 1, b = 1
> 输出: 2

**题解**

1、位运算




```
class Solution:
    def add(self, a: int, b: int) -> int:
        x = 0xffffffff
        a, b = a & x, b & x
        while b != 0:
            a, b = (a ^ b), (a & b) << 1 & x
        return a if a <= 0x7fffffff else ~(a ^ x)
```

2、递归写法

```python
class Solution:
    def add(self, a: int, b: int) -> int:
        x=0xffffffff
        a,b=a&x,b&x
        if b==0:
            return a if a<=0x7fffffff else ~(a^x)

        return self.add(a^b,(a&b)<<1&x)
```

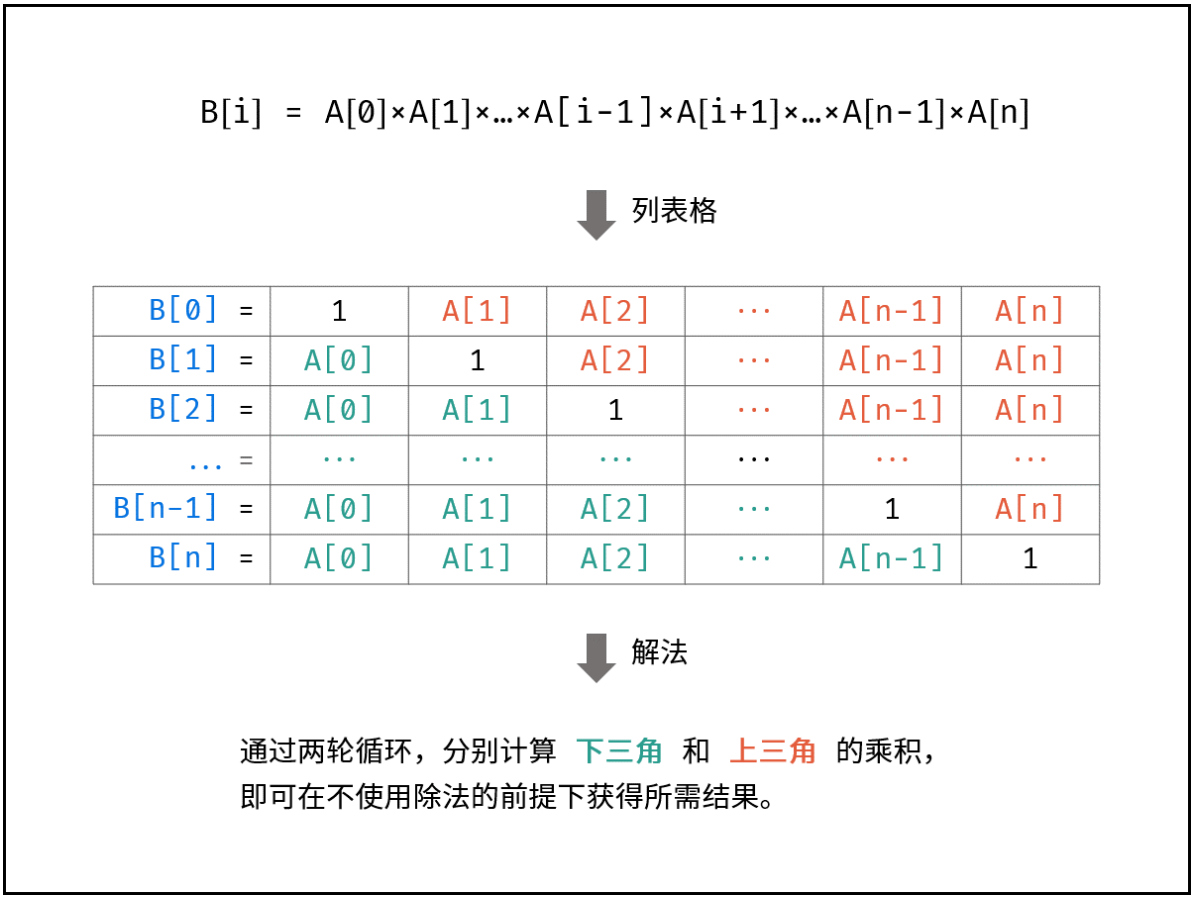
## 66、构建乘积数组

给定一个数组 A[0,1,...,n-1]，请构建一个数组 B[0,1,...,n-1]，其中 B[i] 的值是数组 A 中除了下标 i 以外的元素的积, 即 B[i]=A[0]×A[1]×...×A[i-1]×A[i+1]×...×A[n-1]。不能使用除法。

> 输入: [1,2,3,4,5]
> 输出: [120,60,40,30,24]

**题解**

1、表格分区



```python
class Solution:
    def constructArr(self, a: List[int]) -> List[int]:
        b, tmp = [1] * len(a), 1
        for i in range(1, len(a)):
            b[i] = b[i - 1] * a[i - 1] # 下三角
        for i in range(len(a) - 2, -1, -1):
            tmp *= a[i + 1]            # 上三角
            b[i] *= tmp                # 下三角 * 上三角
        return b
```

## 67、把字符串转换成整数

写一个函数 StrToInt，实现把字符串转换成整数这个功能。不能使用 atoi 或者其他类似的库函数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−231，231 − 1]。如果数值超过这个范围，请返回 INT_MAX (231 − 1) 或 INT_MIN (−231) 。

> 输入: "   -42"
> 输出: -42
> 解释: 第一个非空白字符为 '-', 它是一个负号。
>      我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

> 输入: "4193 with words"
> 输出: 4193
> 解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。

> 输入: "-91283472332"
> 输出: -2147483648
> 解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
>      因此返回 INT_MIN (−231) 。

> 输入: "words and 987"
> 输出: 0
> 解释: 第一个非空字符是 'w', 但它不是数字或正、负号。
>  因此无法执行有效的转换。

**题解**

1、分情况判断

1. 首部空格： 删除之即可；
2. 符号位： 三种情况，即 "+" , "−" , "无符号" ； 新建一个变量保存符号位，返回前判断正负即可。
3. 非数字字符： 遇到首个非数字的字符时，应立即返回。
4. 数字字符：
    1. 字符转数字： “此数字的 ASCII 码” 与 “ 0 的 ASCII 码” 相减即可；
    2. 数字拼接： 若从左向右遍历数字，设当前位字符为 c ， 当前位数字为 x ， 数字结果为 res ， 则数字拼接公式为——`res=10*res+x`

```
class Solution:
    def strToInt(self, str: str) -> int:
        str = str.strip()                      # 删除首尾空格
        if not str: return 0                   # 字符串为空则直接返回
        res, i, sign = 0, 1, 1
        int_max, int_min, bndry = 2 ** 31 - 1, -2 ** 31, 2 ** 31 // 10
        if str[0] == '-': sign = -1            # 保存负号
        elif str[0] != '+': i = 0              # 若无符号位，则需从 i = 0 开始数字拼接
        for c in str[i:]:
            if not '0' <= c <= '9' : break     # 遇到非数字的字符则跳出
            if res > bndry or res == bndry and c > '7': return int_max if sign
== 1 else int_min # 数字越界处理
            res = 10 * res + ord(c) - ord('0') # 数字拼接
        return sign * res
```

即INT_MIN明明是-2147483648，但是为什么却只需要判断当前字符是否大于7，因为按常理来说当符号为负时str.charAt(j)为'8'也是不越界的，但是为什么这里只判断‘7’呢？ 这是因为我们如果直接向一个int变量赋值-2147483648，系统是会报错的（至少在C++中是这样），我们如果想要返回这个数那就需要使用INT_MIN，因为INT_MIN=-2147483648。 那么我们回过头来看这题，考虑字符串为“-2147483648”这种情况，即res==214748364的前提下，当前字符是‘8’，那么按照最常规的思路，这个数是在int的取值范围中，那么我要存放它，但是int变量并不能直接存放这个数，因此我要找一个与-2147483648相等的，可以表示它的数，那就是INT_MIN。因此即使符号是负号，我们仍然可以将判断条件写为str.charAt(j) > '7' ，因为当取‘8’时我们返回的也是INT_MIN。

关于为什么不能直接向int变量赋值-2147483648，我搜索了一些资料，大致意思是说-2147483648是一个常量表达式而非常量，系统会把它分成两部分，即负号 - 和 数字 2147483648，因此会出现越界的情况。

2、正则表达式

- `^`：匹配字符串开头
- `[\+\-]`：代表一个+字符或-字符
- `?`：前面一个字符可有可无
- `\d`：一个数字
- `+`：前面一个字符的一个或多个

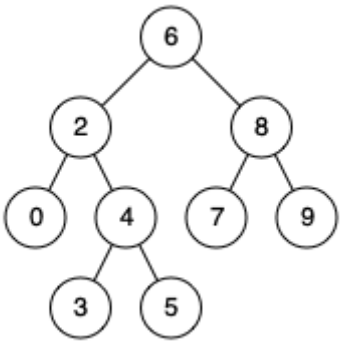
```python
class Solution:
    def strToInt(self, str: str) -> int:
        INT_MAX = 2147483647
        INT_MIN = -2147483648
        s= str.strip()
        res = re.compile(r'^[\+\-]?\d+')
        num = res.findall(s)
        nums = int(*num)
        return max(min(nums, INT_MAX), INT_MIN)
```

## 68-Ⅰ、二叉搜索数的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]




```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

**题解**

1、保存路径，对比路径判断

- 使用dfs找到p，q节点从根的路径
- 反向遍历找到相等的节点

```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        def dfs(root,path,t):
            if root==t:
                return path+[root]
            elif root.val<t.val:
```

```
                return dfs(root.right,path+[root],t)
            else:
                return dfs(root.left,path+[root],t)

        p_num=dfs(root,[],p)
        q_num=dfs(root,[],q)
        for i in p_num[::-1]:
            for j in q_num[::-1]:
                if i==j:
                    return i
```

2、迭代，若 root 是 p,q 的 最近公共祖先 ，则只可能为以下情况之一

- p 和 q 在root 的子树中，且分列 root 的 异侧（即分别在左、右子树中）；
- p = root，且 q 在 root 的左或右子树中；
- q = root，且 p 在 root的左或右子树中；

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        while root:
            if root.val < p.val and root.val < q.val: # p,q 都在 root 的右子树中
                root = root.right # 遍历至右子节点
            elif root.val > p.val and root.val > q.val: # p,q 都在 root 的左子树中
                root = root.left # 遍历至左子节点
            else: break
        return root
```

优化：若可以保证p.val<q.val，可在循环中减少判断条件

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        if p.val > q.val: p, q = q, p # 保证 p.val < q.val
        while root:
            if root.val < p.val: # p,q 都在 root 的右子树中
                root = root.right # 遍历至右子节点
            elif root.val > q.val: # p,q 都在 root 的左子树中
                root = root.left # 遍历至左子节点
            else: break
        return root
```

3、递归

1. 递推工作：
    1. 当 p, q都在 root 的 右子树 中，则开启递归 root.right 并返回；
    2. 否则，当 p, q都在 root的 左子树 中，则开启递归 root.left 并返回；
2. 返回值： 最近公共祖先 root。

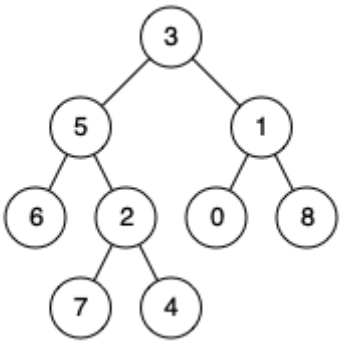
```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        if root.val < p.val and root.val < q.val:
            return self.lowestCommonAncestor(root.right, p, q)
        if root.val > p.val and root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)
        return root
```

## 68-II、二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]




```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。
```

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

**题解**

1、同上第一种解法

- 使用dfs保存根节点到p和q的路径
- 倒序判断那个节点相等，输出相等节点

```
class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
TreeNode:
        def dfs(root,path,t):
            if root==t:
                return path+[root]
            if not root:return []
```

```
            left=dfs(root.left,path+[root],t)
            right=dfs(root.right,path+[root],t)

            return right if len(left)==0 else left

        p_num=dfs(root,[],p)
        q_num=dfs(root,[],q)
        for i in p_num[::-1]:
            for j in q_num[::-1]:
                if i==j:
                    return i
```

2、递归

1. 终止条件：
    1. 当越过叶节点，则直接返回 null；
    2. 当root 等于 p,q，则直接返回 root；
2. 递推工作：
    1. 开启递归左子节点，返回值记为 left；
    2. 开启递归右子节点，返回值记为 right；
3. 返回值： 根据 left 和 right，可展开为四种情况；
    1. 当 left 和 right 同时为空： 说明 root 的左 / 右子树中都不包含 p,q，返回 null；
    2. 当 left 和 right 同时不为空： 说明 p,q 分列在 root 的 异侧 （分别在 左 / 右子树），因
    3. 此 root 为最近公共祖先，返回 root；
    4. 当 left 为空，right 不为空： p,q 都不在 root 的左子树中，直接返回 right 。具体可分为两种情况：
        1. p,q 其中一个在 root 的 右子树 中，此时 right 指向 p（假设为 p ）；
        2. p,q 两节点都在 root 的 右子树 中，此时的 right 指向 最近公共祖先节点；
4. 当 left 不为空， right 为空： 与情况 3. 同理；

```
class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
TreeNode:
        if not root or root == p or root == q: return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if not left: return right
        if not right: return left
        return root
```