1-10

6、Z字形变换

将一个给定字符串 s 根据给定的行数 numRows ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "PAYPALISHIRING" 行数为 3 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数

> 输入：s = "PAYPALISHIRING", numRows = 3
> 输出："PAHNAPLSIIGYIR"

> 输入：s = "PAYPALISHIRING", numRows = 4
> 输出："PINALSIGYAHRPI"
> 解释：
> P   I   N
> A   L S I G
> Y A   H R
> P   I

题解

1、创建numRows行的数组，

- numRows行，则数据 2*numRows-2 为一个循环
- 分别判断位置，

```python
class Solution:
    def convert(self, s: str, numRows: int) -> str:
        if numRows==1:return s

        res=['' for _ in range(numRows)]
        n=numRows*2-2
        for i in range(len(s)):
            x=i%n
            if x<numRows:
                res[x]+=s[i]
            else:
                res[n-x]+=s[i]

        return ''.join(res)
```

使用标志符号

```python
class Solution:
    def convert(self, s: str, numRows: int) -> str:
        if numRows < 2: return s
        res = ["" for _ in range(numRows)]
        i, flag = 0, -1
        for c in s:
            res[i] += c
            if i == 0 or i == numRows - 1: flag = -flag
            i += flag
        return "".join(res)
```

8、字符串转换整数

请你来实现一个 myAtoi(string s) 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 atoi 函数）。

函数 myAtoi(string s) 的算法如下：

- 读入字符串并丢弃无用的前导空格
- 检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。 确定最终结果是负数还是正数。 如果两者都不存在，则假定结果为正。
- 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
- 将前面步骤读入的这些数字转换为整数（即，"123" -> 123， "0032" -> 32）。如果没有读入数字，则整数为 0 。必要时更改符号（从步骤 2 开始）。
- 如果整数数超过 32 位有符号整数范围 [−231，231 − 1]，需要截断这个整数，使其保持在这个范围内。具体来说，小于 −231 的整数应该被固定为 −231 ，大于 231 − 1 的整数应该被固定为 231 − 1 。
- 返回整数作为最终结果。

注意：

- 本题中的空白字符只包括空格字符 ' '。
- 除前导空格或数字后的其余字符串外，请勿忽略 任何其他字符。

> ```
> 输入：s = "   -42"
> 输出：-42
> 解释：
> 第 1 步："   -42"（读入前导空格，但忽视掉）
>          ^
> 第 2 步："   -42"（读入 '-' 字符，所以结果应该是负数）
>           ^
> 第 3 步："   -42"（读入 "42"）
>             ^
> 解析得到整数 -42 。
> 由于 "-42" 在范围 [-231, 231 - 1] 内，最终结果为 -42 。
> ```

```
输入：s = "words and 987"
输出：0
解释：
第 1 步："words and 987"（当前没有读入字符，因为没有前导空格）
         ^
第 2 步："words and 987"（当前没有读入字符，因为这里不存在 '-' 或者 '+'）
         ^
第 3 步："words and 987"（由于当前字符 'w' 不是一个数字，所以读入停止）
         ^
解析得到整数 0 ，因为没有读入任何数字。
由于 0 在范围 [-231, 231 - 1] 内，最终结果为 0 。
```

```
输入：s = "-91283472332"
输出：-2147483648
解释：
第 1 步："-91283472332"（当前没有读入字符，因为没有前导空格）
         ^
第 2 步："-91283472332"（读入 '-' 字符，所以结果应该是负数）
          ^
第 3 步："-91283472332"（读入 "91283472332"）
                     ^
解析得到整数 -91283472332 。
由于 -91283472332 小于范围 [-231, 231 - 1] 的下界，最终结果被截断为 -231 =
-2147483648 。
```

题解

1、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
- \D：一个非数字字符
- *：前面一个字符的0个或多个

```python
class Solution:
    def myAtoi(self, s: str) -> int:
        return max(min(int(*re.findall('^[\+\-]?\d+', s.lstrip())), 2**31 - 1),
-2**31)
```

2、正常遍历

```python
class Solution:
    def myAtoi(self, str: str) -> int:
        i=0
        n=len(str)
        while i<n and str[i]==' ':
            i=i+1
        if n==0 or i==n:
            return 0
        flag=1
        if str[i]=='-':
```

```
            flag=-1
        if str[i]=='+' or str[i]=='-':
            i=i+1
        INT_MAX=2**31-1
        INT_MIN=-2**31
        ans=0
        while i<n and '0'<=str[i]<='9':
            ans=ans*10+int(str[i])-int('0')
            i+=1
            if(ans-1>INT_MAX):
                break

        ans=ans*flag
        if ans>INT_MAX:
            return INT_MAX
        return INT_MIN if ans<INT_MIN else ans
```

12、整数转罗马数字

罗马数字包含以下七种字符： `I`， `V`， `X`， `L`，`C`，`D` 和 `M`。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给你一个整数，将其转为罗马数字。

> ```
> 输入: num = 58
> 输出: "LVIII"
> 解释: L = 50, V = 5, III = 3.
> ```
>
> ```
> 输入: num = 1994
> 输出: "MCMXCIV"
> 解释: M = 1000, CM = 900, XC = 90, IV = 4.
> ```

题解

1、贪心算法

将哈希表按照从大到小的顺序排列，然后遍历哈希表，直到表示完整个输入。

```
class Solution:
    def intToRoman(self, num: int) -> str:
        # 使用哈希表，按照从大到小顺序排列
        hashmap = {1000:'M', 900:'CM', 500:'D', 400:'CD', 100:'C', 90:'XC',
50:'L', 40:'XL', 10:'X', 9:'IX', 5:'V', 4:'IV', 1:'I'}
        res = ''
        for key in hashmap:
            if num // key != 0:
                count = num // key  # 比如输入4000，count 为 4
                res += hashmap[key] * count
                num %= key
        return res
```

2、暴力匹配

```
class Solution:
    def intToRoman(self, num: int) -> str:
        M = ["", "M", "MM", "MMM"] # 1000，2000，3000
        C = ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"] #
100~900
        X = ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"] # 10~90
        I = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"] # 1~9
        return M[num//1000] + C[(num%1000)//100] + X[(num%100)//10] + I[num%10]
```

3、数字位函数处理解法

```
class Solution(object):
    import collections
    def intToRoman(self, num):
        """
        :type num: int
        :rtype: str
        """
        # 数字位函数处理解法
        def handle(digit, c1, c2, c3):
            if digit < 4:
                return digit * c1
            elif digit == 4:
                return c1 + c2
            elif digit == 5:
                return c2
            elif digit < 9:
                return c2 + (digit - 5) * c1
            else:
                return c1 + c3
        d1, d2, d3, d4 = [int(i) for i in '{:04d}'.format(num)]
        return d1 * 'M' + handle(d2, 'C', 'D', 'M') + handle(d3, 'X', 'L', 'C')
+ handle(d4, 'I', 'V', 'X')
```

16、最接近的三数之和

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

> 输入：nums = [-1,2,1,-4], target = 1
> 输出：2
> 解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2) 。

题解

1、双指针，简化题目

- 既然三个数字我们有些无从下手，那么先使用一层for循环，减少一个数字的筛选再来考虑是否就简单了一些。
- 减少一个数字后，我们的题目变成了查找数组中某两个最接近target - num1，是不就变成了一道基础题。
- 实现方案
  - 我们先将nums排序
  - 设置返回值ret，初始为float('inf')无穷大
  - 开始for i in range(len(nums))循环
  - 设置left = i + 1,right = length - 1
  - tmp = nums[left] + nums[rigth]
  - 比较 ret和tmp+nums[i]，哪个更接近target，并赋值给ret
  - 如果tmp = target - nums[i],表示找到了三个数等于target直接返回target
  - 如果tmp > target - nums[i],我们将right -= 1
  - 如果tmp < target - nums[i],我们将left -= 1
  - 最终，即可获取结果。

```
class Solution:
    def threeSumClosest(self, nums, target):
        ret = float('inf')
        nums.sort()
        length = len(nums)
        for i in range(length - 2):
            left = i + 1
            right = length - 1
            while left < right:
                tmp = nums[i] + nums[left] + nums[right]
                ret = tmp if abs(tmp - target) < abs(ret - target) else ret
                if tmp == target:
                    return target
                if tmp > target:
                    right -= 1
                else:
                    left += 1
        return ret
```

18、四数之和

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：答案中不可以包含重复的四元组。

> 输入：nums = [1,0,-1,0,-2,2], target = 0
> 输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

> 输入：nums = [], target = 0
> 输出：[]

题解

1、同三数之和

- 数组排序
- 两层嵌套，然后用双指针选择和相等的四个数
- 最后使用set()函数去重

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        res=[]
        nums.sort()
        for i in range(len(nums)-3):
            for j in range(i+1,len(nums)-2):
                left,right=j+1,len(nums)-1
                while left<right:
                    if nums[i]+nums[j]+nums[left]+nums[right]==target:
                        res.append([nums[i],nums[j],nums[left],nums[right]])
                        left+=1
                    elif nums[i]+nums[j]+nums[left]+nums[right]<target:
                        left+=1
                    else:
                        right-=1

        return list(set([tuple(t) for t in res]))
```

2、回溯算法

1. 当我们选了这个数，但是选了它之后，即时后面连选全数组最大的数，也不能达到target。
   就是说，我们当前这个数太小了。这时候，可以肯定这个数是一定不用选择的。
   target - nums[i] - (3 - len(oneSolution)) * nums[-1] > 0
   - 这里的的3是因为“4数和”题目要求4个数，当前这个选了占用1个，剩3个数。
2. 我们当前的数组的基础上，连续选择当前这个数的话，整体的解的和会比target大，由于我们提前排序了nums，往后去寻找其他的数，只会比当前的数更大，所以这种情况下，我们就不需要往后再去找了。
   target - (4 - len(oneSolution)) * nums[i] < 0
   - 这里的的3是因为“4数和”题目要求4个数。
3. 对于剩下的情况，都有选择，和不选择两种情况。

```python
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        output = []

        def Search(i, target, oneSolution):

            if target == 0 and len(oneSolution) == 4:  # 出口，找到正确的解了
                output.append(oneSolution)
                return
            elif len(oneSolution) > 4 or i >= len(nums):  # 剪枝，超范围了
                return

            if target - nums[i] - (3 - len(oneSolution)) * nums[-1] > 0:  # 当前这个数太小了
                Search(i + 1, target, oneSolution)
            elif target - (4 - len(oneSolution)) * nums[i] < 0:  # 当前组数的和太大了
                return
            else:  # 当前组数似乎没毛病
                Search(i + 1, target, oneSolution)  # 不选这个数
                Search(i + 1, target - nums[i], oneSolution + [nums[i]])  # 选这个数

        Search(0, target, [])

        output1 = []
        output2 = []
        for t in output:
            if set(t) not in output1:
                output1.append(set(t))
                output2.append(t)

        return output2
```

提前去重（剪枝）

```python
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        output = []

        def Search(i, target, oneSolution, notSelected):
            if target == 0 and len(oneSolution) == 4:
                output.append(oneSolution)
                return
            elif len(oneSolution) > 4 or i >= len(nums):
                return

            if target - nums[i] - (3 - len(oneSolution)) * nums[-1] > 0 or nums[i] in notSelected:
                Search(i + 1, target, oneSolution, notSelected)
            elif target - (4 - len(oneSolution)) * nums[i] < 0:
                return
            else:
                Search(i + 1, target, oneSolution, notSelected + [nums[i]])
```

```
                Search(i + 1, target - nums[i], oneSolution + [nums[i]],
notSelected)


        Search(0, target, [], [])

        return output
```

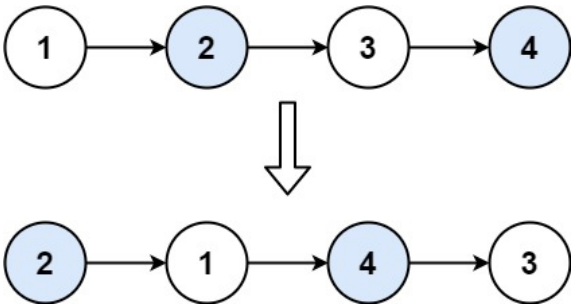
3、N数之和——使用递归+双指针

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        def nSum(nums: List[int], n: int, target: int) -> List[List[int]]:
            res = []
            if len(nums) < n:
                return res
            if n == 2:
                left, right = 0, len(nums) - 1
                while left < right:
                    if nums[left] + nums[right] == target:
                        res.append([nums[left], nums[right]])
                        while left < right and nums[left] == nums[left+1]:
                            left += 1
                        while left < right and nums[right] == nums[right-1]:
                            right -= 1
                        left += 1
                        right -= 1
                    elif nums[left] + nums[right] < target:
                        left += 1
                    else:
                        right -= 1
                return res
            else:
                for i in range(len(nums)-n+1):
                    if i > 0 and nums[i] == nums[i-1]:
                        continue
                    min_sum = sum(nums[i:i+n])
                    if min_sum > target:
                        break
                    max_sum = nums[i] + sum(nums[-n+1:])
                    if max_sum < target:
                        continue
                    sub_res = nSum(nums[i+1:], n-1, target-nums[i])
                    for j in range(len(sub_res)):
                        res.append([nums[i]]+sub_res[j])
                return res
        nums.sort()
        res = nSum(nums, 4, target)

        return res
```

24、两两交换链表中的节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

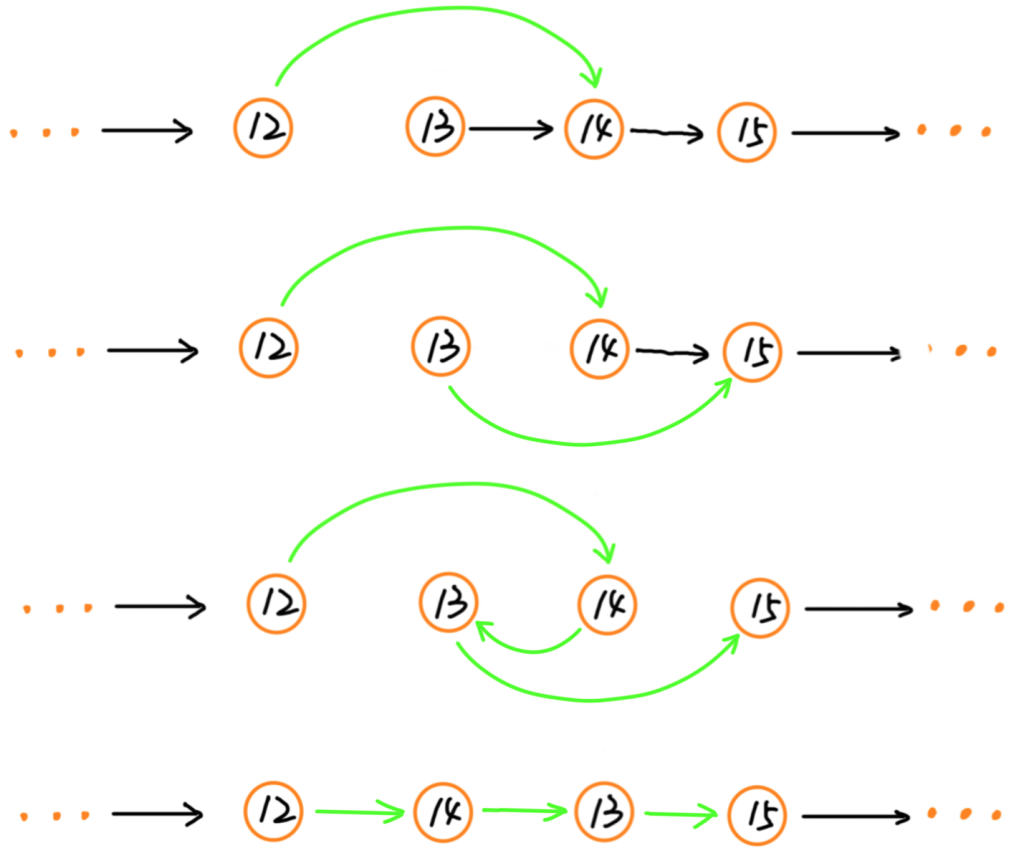



```
输入：head = [1,2,3,4]
输出：[2,1,4,3]
```

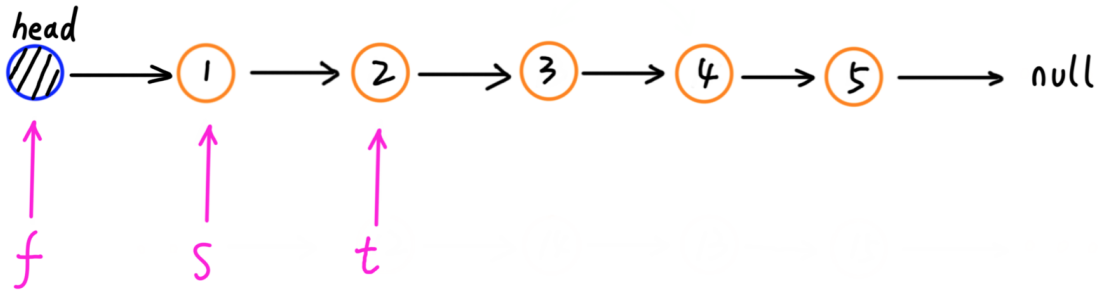
```
输入：head = [1]
输出：[1]
```

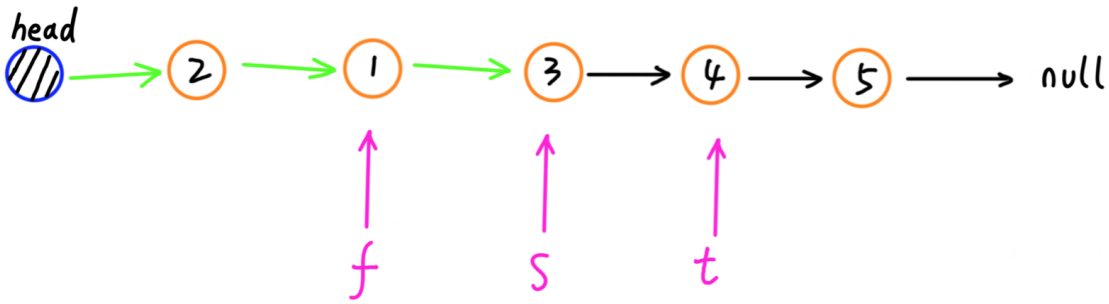
题解

1、链表操作




```
1. 12 -> next = 14
2. 13 -> next = 14 -> next
3. 14 -> next = 13
```






```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        # 已有的链表加一个头部 head node
        resultHead = ListNode()
        resultHead.next = head

        # curNode 遍历链表时用
        curNode = resultHead

        # 开始遍历链表
        while curNode and curNode.next and curNode.next.next:
            f = curNode
            s = curNode.next
            t = curNode.next.next

            # 两两交换链表结点
            f.next = t
            s.next = t.next
            t.next = s
            # 标杆位后移2位
            curNode = curNode.next.next

        return resultHead.next
```

2、递归

```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head
        newHead = head.next
        head.next = self.swapPairs(newHead.next)
        newHead.next = head
        return newHead
```

29、两数相除

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

整数除法的结果应当截去（truncate）其小数部分，例如：truncate(8.345) = 8 以及 truncate(-2.7335) = -2

> 输入: dividend = 10, divisor = 3
> 输出: 3
> 解释: 10/3 = truncate(3.33333..) = truncate(3) = 3

> 输入: dividend = 7, divisor = -3
> 输出: -2
> 解释: 7/-3 = truncate(-2.33333..) = -2

题解

1、倍增法

```python
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:

        # 将被除数和除数转化为正数
        sign = 1
        if divisor * dividend < 0:  # 如果符号不同，则结果返回要变成负数
            sign = -1
            divisor = abs(divisor)
            dividend = abs(dividend)

        elif divisor < 0 and dividend < 0: # 如果被除数和除数都是负值，结果不修改符号
            divisor = abs(divisor)
            dividend = abs(dividend)

        remain = dividend  # 余数
        result = 0  # 商
        while remain >= divisor:
            cur = 1 # 倍增商
            div = divisor # 倍增值
            while div + div < remain:
                cur += cur
                div += div
            remain -= div  # 余数递减
            result += cur  # 商值累计

        if sign==-1:
            result = -result

        if result>=2**31:  # 按照题目要求，溢出处理
            result = 2**31-1

        return result
```

2、二进制搜索思想

- 举例：算 63 / 8
  过程为：63 / 8 = (63-32) / 8 + 4 = (63-32-16) / 8 + 2 + 4 = (63-32-16-8) / 8 + 1+ 2 + 4 = 7
  其中(63-32-16-8) / 8 = 7 / 8 = 0
- 递归

```
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        MIN_INT, MAX_INT = -2147483648, 2147483647  # [-2**31, 2**31-1]
        flag = 1                                    # 存储正负号，并将分子分母转化为正
数
        if dividend < 0: flag, dividend = -flag, -dividend
        if divisor < 0: flag, divisor  = -flag, -divisor

        def div(dividend, divisor):                 # 例：1023 / 1 = 512 + 256 +
128 + 64 + 32 + 16 + 8 + 4 + 1
            if dividend < divisor:
                return 0
            cur = divisor
            multiple = 1
            while cur + cur < dividend:             # 用加法求出保证divisor *
multiple <= dividend的最大multiple
                cur += cur                          # 即cur分别乘以1, 2, 4, 8,
16...2^n，即二进制搜索
                multiple += multiple
            return multiple + div(dividend - cur, divisor)
        res = div(dividend, divisor)

        res = res if flag > 0 else -res             # 恢复正负号

        if res < MIN_INT:                           # 根据是否溢出返回结果
            return MIN_INT
        elif MIN_INT <= res <= MAX_INT:
            return res
        else:
            return MAX_INT
```

- 迭代

```
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        MIN_INT, MAX_INT = -2147483648, 2147483647  # [-2**31, 2**31-1]
        flag = 1                                    # 存储正负号，并将分子分母转化为正
数
        if dividend < 0: flag, dividend = -flag, -dividend
        if divisor < 0: flag, divisor  = -flag, -divisor

        res = 0
        while dividend >= divisor:                  # 例：1023 / 1 = 512 + 256 +
128 + 64 + 32 + 16 + 8 + 4 + 1
            cur = divisor
            multiple = 1
            while cur + cur < dividend:             # 用加法求出保证divisor *
multiple <= dividend的最大multiple
```

```
                cur += cur                                  # 即cur分别乘以1, 2, 4, 8,
16...2^n，即二进制搜索
                multiple += multiple
            dividend -= cur
            res += multiple

        res = res if flag > 0 else -res                     # 恢复正负号

        if res < MIN_INT:                                   # 根据是否溢出返回结果
            return MIN_INT
        elif MIN_INT <= res <= MAX_INT:
            return res
        else:
            return MAX_INT
```

36、有效的数独

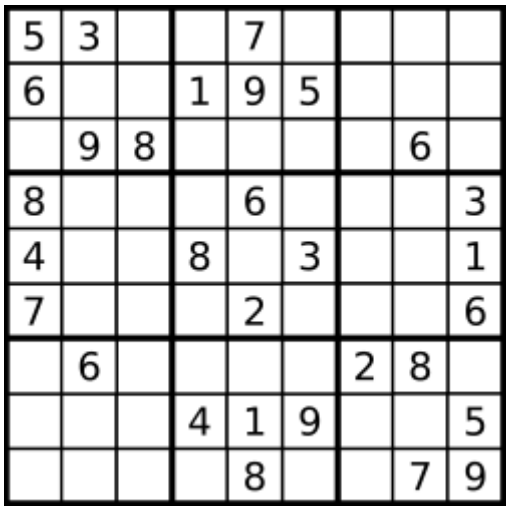
请你判断一个 9x9 的数独是否有效。只需要 根据以下规则 ， 验证已经填入的数字是否有效即可。

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 '.' 表示。

注意：

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。



```
输入：board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".",".","8",".",".","7","9"]]
输出：true
```

题解

1、哈希表

```
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        row = [set() for i in range(9)]
        col = [set() for i in range(9)]
        block = [[set() for i in range(3)] for j in range(3)]
        for i in range(9):
            for j in range(9):
                if board[i][j]!='.':
                    if board[i][j] in row[i] or board[i][j] in col[j] \
                        or board[i][j] in block[i//3][j//3]:
                            print(i,j)
                            return False
                    row[i].add(board[i][j])
                    col[j].add(board[i][j])
                    block[i//3][j//3].add(board[i][j])
        return True
```

2、定义一个判断函数，然后分别判断

```
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        def isvaild9(lyst):
            nums = list(filter(lambda x:x != '.', lyst))
            return len(set(nums)) == len(nums)

        for row in board:#9行
            if not isvaild9(row):
                return False

        for column in zip(*board):#9列
            if not isvaild9(column):
                return False

        for row in range(3):#9块
            for column in range(3):
                tmp = [board[i][j] for i in range(row*3, row*3+3) for j in
range(column*3, column*3+3)]
                if not isvaild9(tmp):
                    return False
```

```
        return True
```

40、组合总和

给定一个数组 candidates 和一个目标数 target ， 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

> ```
> 输入: candidates = [10,1,2,7,6,1,5], target = 8,
> 所求解集为:
> [
> [1, 7],
> [1, 2, 5],
> [2, 6],
> [1, 1, 6]
> ]
> ```
>
> ```
> 输入: candidates = [2,5,2,1,2], target = 5,
> 所求解集为:
> [
> [1,2,2],
> [5]
> ]
> ```

题解

1、回溯算法

1. 剪枝：如果当前tmp数组的和cur已经大于目标target，没必要枚举了，直接return
2. 如果当前tmp数组的和cur正好和目标target相等，找到一个组合，加到结果res中去，并return
3. for循环遍历从index开始的数，选一个数进入下一层递归。
    - 如果从index开始的数有连续出现的重复数字，跳过该数字continue，因为这会产生重复解
    - 因为数不可以重复选择，所以在进入下一层递归时，i要加1，从i之后的数中选择接下来的数

```
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) ->
List[List[int]]:
        def backtrack(tmp, cur, index):
            if cur > target:
                return
            if cur == target:
                res.append(tmp)
                return
            for i in range(index, n):
                if i > index and candidates[i] == candidates[i - 1]:
```

```
                    continue
                backtrack(tmp + [candidates[i]], cur + candidates[i], i + 1)

        res = []
        n = len(candidates)
        candidates.sort()
        backtrack([], 0, 0)
        return res
```

43、字符串相乘

给定两个以字符串形式表示的非负整数 `num1` 和 `num2`，返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

> 输入: num1 = "2", num2 = "3"
> 输出: "6"

> 输入: num1 = "123", num2 = "456"
> 输出: "56088"

题解

1、字典保存字符对应数字，然后从个位遍历，得到数之后相乘

```
class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        dic={}
        for i in range(10):
            dic[str(i)]=i
        n1,n2=0,0
        for i in range(len(num1)):
            n1+=dic[num1[-1-i]]*(10**i)
        for i in range(len(num2)):
            n2+=dic[num2[-1-i]]*(10**i)

        return str(n1*n2)
```

2、模拟手算

```
class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        res = [0] * (len(num1) + len(num2))
        for i in range(len(num1)-1, -1, -1):
            for j in range(len(num2)-1, -1, -1):
                mul = int(num1[i]) * int(num2[j])
                p1 = i+j
                p2 = i + j + 1
                sum_ = res[p2] + mul
                res[p2] = sum_ % 10
                res[p1] += sum_ // 10
```

```
        for i in range(len(res)):    # 去除res前面的0
            if res[i] != 0:
                break
        res = [str(v) for v in res[i:]]

        return "".join(res)
```

11-20

45、跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

> 输入: [2,3,1,1,4]
> 输出: 2
> 解释: 跳到最后一个位置的最小跳跃数是 2。
> 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

> 输入: [2,3,0,1,4]
> 输出: 2

题解

1、贪心算法

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        """
        n 长度
        step 步长
        left , right  当前位置元素能到达的第一个元素和最后一个元素
        cur 当前位置的元素
        """
        n = len(nums)
        step = 0
        left, right, cur = 0, 0, 0

        # 1. 当元素个数为1 直接返回
        if len(nums) == 1:
            return 0

        # 2. 循环
        while left < n:
            # 1. 每次更新当前元素的 left,  right
            left = right + 1
            right = cur + nums[cur]
```

```
            # 4. 如果 right 值大于 n-1了，说明下一步一定能到，返回
            if right >= n-1:
                return step + 1

            # 2. 根据贪心算法，找到能走的最远的 下一个元素
            temp = 0
            for i in range(left, right+1):
                if i + nums[i] > temp:
                    temp = i + nums[i]
                    cur = i
            # 3. 步数加一
            step += 1
```

2、动态规划——超时间

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        dp=[float('inf')]*len(nums)
        dp[0]=0
        for i in range(1,len(nums)):
            for j in range(i):
                if i-j<=nums[j]:
                    dp[i]=min(dp[i],dp[j]+1)

        return dp[-1]
```

优化：

1. 首先初始化DP数组，默认不跳，每个格子的走的次数就是它的下标
2. 每到一个格子，更新它所有能到的格子的下标，如果比原来存储的少，则替换

```
class Solution(object):
    def jump(self, nums):
        res = [i for i in range(len(nums))]
        for i in range(len(nums)):
            # temp =i(现在位置的坐标)+nums[i]（能飞的最大距离）
            temp = i+nums[i]
            # 每个位置更新一遍
            for j in range(i,temp+1):
                if j<len(nums):
                    res[j]=min(res[j],res[i]+1)
        return res[-1]
```

47、全排列II

给定一个可包含重复数字的序列 `nums` ，**按任意顺序** 返回所有不重复的全排列。

> 输入：nums = [1,1,2]
> 输出：
> [[1,1,2],
> [1,2,1],
> [2,1,1]]

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

题解

1、递归

不重复一样的列表元素

```python
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        result = []
        def permuteUnique(ans, nums):
            if not nums:
                return result.append(ans)

            for i in set(nums):
                new_ans = [i]
                new_nums = nums.copy()
                new_nums.remove(i)
                permuteUnique(ans + new_ans, new_nums)
        permuteUnique([], nums)
        return result
```

50、Pow(x,n)

实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数（即，x**n）

```
输入：x = 2.00000, n = 10
输出：1024.00000
```

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

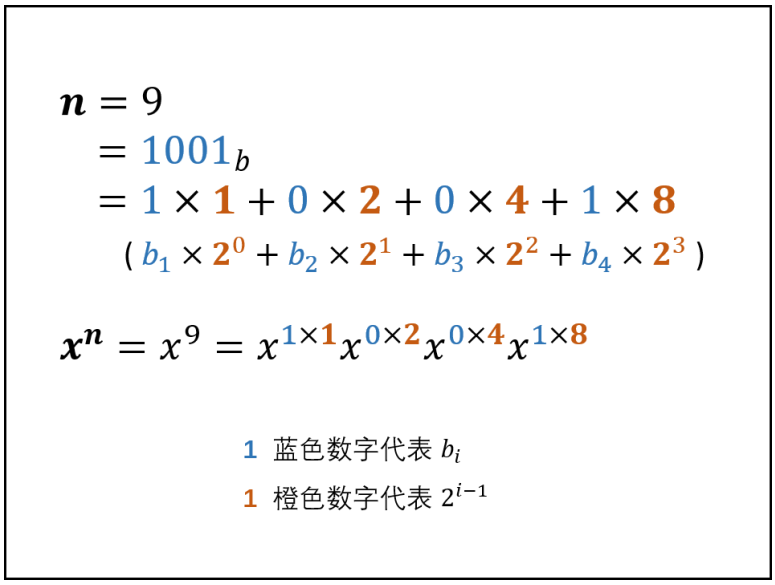
```
输入：x = 2.00000, n = -2
输出：0.25000
解释：2-2 = 1/22 = 1/4 = 0.25
```

题解

1、快速幂

- 对于任何十进制正整数 n，设其二进制为 "bm...b3b2b1“（bi为二进制某位值，i∈[1，m]），则有：
  - 二进制转十进制：n=1b1+2b2+4b3+.....+2^m-1bm
  - 幂的二进制展开：$x^{n=x}$(1b1+2b2+3b3+...+2^m-1bm)
- 根据以上推导，可把计算 x^n 转化为解决以下两个问题：
  - 计算$x^{1,x}2,x^{4,...,x}2$(m-1)的值，循环赋值操作x=x^2即可

- 获取二进制各位b1,b2,b3,...,bm的值，循环执行以下操作即可：
    1. n&1(与操作)：判断n二进制最后一位是否为1
    2. n>>1(移位操作)：n右移一位（可以理解为删除最后一位）



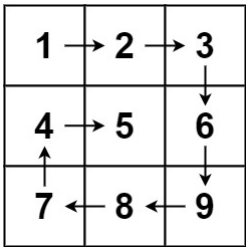

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if x == 0.0: return 0.0
        res = 1
        if n < 0: x, n = 1 / x, -n
        while n:
            if n & 1: res *= x
            x *= x
            n >>= 1          #另一种写法：n//2
        return res
```

2、递归版本

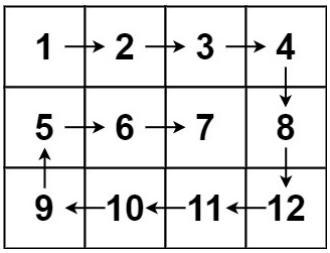
```python
class Solution(object):
    def myPow(self, x, n):
        """
        :type x: float
        :type n: int
        :rtype: float
        """
        if n == 0:
            return 1
        if n < 0:
            x = 1 / x
            n = -n
        if n % 2:
            return x * self.myPow(x, n - 1)
        return self.myPow(x * x, n / 2)
```

54、螺旋矩阵

给你一个 `m` 行 `n` 列的矩阵 `matrix` ，请按照 **顺时针螺旋顺序** ，返回矩阵中的所有元素。



```
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)
```

57、插入区间

给你一个 **无重叠的** ，按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

> 输入：intervals = [[1,3],[6,9]], newInterval = [2,5]
> 输出：[[1,5],[6,9]]

```
输入：intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
输出：[[1,2],[3,10],[12,16]]
解释：这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。
```

```
输入：intervals = [[1,5]], newInterval = [2,3]
输出：[[1,5]]
```

题解

1、二分法——寻找左右边界

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) ->
List[List[int]]:
        L, R = newInterval
        # 二分查找（左），参考bisect.bisect_left
        pL, r = 0, len(intervals)
        while(pL<r):
            m = (pL+r)//2
            if intervals[m][1]<L: pL = m+1
            else: r = m
        # 二分查找（右），参考bisect.bisect_right
        pR, r =  pL, len(intervals)
        while(pR<r):
            m = (pR+r)//2
            if intervals[m][0]>R: r = m
            else: pR = m+1

        # python slice替换，待替换片段和原片段等长时，时间复杂度为O(R-L)，否则为O(N-L)
        intervals[pL:pR] = [newInterval] if pL==pR else [[min(L, intervals[pL]
[0]), max(R, intervals[pR-1][1])]]
        return intervals
```

2、简单遍历——分别考虑不同情况

```
def insert(intervals, newInterval):
    n = len(intervals)
    if n==0: return [newInterval]
    # 不需要 merge 的情形 加在末端or开始
    if newInterval[0]>intervals[-1][1]: return intervals+[newInterval]
    if newInterval[1]<intervals[0][0]: return [newInterval]+intervals
    merged,k = [],0
    while k<n:
        if intervals[k][1]<newInterval[0]:
            merged.append(intervals[k])
            k+=1
        elif intervals[k][1]>=newInterval[0]:
            # newInterval 在 intervals[k] 的左侧不相交
            # 例: intervals=[[3,5],[12,15]] newInterval = [6,6]
            if newInterval[1]<intervals[k][0]:
                return merged + [newInterval] + intervals[k:]
            # newInterval 和 intervals的最后一个区间有交集
            # 例: intervals=[[1,5]] newInterval =[2,3]
            if newInterval[1]>=intervals[-1][0]:
```

```
                return merged + [[min(intervals[k]
[0],newInterval[0]),max(newInterval[1], intervals[-1][1])]]
            # newInterval和intervals[k]有交集 且 intervals[k_new]是右侧第一个不相交的
区间
            k_new = k+1
            while k_new < n and intervals[k_new][0]<=newInterval[1]:
                k_new+=1
            interval_merge = [[min(intervals[k][0],newInterval[0]),
max(intervals[k_new-1][1], newInterval[1])]]
            return merged+interval_merge+intervals[k_new:]
```

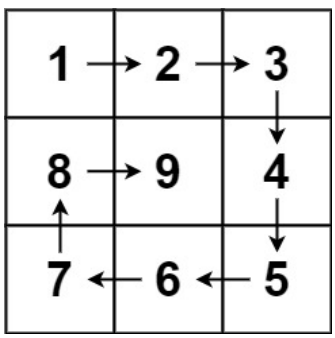
3、原地修改

- 先找到第一个与待插入重叠的区间， 向后扩大区间直到没有重叠区域
- 最后删除所有曾重叠的子区间，插入最终新增的区间

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) ->
List[List[int]]:
        # 初始状况判断
        if not newInterval:
            return intervals
        if not intervals:
            return [newInterval]
        # 已经是起点有序的了
        i = 0
        intervalsLen = len(intervals)
        while i < intervalsLen and intervals[i][1] < newInterval[0]:
            i += 1
        # 保存删除之前的位置，最后在这个位置上插入
        tempI = i
        while i < intervalsLen and intervals[i][0] <= newInterval[1]:
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1
        else:
            del intervals[tempI:i]
            intervals.insert(tempI, newInterval)
        return intervals
```

59、螺旋矩阵II

给你一个正整数 `n` ，生成一个包含 `1` 到 `n2` 所有元素，且元素按顺时针顺序螺旋排列的 `n x n` 正方形矩阵 `matrix` 。



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

题解

1、同54，定义标志，循环走一圈一圈的

```
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        res=[[0]*n for _ in range(n)]
        x,y=0,0
        left,right,up,down=0,n-1,0,n-1
        curd=0
        dirs = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for i in range(1,n*n+1):
            res[x][y]=i
            if curd==0 and y==right:
                curd+=1
                up+=1
            elif curd==1 and x==down:
                curd+=1
                right-=1
            elif curd==2 and y==left:
                curd+=1
                down-=1
            elif curd==3 and x==up:
                curd+=1
                left+=1
            curd%=4
            x+=dirs[curd][0]
            y+=dirs[curd][1]
        return res
```

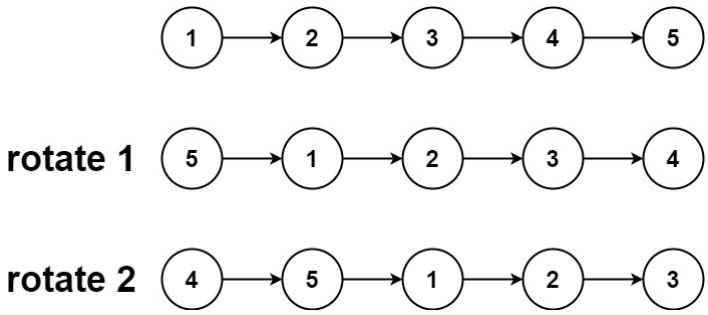
2、设定边界

```
class Solution:
    def generateMatrix(self, n: int) -> [[int]]:
        l, r, t, b = 0, n - 1, 0, n - 1
        mat = [[0 for _ in range(n)] for _ in range(n)]
        num, tar = 1, n * n
        while num <= tar:
            for i in range(l, r + 1): # left to right
                mat[t][i] = num
                num += 1
            t += 1
            for i in range(t, b + 1): # top to bottom
                mat[i][r] = num
                num += 1
            r -= 1
            for i in range(r, l - 1, -1): # right to left
                mat[b][i] = num
                num += 1
            b -= 1
            for i in range(b, t - 1, -1): # bottom to top
```

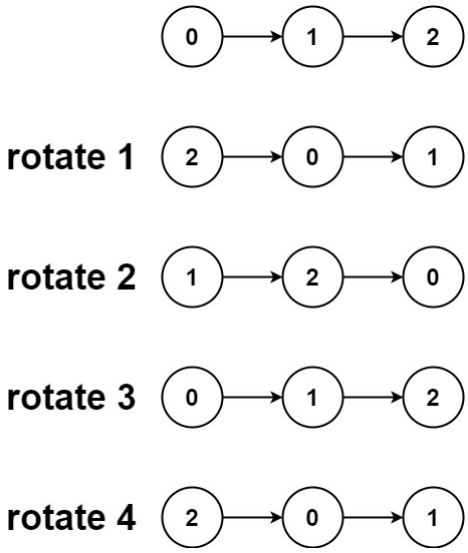
```
                mat[i][l] = num
                num += 1
            l += 1
        return mat
```

61、旋转链表

给你一个链表的头节点 `head` ，旋转链表，将链表每个节点向右移动 `k` 个位置。




```
输入：head = [1,2,3,4,5], k = 2
输出：[4,5,1,2,3]
```




```
输入：head = [0,1,2], k = 4
输出：[2,0,1]
```

题解

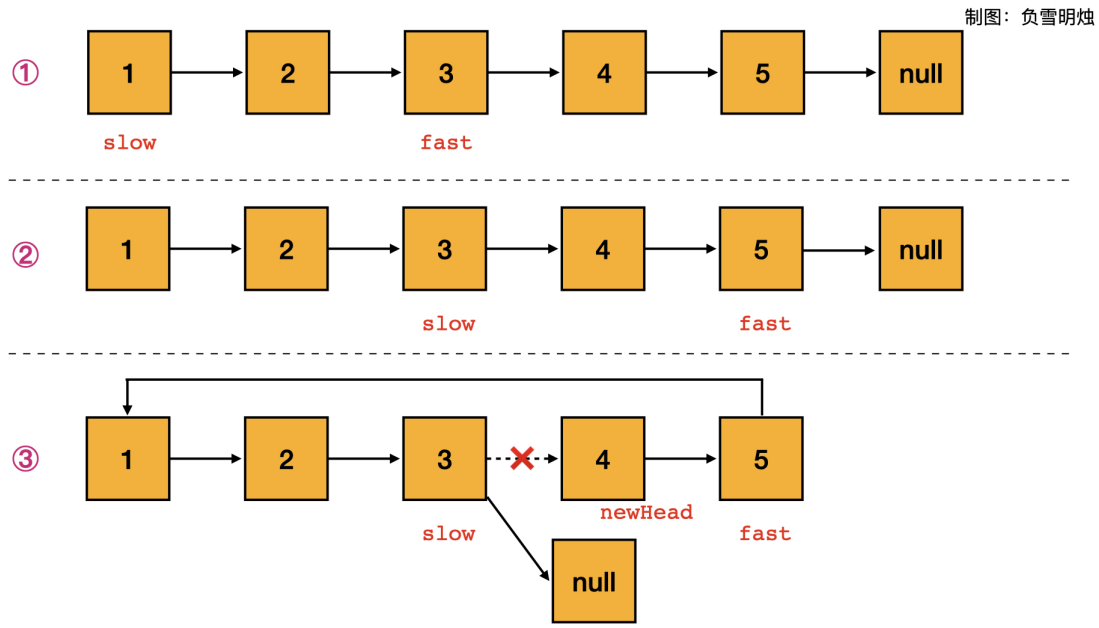
1、链表转数组，数组转链表

```
class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        if not head:return head
        res=[]
        while head:
            res.append(head.val)
            head=head.next
```

```
        k%=len(res)
        res=res[-k:]+res[:-k]
        root=ListNode(None)
        r=root
        for i in res:
            r.next=ListNode(i)
            r=r.next

        return root.next
```

2、链表拼接

1. 两个指针 slow 和 fast 值距离是 k，先让 fast 指向链表的第 k + 1个节点，slow 指向第 1 个节点；
2. 然后 slow 和 fast 同时向后移动，当 fast 移动到链表的最后一个节点的时候，那么 slow 指向链表的倒数第 k + 1个节点。
3. 链表重整
    1. newHead 是新链表的头部，它应该是原链表倒数第 kk 个节点，即 slow.next；
    2. slow 需要跟 slow.next 断开；
    3. fast 是老链表的结尾，将 fast.next 设置为老链表的开头，实现首尾相接。




```
class Solution:
    def rotateRight(self, head, k):
        if not head or not head.next: return head
        # 求链表长度
        _len = 0
        cur = head
        while cur:
            _len += 1
            cur = cur.next
        # 对长度取模
        k %= _len
        if k == 0: return head
        # 让 fast 先向后走 k 步
        fast, slow = head, head
        while k:
            fast = fast.next
            k -= 1
```

```
        # 此时 slow 和 fast 之间的距离是 k；fast 指向第 k+1 个节点
        # 当 fast.next 为空时，fast 指向链表最后一个节点，slow 指向倒数第 k + 1 个节点
        while fast.next:
            fast = fast.next
            slow = slow.next
        # newHead 是倒数第 k 个节点，即新链表的头
        newHead = slow.next
        # 让倒数第 k + 1 个节点 和 倒数第 k 个节点断开
        slow.next = None
        # 让最后一个节点指向原始链表的头
        fast.next = head
        return newHead
```

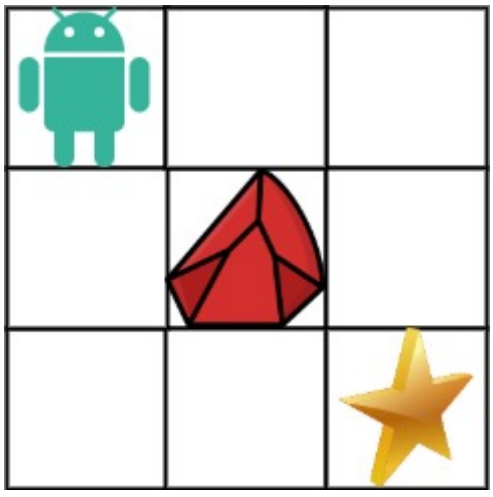
63、不同路径II

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径?

网格中的障碍物和空位置分别用 `1` 和 `0` 来表示。



> 输入：obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
> 输出：2
> 解释：
> 3x3 网格的正中间有一个障碍物。
> 从左上角到右下角一共有 2 条不同的路径：
> 1. 向右 -> 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右 -> 向右

题解

1、动态规划

```
def uniquePathsWithObstacles(grid):
    m, n=len(grid), len(grid[0])
    dp = [[0]*n for _ in range(m)]
    i=0 # 第一列初始化
    while i<m and grid[i][0]!=1:
        dp[i][0]=1
        i+=1
    j=0 # 第一行初始化
    while j<n and grid[0][j]!=1:
        dp[0][j]=1
        j+=1
    # 递归
    for i in range(1,m):
        for j in range(1,n):
            if grid[i][j]!=1:
                if not grid[i-1][j]: dp[i][j]+=dp[i-1][j]
                if not grid[i][j-1]: dp[i][j]+=dp[i][j-1]
    return dp[-1][-1]
```

优化

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m, n = len(obstacleGrid[0]), len(obstacleGrid)
        dp = [1] + [0]*m
        for i in range(0, n):
            for j in range(0, m):
                dp[j] = 0 if obstacleGrid[i][j] else dp[j]+dp[j-1]
        return dp[-2]       #dp[-1]是为了当j = 0时方便计算，不用判断边界，所以dp[-2]才是
解
```

71、简化路径

给你一个字符串 path ，表示指向某一文件或目录的 Unix 风格 绝对路径 （以 '/' 开头），请你将其转化为更加简洁的规范路径。

在 Unix 风格的文件系统中，一个点（.）表示当前目录本身；此外，两个点 （..） 表示将目录切换到上一级（指向父目录）；两者都可以是复杂相对路径的组成部分。任意多个连续的斜杠（即，'//'）都被视为单个斜杠 '/' 。 对于此问题，任何其他格式的点（例如，'...'）均被视为文件/目录名称。

请注意，返回的 规范路径 必须遵循下述格式：

- 始终以斜杠 '/' 开头。
- 两个目录名之间必须只有一个斜杠 '/' 。
- 最后一个目录名（如果存在）不能 以 '/' 结尾。
- 此外，路径仅包含从根目录到目标文件或目录的路径上的目录（即，不含 '.' 或 '..'）。

返回简化后得到的 规范路径 。

```
输入：path = "/home/"
输出："/home"
解释：注意，最后一个目录名后面没有斜杠。
```

```
输入：path = "/../"
输出："/"
解释：从根目录向上一级是不可行的，因为根目录是你可以到达的最高级。
```

```
输入：path = "/home//foo/"
输出："/home/foo"
解释：在规范路径中，多个连续斜杠需要用一个斜杠替换。
```

```
输入：path = "/a/./b/../../c/"
输出："/c"
```

题解

1、标准栈

```python
class Solution:
    def simplifyPath(self, path: str) -> str:
        stack = []
        for p in path.split('/'):
            if p not in ['.', '..', '']:
                stack.append(p)
            elif p == '..' and stack:
                stack.pop()
        return f"/{'/'.join(stack)}"
```

2、python标准库，os模块

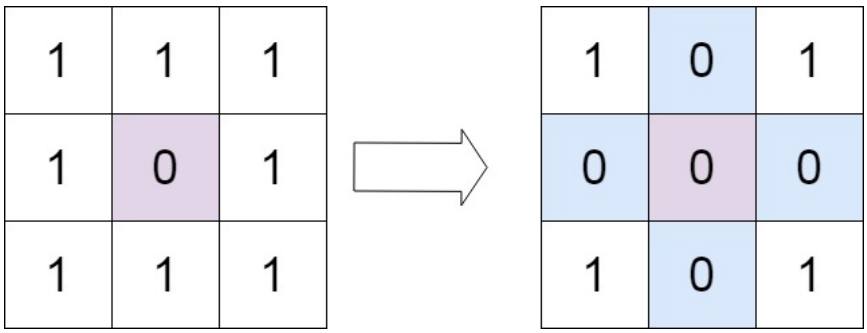
```python
import os
class Solution:
    def simplifyPath(self, path: str) -> str:
        return os.path.abspath(path)
```

73、矩阵置零

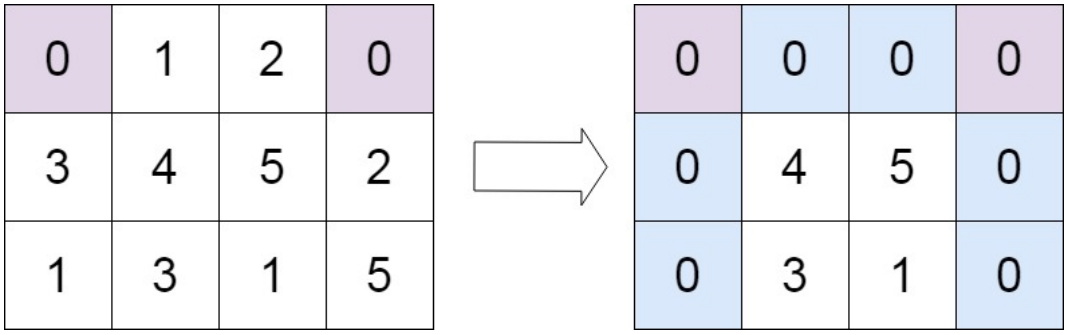
给定一个 m x n 的矩阵，如果一个元素为 0 ，则将其所在行和列的所有元素都设为 0 。请使用 原地 算法。

进阶：

- 一个直观的解决方案是使用 O(mn) 的额外空间，但这并不是一个好的解决方案。
- 一个简单的改进方案是使用 O(m + n) 的额外空间，但这仍然不是最好的解决方案。
- 你能想出一个仅使用常量空间的解决方案吗？




```
输入：matrix = [[1,1,1],[1,0,1],[1,1,1]]
输出：[[1,0,1],[0,0,0],[1,0,1]]
```




```
输入：matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
输出：[[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

题解

1、两遍扫matrix，第一遍用集合记录哪些行，那些有0，第二遍置0，——O（m+n）

```python
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        row = len(matrix)
        col = len(matrix[0])
        row_zero = set()
        col_zero = set()
        for i in range(row):
            for j in range(col):
                if matrix[i][j] == 0:
                    row_zero.add(i)
                    col_zero.add(j)
        for i in range(row):
            for j in range(col):
                if i in row_zero or j in col_zero:
                    matrix[i][j] = 0
```

2、用`matrix`第一行和第一列记录该行该列是否有`0`,作为标志位，但是对于第一行,和第一列要设置一个标志位,为了防止自己这一行(一列)也有`0`的情况.——O（1）

```python
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
```

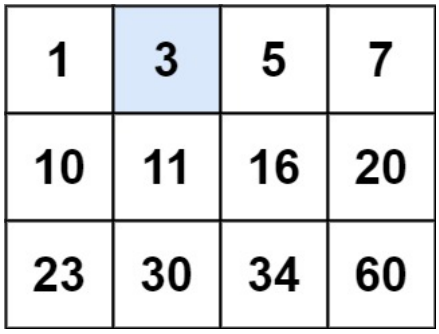
```
        """
        Do not return anything, modify matrix in-place instead.
        """
        row = len(matrix)
        col = len(matrix[0])
        row0_flag = False
        col0_flag = False
        # 找第一行是否有0
        for j in range(col):
            if matrix[0][j] == 0:
                row0_flag = True
                break
        # 第一列是否有0
        for i in range(row):
            if matrix[i][0] == 0:
                col0_flag = True
                break

        # 把第一行或者第一列作为 标志位
        for i in range(1, row):
            for j in range(1, col):
                if matrix[i][j] == 0:
                    matrix[i][0] = matrix[0][j] = 0
        #print(matrix)
        # 置0
        for i in range(1, row):
            for j in range(1, col):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        if row0_flag:
            for j in range(col):
                matrix[0][j] = 0
        if col0_flag:
            for i in range(row):
                matrix[i][0] = 0
```

21-30

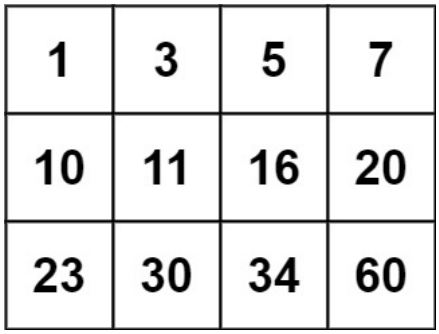
74、搜索二维矩阵

编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。



```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出：true
```



```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
输出：false
```

题解

1、遍历行，判断是否大于第一个数和小于最后一个数，如果在该范围，使用二分法判断是否在这一行

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        for i in range(len(matrix)):
            if matrix[i][0]<=target<=matrix[i][-1]:
                left,right=0,len(matrix[0])
                while left<right:
                    mid=(left+right)//2
                    if matrix[i][mid]==target:
                        return True
                    elif matrix[i][mid]<target:
                        left=mid+1
                    else:
                        right=mid
                return False

        return False
```

优化：两次二分

```python
def searchMatrix(matrix, target):
    m, n=len(matrix), len(matrix[0])
    # 第一步：先找到target所在行 记每行的第一个数组成的列表为 [a[0],...,a[m-1]]
    # 先二分搜索找到 a[i]<=target<a[i+1] i=max{k |target>=a[k]}
    l, r = 0, m-1
    while l<r:
```

```
        mid=(l+r+1)//2
        if target>=matrix[mid][0]: #去搜寻看 k 大一点是否仍满足target>=matrix[k][0]
            l=mid
        else: #target<matrix[mid][0] 需要更小的k使target>=matrix[k][0]满足
            r=mid-1
    row=l
    # 第二步: 找到 matrix[l][i]<=target<matrix[l][i+1] '<','<='也行 为了和前面一致
    l, r = 0, n-1
    while l<r:
        mid=(l+r+1)//2
        if target>=matrix[row][mid]: #去搜寻看 k 大一点是否仍满足target>=matrix[k][0]
            l=mid
        else: #target<matrix[mid][0] 需要更小的k使target>=matrix[k][0]满足
            r=mid-1
    return matrix[row][l]==target
```

2、把数组按行拼接，然后对一个长的一维数组使用二分法（时间更快！99%）

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        mt=[]
        for i in range(len(matrix)):
            mt.extend(matrix[i])
        left,right=0,len(mt)
        while left<right:
            mid=(left+right)//2
            if mt[mid]==target:
                return True
            elif mt[mid]<target:
                left=mid+1
            else:
                right=mid

        return False
```

3、从左下角或者右上角开始查找

- 如果要搜索的 target 比当前元素大，那么让行增加；
- 如果要搜索的 target 比当前元素小，那么让列减小；

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        if not matrix or not matrix[0]:
            return False
        rows = len(matrix)
        cols = len(matrix[0])
        row, col = 0, cols - 1
        while True:
            if row < rows and col >= 0:
                if matrix[row][col] == target:
                    return True
                elif matrix[row][col] < target:
                    row += 1
                else:
                    col -= 1
            else:
```

```
                return False
```

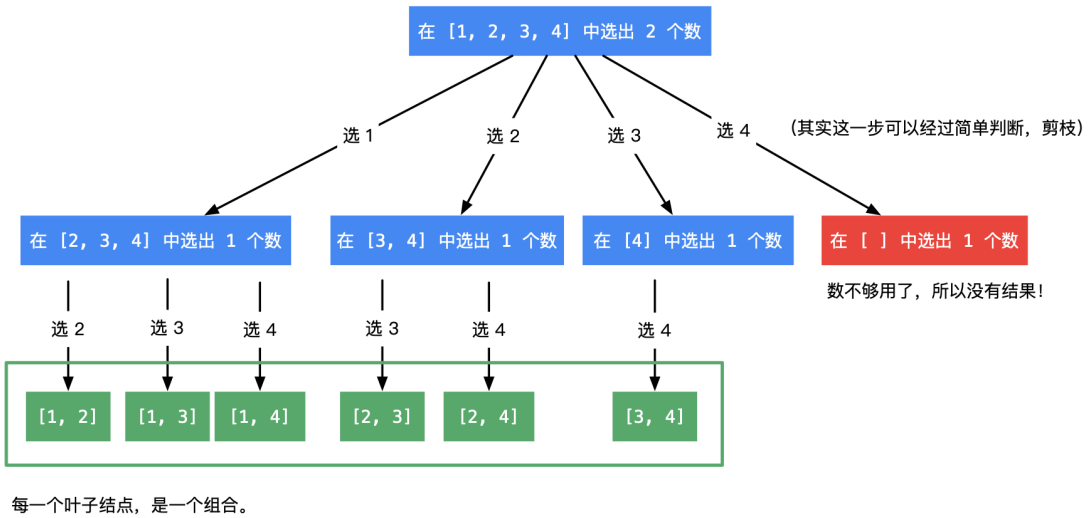
77、组合

给定两个整数 *n* 和 *k*，返回 1 ... *n* 中所有可能的 *k* 个数的组合。

```
输入: n = 4, k = 2
输出:
[
[2,4],
[3,4],
[2,3],
[1,2],
[1,3],
[1,4],
]
```

题解

1、回溯+剪枝



每一个叶子结点，是一个组合。

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        self.res=[]#用于存放结果的数组
        self.path=[]#用于存放临时结果的数组

        #设定递归函数参数和返回值
        def backtrack(n,k,startindex):
            #确定终止条件
            if len(self.path)==k:#设定返回条件，当临时数组达到相应长度k时认为需要返回
                self.res.append(list(self.path))#将结果保存进res数组
                return
            #单层递归逻辑
            for i in range(startindex,n+1):#从startindex开始一直到n+1(此处只取到n)
```

```
                if i > n-(k-len(self.path))+1:break#剪枝操作，根据此时path中的元素数来舍去不需要考虑的数字

                self.path.append(i)#结果加入临时数组
                backtrack(n,k,i+1)#继续向里搜索下一个数字，从i+1出开始搜索，避免出现[1,1]这样的结果对
                self.path.pop()#回溯时需要弹出结果，才能在后面继续加结果，例如[1,2]需要弹出2才能加入3变成[1,3]
        #运行回溯函数
        backtrack(n,k,1)
        return self.res
```

80、删除有序数组中的重复项II

给你一个有序数组 nums ，请你 原地 删除重复出现的元素，使每个元素 最多出现两次 ，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

> 输入：nums = [1,1,1,2,2,3]
> 输出：5, nums = [1,1,2,2,3]
> 解释：函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1, 2, 2, 3 。 不需要考虑数组中超出新长度后面的元素。

> 输入：nums = [0,0,1,1,1,1,2,3,3]
> 输出：7, nums = [0,0,1,1,2,3,3]
> 解释：函数应返回新长度 length = 7, 并且原数组的前五个元素被修改为 0, 0, 1, 1, 2, 3, 3 。 不需要考虑数组中超出新长度后面的元素。

题解

1、快慢指针

- **慢指针 slow**：指向当前即将放置元素的位置；则 `slow - 1` 是刚才已经放置了元素的位置。
- **快指针 fast**：向后遍历所有元素；

```
class Solution(object):
    def removeDuplicates(self, nums):
        slow = 0
        for fast in range(len(nums)):
            if slow < 2 or nums[fast] != nums[slow - 2]:
                nums[slow] = nums[fast]
                slow += 1
        return slow
```

2、计数，相同的值出现次数不能超过2 index修改，每次修改后index+1继续遍历

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        #index计数， 也是修改索引,从索引1开始修改
        #count记录值出现的次数,不能超过2，从计数为1开始
        index = count = 1
```

```
        #遍历nums，从索引1位置开始遍历
        for i in range(1, len(nums)):
            #如果nums[i] == nums[i - 1],则相同值的计数加1
            if nums[i] == nums[i - 1]:
                count += 1
            #不等的话，计数值重置为1
            else:
                count = 1
            #如果count小于等于2，则可以修改index位置的元素：nums[index] = nums[i]
            if count <= 2:
                nums[index] = nums[i]
                #修改位置索引index+1
                index += 1
        #返回修改的索引值，也就是修改后的nums长度
        return index
```

81、搜索旋转排序数组

已知存在一个按非降序排列的整数数组 nums ， 数组中的值不必互不相同。

在传递给函数之前，nums 在预先未知的某个下标 k（0 <= k < nums.length）上进行了 旋转 ， 使数组变为 [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]（下标 从 0 开始 计数）。例如， [0,1,2,4,4,4,5,6,6,7] 在下标 5 处经旋转后可能变为 [4,5,6,6,7,0,1,2,4,4] 。

给你 旋转后 的数组 nums 和一个整数 target ， 请你编写一个函数来判断给定的目标值是否存在于数组中。如果 nums 中存在这个目标值 target ， 则返回 true ， 否则返回 false 。

> 输入：nums = [2,5,6,0,0,1,2], target = 0
> 输出：true

> 输入：nums = [2,5,6,0,0,1,2], target = 3
> 输出：false

题解

1、直接查找

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        if target in nums:
            return True

        return False
```

2、先排序，再二分法

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        nums.sort()
        left,right=0,len(nums)
        while left<right:
```

```
            mid=(left+right)//2
            if nums[mid]==target:
                return True
            elif nums[mid]<target:
                left=mid+1
            else:
                right=mid

        return False
```

3、找出旋转的索引，然后二分法查找

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        t=-1
        for i in range(len(nums)-1):
            if nums[i]>nums[i+1]:
                t=i
                break
        nums=nums[t+1:]+nums[:t+1]

        left,right=0,len(nums)
        while left<right:
            mid=(left+right)//2
            if nums[mid]==target:
                return True
            elif nums[mid]<target:
                left=mid+1
            else:
                right=mid

        return False
```

4、直接二分查找

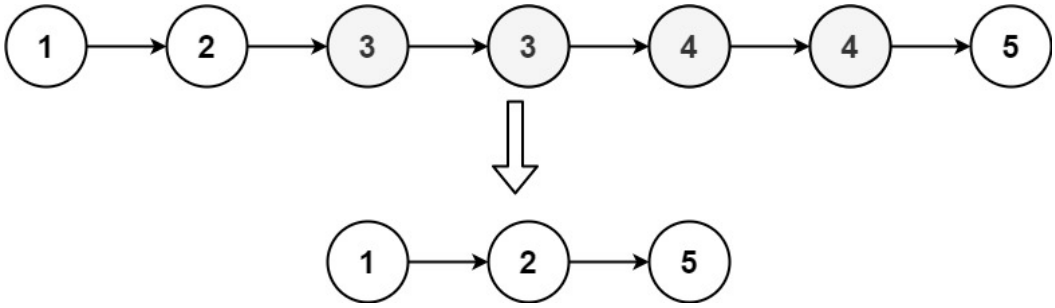
```
class Solution(object):
    def search(self, nums, target):
        if not nums: return -1
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) / 2
            if nums[mid] == target:
                return True
            if nums[left] == nums[right]:
                left += 1
                continue
            if nums[mid] <= nums[right]:
                if target > nums[mid] and target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
            else:
                if target < nums[mid] and target >= nums[left]:
                    right = mid - 1
                else:
                    left = mid + 1
```

```
        return False
```

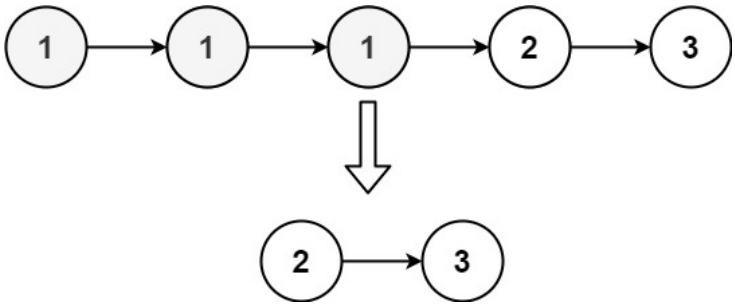
82、删除排序链表中的重复元素II

存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中 没有重复出现 的数字。

返回同样按升序排列的结果链表。




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




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

题解

1、递归

- 递归终止条件
  - 如果 head 为空，那么肯定没有值出现重复的节点，直接返回 head；
  - 如果 head.next 为空，那么说明链表中只有一个节点，也没有值出现重复的节点，也直接返回 head。
- 递归调用
  - 如果 head.val != head.next.val ，说明头节点的值不等于下一个节点的值，所以当前的 head 节点必须保留；但是 head.next 节点要不要保留呢？我们还不知道，需要对 head.next 进行递归，即对 head.next 作为头节点的链表，去除值重复的节点。所以 head.next = self.deleteDuplicates(head.next).
  - 如果 head.val == head.next.val ，说明头节点的值等于下一个节点的值，所以当前的 head 节点必须删除，并且 head 之后所有与 head.val 相等的节点也都需要删除；删除到哪个节点为止呢？需要用 move 指针一直向后遍历寻找到与 head.val 不等的节点。此时 move 之前的节点都不保留了，因此返回 deleteDuplicates(move);

- 返回结果
  - 如果 head.val != head.next.val ，头结点需要保留，因此返回的是 head；
  - 如果 head.val == head.next.val ，头结点需要删除，需要返回的是 deleteDuplicates(move);。

```
class Solution(object):
    def deleteDuplicates(self, head):
        if not head or not head.next:
            return head
        if head.val != head.next.val:
            head.next = self.deleteDuplicates(head.next)
        else:
            move = head.next
            while move and head.val == move.val:
                move = move.next
            return self.deleteDuplicates(move)
        return head
```

2、一边遍历、一边统计相邻节点的值是否相等，如果值相等就继续后移找到值不等的位置，然后删除值相等的这个区间。

```
class Solution(object):
    def deleteDuplicates(self, head):
        if not head or not head.next:
            return head
        dummy = ListNode(0)
        dummy.next = head
        pre = dummy
        cur = head
        while cur:
            # 跳过当前的重复节点，使得cur指向当前重复元素的最后一个位置
            while cur.next and cur.val == cur.next.val:
                cur = cur.next
            if pre.next == cur:
                 # pre和cur之间没有重复节点，pre后移
                pre = pre.next
            else:
                # pre->next指向cur的下一个位置（相当于跳过了当前的重复元素）
                # 但是pre不移动，仍然指向已经遍历的链表结尾
                pre.next = cur.next
            cur = cur.next
        return dummy.next
```

3、两次遍历，计数

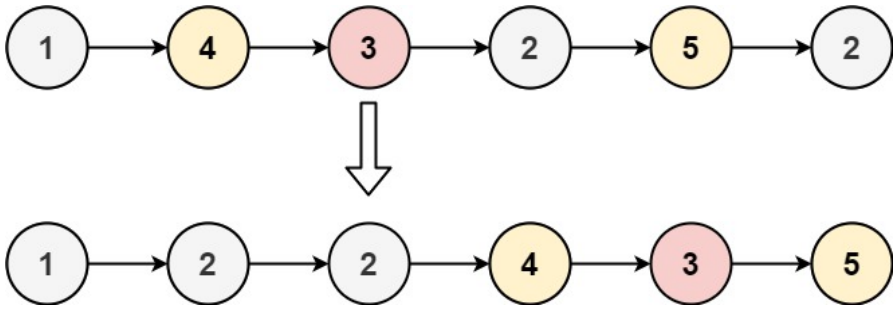
```
class Solution:
    def deleteDuplicates(self, head):
        dummy = ListNode(0)
        dummy.next = head
        val_list = []
        while head:
            val_list.append(head.val)
            head = head.next
        counter = collections.Counter(val_list)
        head = dummy
```

```
        while head and head.next:
            if counter[head.next.val] != 1:
                head.next = head.next.next
            else:
                head = head.next
        return dummy.next
```

86、分隔链表

给你一个链表的头节点 head 和一个特定值 x ，请你对链表进行分隔，使得所有 小于 x 的节点都出现在大于或等于 x 的节点之前。

你应当 保留 两个分区中每个节点的初始相对位置。




```
输入：head = [1,4,3,2,5,2], x = 3
输出：[1,2,2,4,3,5]
```

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

题解

1、大小链表

- 遍历一次head，
  - 将大于x的值追加到大链表
  - 将小于x的值追加到小链表
  - 然后合并两个链表

```
class Solution:
    def partition(self, head, x):
        if not head:
            return head
        small = ListNode()
        sm = small
        large = ListNode()
        lg = large
        while head:
            if head.val < x:
                sm.next = head
                sm = sm.next
            else:
                lg.next = head
```

```
                lg = lg.next
            head = head.next
        lg.next = None
        sm.next = large.next
        return small.next
```

2、寻找第一个large链表头

```
class Solution:
    def partition(self, head, x):
        tmp = ret = ListNode()
        li = head
        ins = None
        while li:
            if li.val < x:
                tmp.next = li
                tmp = tmp.next
                li = li.next
            else:
                ins = li
                break
        while li and li.next:
            if li.next.val < x:
                tmp.next = li.next
                tmp = li.next
                li.next = li.next.next
            else:
                li = li.next
        tmp.next = ins
        return ret.next
```

3、双指针原地修改

1. 参数定义
    - `left`：左指针，始终指向左边第一个大于等于`x`节点的前一个位置
    - `right`：右指针，寻找右边第一个小于`x`的节点插到`left`后面
2. 思路
    - left指针从左到右找到第一个大于等于x的前一个位置
    - right从大于等于x的位置遍历到第一个小于x的位置tmp，即tmp=right.next
    - 原地移动：right指向tmp的后一位，right.next=tmp.next,tmp节点移动到left的下一个位置left.next=tmp，同时left移动到tmp,left.next=tmp

```
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode:
        if not head:
            return
        dummy=ListNode(0)
        dummy.next=head
        left=dummy
        while left.next and left.next.val<x:
                left=left.next

        right=left.next
        while right:
            while right.next and right.next.val>=x:
```

```
                right=right.next
            if not right.next:
                break
            # 原地移动
            tmp=right.next                  #tmp指向比x小的数
            right.next=tmp.next             #右指针跳过tmp指向tmp的下一个数
            tmp.next=left.next              #tmp的下一个指向第一个大于x的地方
            left.next=tmp                   #左指针指向tmp
            left = tmp                      #做指针更新

        return dummy.next
```

89、格雷编码

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。即使有多个不同答案，你也只需要返回其中一种。

格雷编码序列必须以 0 开头。

> ```
> 输入: 2
> 输出: [0,1,3,2]
> 解释:
> 00 - 0
> 01 - 1
> 11 - 3
> 10 - 2
>
> 对于给定的 n，其格雷编码序列并不唯一。
> 例如，[0,2,3,1] 也是一个有效的格雷编码序列。
>
> 00 - 0
> 10 - 2
> 11 - 3
> 01 - 1
> ```

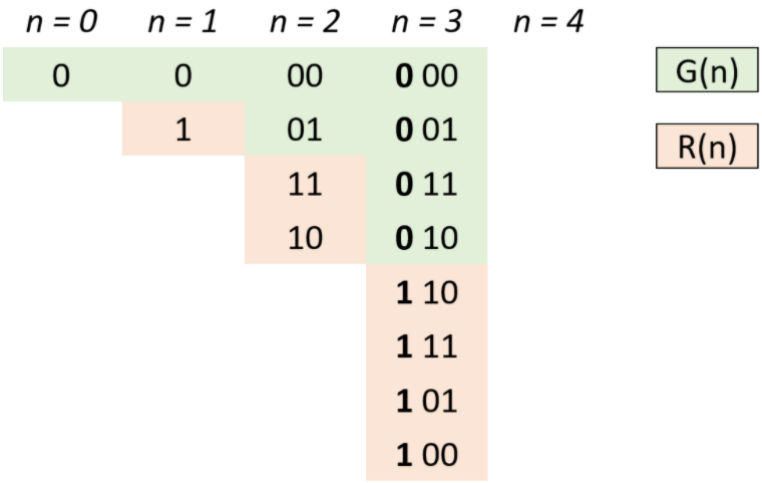
> ```
> 输入: 0
> 输出: [0]
> 解释: 我们定义格雷编码序列必须以 0 开头。
>      给定编码总位数为 n 的格雷编码序列，其长度为 2n。当 n = 0 时，长度为 20 = 1。
>      因此，当 n = 0 时，其格雷编码序列为 [0]。
> ```

题解

1、镜像反射法

- 设 n 阶格雷码集合为 G(n)，则 G(n+1)阶格雷码为：
  - 给 G(n)阶格雷码每个元素二进制形式前面添加 0，得到 G'(n)
  - 设 G(n) 集合倒序（镜像）为 R(n)，给 R(n) 每个元素二进制形式前面添加 1，得到 R'(n)
  - G(n+1) = G'(n) ∪ R'(n)G(n+1) 拼接两个集合即可得到下一阶格雷码。
- 根据以上规律，可从 00 阶格雷码推导致任何阶格雷码。

- 由于最高位前默认为 00，因此 G'(n) = G(n)，只需在 res(即 G(n))后添加 R'(n)即可；
- 计算 R'(n)：执行 head = 1 << i 计算出对应位数，以给 R(n)前添加 1 得到对应 R'(n)
- 倒序遍历 res(即 G(n)G(n))：依次求得 R'(n)各元素添加至 res 尾端，遍历完成后 res(即 G(n+1)) 。




```
class Solution:
    def grayCode(self, n: int) -> List[int]:
        res, head = [0], 1
        for i in range(n):
            for j in range(len(res) - 1, -1, -1):
                res.append(head + res[j])
            head <<= 1
        return res
```

2、找规律

- n=0 时， g(n)=[0]
- n>1时，`g(n)=g(n-1)*2 +reverse(g(n-1)*2+1)`
    - g(2)=[00,10]+reverse([01,11])=[00,10,11,01]=[0,2,3,1]

```
def grayCode(n):
    if n==0: return [0]
    res=[0]
    for i in range(1,n+1):
        res=[2*k for k in res]+[2*k+1 for k in res[::-1]]
    return res
```

3、递归

```python
class Solution:
    def grayCode(self, n: int) -> List[int]:
        def gray(n):
            if n==0:
                return ["0"]
            if n==1:
                return ["0","1"]
            n_1 = gray(n-1)
            n_1_reverse = n_1[::-1]
            return [x+"0" for x in n_1]+[x+"1" for x in n_1_reverse]

        return [int(x,2) for x in gray(n)]
```

90、子集II

给你一个整数数组 nums ，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。返回的解集中，子集可以按 任意顺序排列。

> 输入：nums = [1,2,2]
> 输出：[[],[1],[1,2],[1,2,2],[2],[2,2]]

> 输入：nums = [0]
> 输出：[[],[0]]

题解

1、回溯

```python
class Solution(object):
    def subsetsWithDup(self, nums):
        res = []
        nums.sort()
        self.dfs(nums, 0, res, [])
        return res

    def dfs(self, nums, index, res, path):
        if path not in res:
            res.append(path)
        for i in range(index, len(nums)):
            if i > index and nums[i] == nums[i - 1]:
                continue
            self.dfs(nums, i + 1, res, path + [nums[i]])
```

2、一次遍历

```python
class Solution(object):
    def subsetsWithDup(self, nums):
        res = [[]]
        nums.sort()
        for i in range(len(nums)):
            if i >= 1 and nums[i] == nums[i-1]:
                new_subsets = [subset + [nums[i]] for subset in new_subsets]
            else:
                new_subsets = [subset + [nums[i]] for subset in res]
            res = new_subsets + res
        return res
```

优化

```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        ans=[[]]
        for n in nums:
            ans+=[x+[n] for x in ans if x+[n] not in ans]
        return ans
```

91、解码方法

一条包含字母 `A-Z` 的消息通过以下映射进行了 **编码** ：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

要 解码 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，"11106" 可以映射为：

"AAJF" ，将消息分组为 (1 1 10 6)
"KJF" ，将消息分组为 (11 10 6)
注意，消息不能分组为 (1 11 06) ，因为 "06" 不能映射为 "F" ，这是由于 "6" 和 "06" 在映射中并不等价。

给你一个只含数字的 非空 字符串 s ，请计算并返回 解码 方法的 总数 。

题目数据保证答案肯定是一个 32 位 的整数。

> 输入：s = "12"
> 输出：2
> 解释：它可以解码为 "AB"（1 2）或者 "L"（12）。

> 输入：s = "226"
> 输出：3
> 解释：它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。

> 输入：s = "06"
> 输出：0
> 解释："06" 不能映射到 "F" ，因为字符串含有前导 0（"6" 和 "06" 在映射中并不等价）。

题解

1、动态规划

若前 i 个字符**可以解码**，dp(i) 表示前 i 个字符可以解码的方法数。

```
def numDecodings(self, s: str) -> int:
    if s.startswith('0'):  # 开头有 '0' 直接返回
        return 0

    n = len(s)
    dp = [1] * (n+1)  # 重点是 dp[0], dp[1] = 1, 1

    for i in range(2, n+1):
        if s[i-1] == '0' and s[i-2] not in '12':  # 出现前导 '0' 的情况，不能解码，直接返回
            return 0
        if s[i-2:i] in ['10', '20']:  # 只有组合在一起才能解码
            dp[i] = dp[i-2]
        elif '10' < s[i-2:i] <= '26': # 两种解码方式
            dp[i] = dp[i-1] + dp[i-2]
        else:                         # '01'到 '09' 或 > '26'，只有单独才能解码
            dp[i] = dp[i-1]
    return dp[n]
```

2、动态规划

对于字符串 s 的某个位置 i 而言，我们只关心「位置 i 自己能否形成独立 item 」和「位置 i 能够与上一位置（i-1）能否形成 item」，而不关心 i-1 之前的位置。

对于字符串 s 的任意位置 i 而言，其存在三种情况：

1. 只能由位置 i 的单独作为一个 item，设为 a，转移的前提是 a 的数值范围为 [1,9]，转移逻辑为 f[i] = f[i - 1]。
2. 只能由位置 i 的与前一位置（i-1）共同作为一个 item，设为 b，转移的前提是 b 的数值范围为 [10,26]，转移逻辑为 f[i] = f[i - 2]。
3. 位置 i 既能作为独立 item 也能与上一位置形成 item，转移逻辑为 f[i] = f[i - 1] + f[i - 2]。

$f[i]=f[i-1], 1 \leqslant a \leq 9$

$f[i]=f[i-2], 10 \leqslant b \leqslant 26$

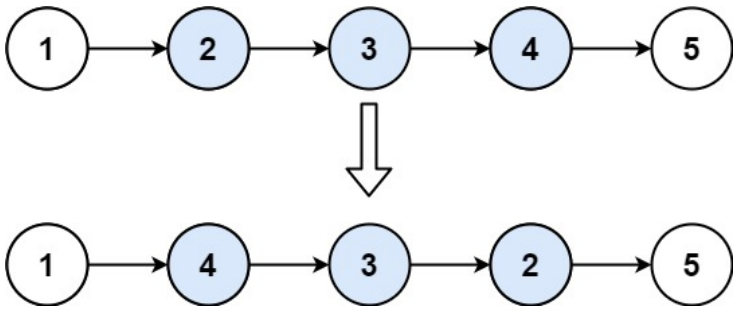
$f[i]=f[i-1]+f[i-2], 1 \leqslant a \leq 9, 10 \leqslant b \leqslant 26$

```
class Solution:
    def numDecodings(self, s: str) -> int:
        n = len(s)
        s = ' ' + s
        f = [0] * (n + 1)
        f[0] = 1
        for i in range(1,n + 1):
            a = ord(s[i]) - ord('0')
            b = ( ord(s[i - 1]) - ord('0') ) * 10 + ord(s[i]) - ord('0')
```

```
            if 1 <= a <= 9:
                f[i] = f[i - 1]
            if 10 <= b <= 26:
                f[i] += f[i - 2]
        return f[n]
```

92、反转链表II

给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表 。



> 输入：head = [1,2,3,4,5], left = 2, right = 4
> 输出：[1,4,3,2,5]

> 输入：head = [5], left = 1, right = 1
> 输出：[5]

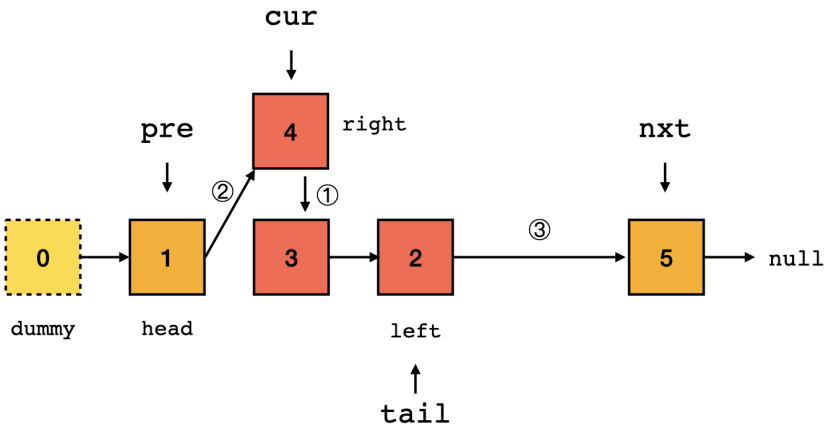
题解

1、转成数组再存成链表

```
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        l,arr,r=[],[],[]
        cur=head
        for _ in range(left-1):
            l.append(cur.val)
            cur=cur.next
        for _ in range(right-left+1):
            arr.append(cur.val)
            cur=cur.next
        while cur:
            r.append(cur.val)
            cur=cur.next

        res=ListNode(None)
        cur=res
        for a in l+arr[::-1]+r:
            cur.next=ListNode(a)
            cur=cur.next

        return res.next
```

2、指针反转

1. 指向 left 左边元素的指针 pre ，它表示未翻转的链表，需要把当前要翻转的链表结点放到 pre 之后。
2. cur 指向当前要翻转的链表结点。
3. nxt 指向 cur.next ，表示下一个要被翻转的链表结点。
4. tail 指向已经翻转的链表的结尾，用它来把已翻转的链表和剩余链表进行拼接



制图：负雪明烛

翻转 4

- cur 和 nxt 指针右移表示要翻转下一个节点
- tail 标记的是已翻转链表的结尾，一直不用动
- 需要把当前结点 4 插入到 pre 之后

```
class Solution(object):
    def reverseBetween(self, head, left, right):
        count = 1
        dummy = ListNode(0)
        dummy.next = head
        pre = dummy
        while pre.next and count < left:
            pre = pre.next
            count += 1
        cur = pre.next
        tail = cur
        while cur and count <= right:
            nxt = cur.next
            cur.next = pre.next
            pre.next = cur
            tail.next = nxt
            cur = nxt
            count += 1
        return dummy.next
```

31-40

93、复原IP地址

给定一个只包含数字的字符串，用以表示一个 IP 地址，返回所有可能从 s 获得的 有效 IP 地址 。你可以按任何顺序返回答案。

有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔。

例如："0.1.2.201" 和 "192.168.1.1" 是 有效 IP 地址，但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是 无效 IP 地址。

> 输入：s = "25525511135"
> 输出：["255.255.11.135","255.255.111.35"]

> 输入：s = "010010"
> 输出：["0.10.0.10","0.100.1.0"]

> 输入：s = "101023"
> 输出：["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]

题解

1、暴力法：把所有出现可能都列举出来,看是否满足条件.

```
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        n = len(s)
        res = []
        # 判读是否满足ip的条件
        def helper(tmp):
            if not tmp or (tmp[0] == "0" and len(tmp) > 1) or int(tmp) > 255:
                return False
            return True
        # 三个循环,把数字分成四份
        for i in range(3):
            for j in range(i + 1, i + 4):
                for k in range(j + 1, j + 4):
                    if i < n and j < n and k < n:
                        tmp1 = s[:i + 1]
                        tmp2 = s[i + 1:j + 1]
                        tmp3 = s[j + 1:k + 1]
                        tmp4 = s[k + 1:]
                        # print(tmp1, tmp2, tmp3, tmp4)

                        if all(map(helper, [tmp1, tmp2, tmp3, tmp4])):
                            res.append(tmp1 + "." + tmp2 + "." + tmp3 + "." +
tmp4)
        return res
```

2、回溯+剪枝

```
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
```

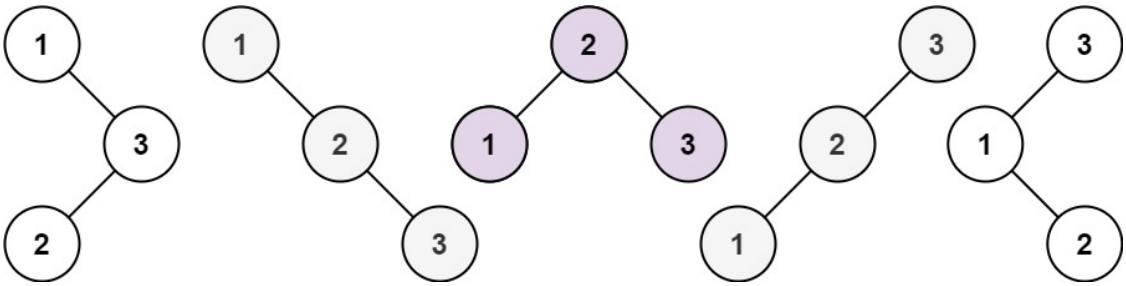
```
        #结果存储函数
        result = []
        #当前路径
        path = []
        #回溯函数
        def back_track(s, index):
            #减枝，如果搜索路径大于4，直接返回
            if len(path) > 4:
                return
            #全部搜素完成，搜索路径等于4，则加入结果列表
            if index == len(s) and len(path) == 4:
                result.append(".".join(path))
                return
            #遍历整个字符串，对每一个满足的子串递归回溯
            for i in range(index, len(s)):
                #减枝，如果当前值在0-255之前，则开始回溯
                if 0 <= int(s[index : i+ 1]) <= 255:
                    #如果当前值是0，但是不是一个单"0"则剪掉
                    if int(s[index : i+ 1]) == 0 and i != index:
                        continue
                    #如果当前值不是0，但是缺以"0XXX"开头，也应该剪掉
                    if int(s[index : i+ 1]) > 0 and s[index] == "0":
                        continue
                    #加入当前path
                    path.append(s[index: i+ 1])
                    #从当前节点开始递归
                    back_track(s, i + 1)
                    #回溯
                    path.pop()
        back_track(s, 0)
        return result
```

3、简单回溯

```
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        r = []
        def restore(count=0, ip='', s=''): # count record split times, ip record
ip, s record remaining string
            if count == 4:
                if s == '':
                    r.append(ip[:-1])
                return
            if len(s) > 0:
                restore(count+1, ip+s[0]+'.', s[1:])
            if len(s) > 1 and s[0] != '0':
                restore(count+1, ip+s[:2]+'.', s[2:])
            if len(s) > 2 and s[0] != '0' and int(s[0:3]) < 256:
                restore(count+1, ip+s[:3]+'.', s[3:])
        restore(0, '', s)
        return r
```

95、不同的二叉搜索树II

给你一个整数 `n` ，请你生成并返回所有由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的不同 **二叉搜索树** 。可以按 **任意顺序** 返回答案。



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

题解

1、递归

1. 特判，若n==0，返回[][]
2. 定义生成树函数build_Trees(left,right)build_Trees(left,right)，表示生成[left,...,right]的所有可能二叉搜索树：
   - 若left>right，说明为空树，返回[None]
   - 初始化所有可能的二叉搜索树列表all_trees=[]
   - 遍历每一种可能的节点i，遍历区间[left,right+1):
     - 所有可能的左子树列表left_trees=build_Trees(left,i-1)
     - 所有可能的右子树列表right_trees=build_Trees(i+1,right)
     - 组合所有的方式，遍历左子树列表,l：遍历右子树列表,r
       - 建立当前树节点cur_tree=TreeNode(i)
       - 将cur_tree左子树置为l
       - 将cur_tree右子树置为r
       - 将cur_tree加入树列表中
   - 返回树列表all_tree
3. 返回build_Trees(1,n)

```
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if(n==0):
            return []
        def build_Trees(left,right):
            all_trees=[]
            if(left>right):
                return [None]
            for i in range(left,right+1):
                left_trees=build_Trees(left,i-1)
                right_trees=build_Trees(i+1,right)
                for l in left_trees:
                    for r in right_trees:
```

```
                        cur_tree=TreeNode(i)
                        cur_tree.left=l
                        cur_tree.right=r
                        all_trees.append(cur_tree)
            return all_trees
        res=build_Trees(1,n)
        return res
```

2、动态规划

1. 定义dp数组，`dp[i][j]`存储数字为i---j之间的所有可能的情况,dp为三维数组
2. 规定初始值，如果i==j,表示只有一个节点的情况，添加节点val为i即可，i>j初始化为空列表，其他情况初始化为[None]，画一个矩阵就可以看出来了。
3. 确定dp方程，求`dp[i][j]`,以i-j之间数字为顶点的所有情况相加即可，例如，求`dp[1][3]`，分别以1,2,3为顶点，以1为顶点，左边值更大，右边值更小，
   左子树为`dp[1][0]`，右子树为`dp[2][3]`，两个树排列组合就可以,`dp[1][0]`为None,
4. 确定循环顺序，画矩阵可以看出来，从下向上进行循环，求`dp[1][3]`必须要知道`dp[2][3]`，求`dp[2][4]`必须要知道`dp[3][4]`,`dp[2][3]`，
5. 写代码时候注意边界情况需要额外判断

```
class Solution:
    def generateTrees(self, n: int):
        if n == 0:
            return None
        # 对dp进行初始化
        dp = []
        for i in range(0, n+1):    # 初始化dp
            dp.append([])
            for j in range(0, n+1):
                if i == j:
                    dp[i].append([TreeNode(i)])
                elif i < j:
                    dp[i].append([])
                else:
                    dp[i].append([None])
        dp[0][0] = [None]
        for i in range(n-1, 0, -1):  # 自下向上进行循环
            for j in range(i+1, n+1):
                for r in range(i, j+1):   # i-j每一个节点为顶点的情况
                    left = r+1 if r < j else r    # 右边的值需要边界判断，不然会溢出数
组
                    for x in dp[i][r-1]:          # 左右子树排列组合
                        for y in dp[left][j]:
                            node = TreeNode(r)
                            node.left = x
                            node.right = y
                            if r == j:
                                node.right = None
                            dp[i][j].append(node)      # dp[i][j]添加此次循环的值
        return dp[1][n]
```

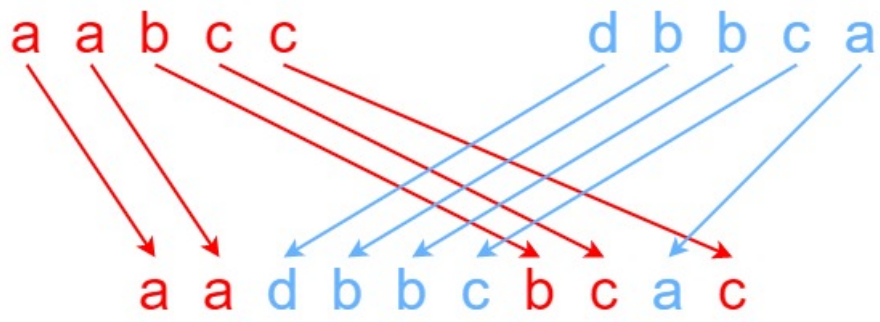
95、交错字符串

给定三个字符串 s1、s2、s3，请你帮忙验证 s3 是否是由 s1 和 s2 交错 组成的。

两个字符串 s 和 t 交错 的定义与过程如下，其中每个字符串都会被分割成若干 非空 子字符串：

- s = s1 + s2 + ... + sn
- t = t1 + t2 + ... + tm
- |n - m| <= 1
- 交错 是 s1 + t1 + s2 + t2 + s3 + t3 + ... 或者 t1 + s1 + t2 + s2 + t3 + s3 + ...

提示：a + b 意味着字符串 a 和 b 连接。




```
输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出：true
```

```
输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
输出：false
```

```
输入：s1 = "", s2 = "", s3 = ""
输出：true
```

题解

1、动态规划

	""	d	b	b	c	a
""	T	F	F	F	F	F
a	T	F	F	F	F	F
a	T	T	T	T	T	F
b	F	T	T	F	T	F
c	F	F	T	T	T	T
c	F	F	F	T	F	T
	s_3="aadbbcbcac"					

1. 初始化 s1,s2,s3 的长度分别为 len1,len2,len3
2. 若 len1+len2!=len3，表示一定不能构成交错字符串，返回 False
3. 初始化 dp 为 (len1+1)*(len2+1) 的 False 数组。
4. 初始化第一列 `dp[i][0]`，遍历第一列，遍历区间 [1,len1+1)：
    - `dp[i][0]=dp[i-1][0] and s1[i-1]==s3[i-1]`。表示 s1 的前 i 位是否能构成 s3 的前 i 位。因此需要满足的条件为，前 i-1 位可以构成 s3 的前 i-1 位且 s1 的第 i 位（s1[i-1]）等于

s3 的第 i 位（s3[i-1]）

5. 初始化第一行 `dp[0][j]`，遍历第一行，遍历区间 [1,len2+1):
    - `dp[0][i]=dp[0][i-1] and s2[i-1]==s3[i-1]`。表示 s2 的前 i 位是否能构成 s3 的前 i 位。因此需要满足的条件为，前 i-1位可以构成 s3 的前 i−1 位且 s2 的第 i 位（s2[i-1]）等于 s3 的第 i 位（s3[i-1]）
6. 遍历 dp 数组，每一行 i，遍历区间 [1,len1+1):
    - 每一列 j，遍历区间 [1,len2+1):
        - `dp[i][j]=(dp[i][j-1] and s2[j-1]==s3[i+j-1]) or (dp[i-1][j] and s1[i-1]==s3[i+j-1])`。解释：$s1$ 前i 位和 $s2$ 的前 j 位能否组成 $s3$ 的前 $i+j$ 位取决于两种情况：
        - s1 的前 i 个字符和 s2 的前 j-1 个字符能否构成 s3 的前 i+j-1 位，且 s2 的第 j 位（s2[j-1]）是否等于 s3 的第 i+j 位（s3[i+j-1]）。
7. 返回 `dp[-1][-1]`

```python
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        len1=len(s1)
        len2=len(s2)
        len3=len(s3)
        if(len1+len2!=len3):
            return False
        dp=[[False]*(len2+1) for i in range(len1+1)]
        dp[0][0]=True
        for i in range(1,len1+1):
            dp[i][0]=(dp[i-1][0] and s1[i-1]==s3[i-1])
        for i in range(1,len2+1):
            dp[0][i]=(dp[0][i-1] and s2[i-1]==s3[i-1])
        for i in range(1,len1+1):
            for j in range(1,len2+1):
                dp[i][j]=(dp[i][j-1] and s2[j-1]==s3[i+j-1]) or (dp[i-1][j] and
s1[i-1]==s3[i+j-1])
        return dp[-1][-1]
```

2、DFS

```python
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        import functools
        n1 = len(s1)
        n2 = len(s2)
        n3 = len(s3)
        @functools.lru_cache(None)
        def helper(i, j, k):
            if i == n1 and j == n2 and k == n3:
                return True
            if k < n3:
                if i < n1 and s1[i] == s3[k] and helper(i+1, j, k+1):
                    return True
                if j < n2 and s2[j] == s3[k] and helper(i, j+1, k+1):
                    return True
            return False
        return helper(0,0,0)
```

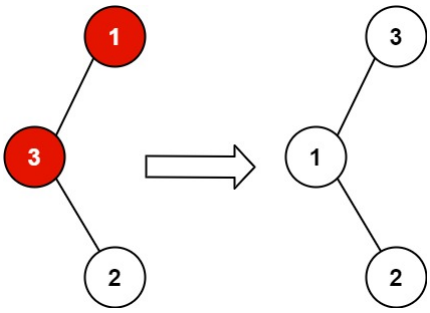
3、BFS

```python
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        from collections import deque
        n1 = len(s1)
        n2 = len(s2)
        n3 = len(s3)
        if n1 + n2 != n3: return False

        queue = deque()
        queue.appendleft((0, 0))
        visited = set()
        while queue:
            i, j = queue.pop()
            if i == n1 and j == n2:
                return True
            if i < n1 and s1[i] == s3[i + j] and (i + 1, j) not in visited:
                visited.add((i + 1, j))
                queue.appendleft((i + 1, j))
            if j < n2 and s2[j] == s3[i + j] and (i, j + 1) not in visited:
                visited.add((i, j + 1))
                queue.appendleft((i, j + 1))
        return False
```

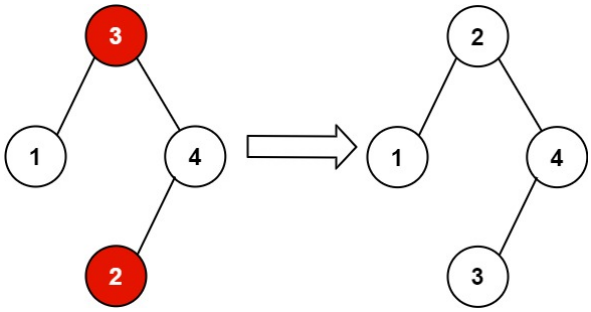
99、恢复二叉搜索树

给你二叉搜索树的根节点 root ， 该树中的两个节点被错误地交换。请在不改变其结构的情况下，恢复这棵树。

进阶：使用 O(n) 空间复杂度的解法很容易实现。你能想出一个只使用常数空间的解决方案吗？




```
输入：root = [1,3,null,null,2]
输出：[3,1,null,null,2]
解释：3 不能是 1 左孩子，因为 3 > 1 。交换 1 和 3 使二叉搜索树有效。
```



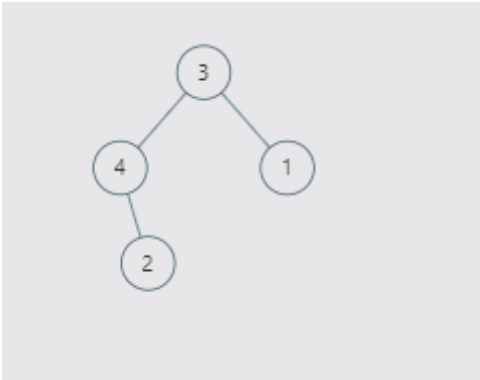
> 输入：root = [3,1,4,null,null,2]
> 输出：[2,1,4,null,null,3]
> 解释：2 不能在 3 的右子树中，因为 2 < 3 。交换 2 和 3 使二叉搜索树有效。

题解

1、递归

二叉树搜索树的中序遍历(中序遍历遍历元素是递增的)

如下图所示，中序遍历顺序是 4,2,3,1，我们只要找到节点 4 和节点 1 交换顺序即可！



- 这里我们有个规律发现这两个节点：
    1. 第一个节点，是第一个按照中序遍历时候前一个节点大于后一个节点，我们选取前一个节点，这里指节点 4；
    2. 第二个节点，是在第一个节点找到之后，后面出现前一个节点大于后一个节点，我们选择后一个节点，这里指节点 1；

```
class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        self.firstNode = None
        self.secondNode = None
        self.preNode = TreeNode(float("-inf"))

        def in_order(root):
            if not root:
                return
            in_order(root.left)
            if self.firstNode == None and self.preNode.val >= root.val:
                self.firstNode = self.preNode
            if self.firstNode and self.preNode.val >= root.val:
                self.secondNode = root
            self.preNode = root
            in_order(root.right)

        in_order(root)
        self.firstNode.val, self.secondNode.val = self.secondNode.val,
self.firstNode.val
```

2、迭代

```python
class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        firstNode = None
        secondNode = None
        pre = TreeNode(float("-inf"))

        stack = []
        p = root
        while p or stack:
            while p:
                stack.append(p)
                p = p.left
            p = stack.pop()

            if not firstNode and pre.val > p.val:
                    firstNode = pre
            if firstNode and pre.val > p.val:
                #print(firstNode.val,pre.val, p.val)
                secondNode = p
            pre = p
            p = p.right
        firstNode.val, secondNode.val = secondNode.val, firstNode.val
```

3、中序遍历

1. 中序遍历将树转成集合
2. 集合排序
3. 对比集合
4. 交换值

```python
class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        Trees = lambda x: [] if not x else Trees(x.left) + [x] + Trees(x.right)
        a = Trees(root)
        sa = sorted(a, key=lambda x: x.val)
        p, q = [a[i] for i in range(len(a)) if a[i] != sa[i]]
        p.val, q.val = q.val, p.val
```

103、二叉树的锯齿形层序遍历

给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

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

返回锯齿形层序遍历如下：

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

题解

1、层序遍历，分层倒序输出

```
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:return []
        res=[[]]
        stack=[[root,0]]
        while stack:
            roo,n=stack.pop()
            res.append([])
            if roo.left:stack.append([roo.left,n+1])
            if roo.right:stack.append([roo.right,n+1])
            res[n].append(roo.val)
        r=[]
        for i in range(len(res)):
            if len(res[i])==0:
                break
            if i%2==0:
                r.append(res[i][::-1])
            else:
                r.append(res[i])

        return r
```

2、根据层数，判断奇偶，决定是尾插还是头插

```
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        result = []
        que = collections.deque()
        que.append(root)
        k = 0
        while que:
            tmp = []
            k += 1
            for i in range(len(que)):
```

```
                node = que.popleft()
                if k % 2 != 0:
                    tmp.append(node.val)
                else:
                    tmp.insert(0, node.val)
                if node.left:
                    que.append(node.left)
                if node.right:
                    que.append(node.right)
            result.append(tmp)
        return result
```

递归写法

```
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        res = []

        def helper(root, depth):
            if not root: return
            if len(res) == depth:
                res.append([])
            if depth % 2 == 0:res[depth].append(root.val)
            else: res[depth].insert(0, root.val)
            helper(root.left, depth + 1)
            helper(root.right, depth + 1)
        helper(root, 0)
        return res
```

106、从中序与后序遍历序列构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

> 中序遍历 inorder = [9,3,15,20,7]
> 后序遍历 postorder = [9,15,7,20,3]

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

题解

1、递归

1. 根据中序遍历的特性：

   根节点左侧必然是左子树，右侧必然是右子树

2. 根据后序遍历的特性：

最后一个元素必然是根节点

对根节点和左右子树清晰之后，我们就可以使用递归，调用递归构造当前节点的左子树，调用递归构造当前节点的右子树

注意：传入递归函数中的遍历区间就是上面图中的区间，比如构建左子树时，传入区间就是紫色范围，构建右子树时，传入递归函数的区间就是绿色范围。

```
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        # inorder和postorder的长度一致，所以判断哪个都行
        if not postorder:
            return None
        node = TreeNode(postorder[-1])
        idx = inorder.index(node.val)
        node.left = self.buildTree(inorder[:idx], postorder[:idx])
        node.right = self.buildTree(inorder[idx+1:], postorder[idx:-1])
        return node
```

2、哈希表+递归

```
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        def recur(root_pos, left_ptr, right_ptr):
            """
                root_pos: 当前子树在前序遍历中，根节点的位置
                left_ptr: 当前子树在中序遍历中，最左侧的位置
                right_ptr: 当前子树在中序遍历中，最右侧的位置
            """
            # 递归终止条件
            if left_ptr > right_ptr: return
            # 建立当前子树的根节点，
            node = TreeNode(postorder[root_pos])
            # 根据根节点在前序遍历中的位置，找到其在中序遍历中的位置索引
            idx = mem[node.val]
            # 构建左子树，其中right_ptr-idx指的是当前左子树的长度，root_pos-1是指左子树起点，相减即可得到右子树根节点
            node.left = recur(root_pos-1-right_ptr+idx, left_ptr, idx-1)
            # 构建右子树，
            node.right = recur(root_pos-1, idx+1, right_ptr)
            # 回溯返回
            return node

        # 创建哈希表，将中序遍历的数组存起来，等同于 将 inorder.index的时间复杂度降低为O(1)
```

```
        mem = {}
        for i, cur in enumerate(inorder):
            mem[cur] = i
        return recur(len(postorder)-1, 0, len(postorder)-1)
```

107、二叉树的层序遍历II

给定一个二叉树，返回其节点值自底向上的层序遍历。 （即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

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

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

题解

1、层序遍历，输出倒序

```
class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if not root:return []
        stack = collections.deque()
        stack.append(root)
        res=[]
        k=0
        while stack:
            temp=[]
            k+=1
            for i in range(len(stack)):
                node=stack.popleft()
                temp.append(node.val)
                if node.left:stack.append(node.left)
                if node.right:stack.append(node.right)
            res.append(temp)

        return res[::-1]
```

递归

```
class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
```

```
        res_list = [[root.val]]
        def bfs(n, node):
            if not (node.left or node.right):
                return
            if len(res_list) <= n:
                res_list.append([])
            if node.left:
                res_list[n].append(node.left.val)
                bfs(n + 1, node.left)
            if node.right:
                res_list[n].append(node.right.val)
                bfs(n + 1, node.right)
        bfs(1, root)
        return res_list[::-1]
```

109、有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

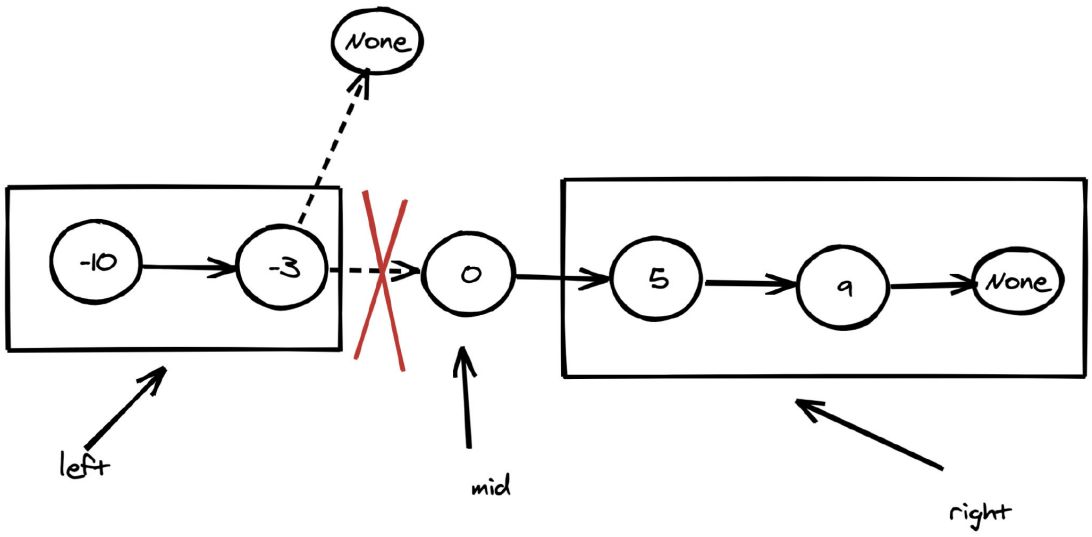
本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

> 给定的有序链表： [-10, -3, 0, 5, 9],
>
> 一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树：
>
> ```
>    0
>   / \
> -3   9
> /   /
> -10  5
> ```

题解

1、选择中点作为根节点，根节点左侧的作为左子树，右侧的作为右子树即可

同108，使用快慢指针找到中点

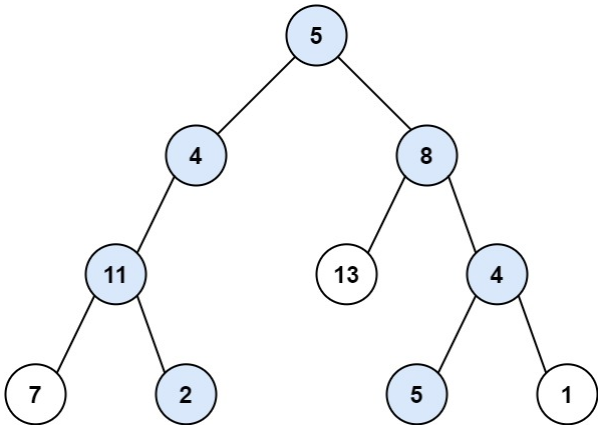


```python
class Solution:
    def sortedListToBST(self, head: ListNode) -> TreeNode:
        if not head:
            return head
        pre, slow, fast = None, head, head

        while fast and fast.next:
            fast = fast.next.next
            pre = slow
            slow = slow.next
        if pre:
            pre.next = None
        node = TreeNode(slow.val)
        if slow == fast:
            return node
        node.left = self.sortedListToBST(head)
        node.right = self.sortedListToBST(slow.next)
        return node
```

113、路径总和

给你二叉树的根节点 root 和一个整数目标和 targetSum ， 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。

叶子节点 是指没有子节点的节点。




```
输入：root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
输出：[[5,4,11,2],[5,8,4,5]]
```

题解

1、DFS，保存路径和路径和

```python
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) ->
List[List[int]]:
        if not root:return []
        res=[]
        def dfs(root,path,num):
            if not root.left and not root.right:
```

```
                if num+root.val==targetSum:
                    path.append(root.val)
                    res.append(path)
                    return
                else:
                    return
            if root.left:
                dfs(root.left,path+[root.val],num+root.val)
            if root.right:
                dfs(root.right,path+[root.val],num+root.val)

        dfs(root,[],0)
        return res
```

116、填充每个节点的下一个右侧节点指针

给定一个 **完美二叉树** ，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

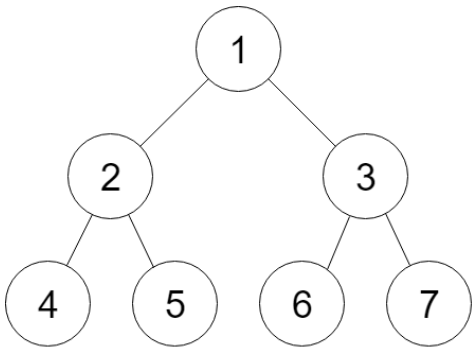


Figure A

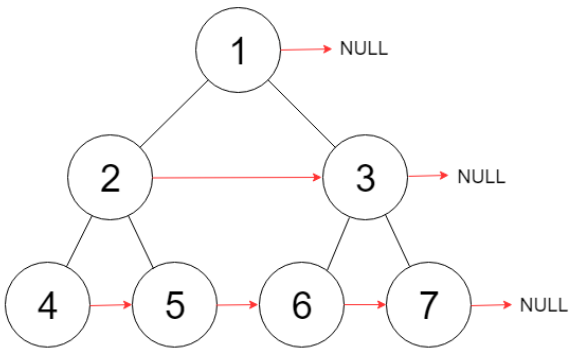


Figure B

```
输入：root = [1,2,3,4,5,6,7]
输出：[1,#,2,3,#,4,5,6,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化的输出按层序遍历排列，同一层节点由 next 指针连接，'#' 标志着每一层的结束。
```

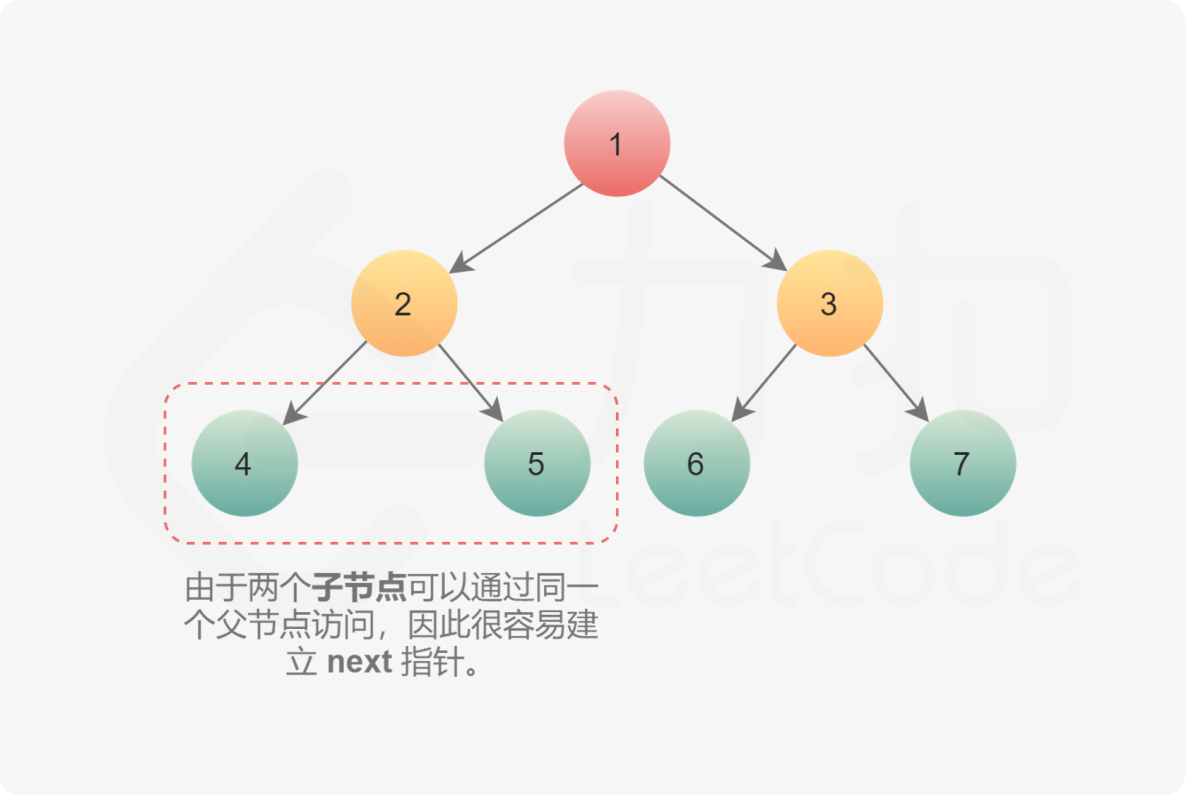
题解

1、层序遍历

```
class Solution:
```

```
    def connect(self, root: 'Node') -> 'Node':
        if not root:return root
        stack = collections.deque()
        stack.append(root)
        while stack:
            n=len(stack)
            for i in range(n):
                node=stack.popleft()
                if node.left:stack.append(node.left)
                if node.right:stack.append(node.right)
                if i<n-1:
                    node.next=stack[0]

        return root
```

2、使用已建立的next指针




```
class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return root
        # 从根节点开始
        leftmost = root
        while leftmost.left:
            # 遍历这一层节点组织成的链表，为下一层的节点更新 next 指针
            head = leftmost
            while head:
                # CONNECTION 1
                head.left.next = head.right
                # CONNECTION 2
                if head.next:
                    head.right.next = head.next.left
                # 指针向后移动
                head = head.next
```

```
            # 去下一层的最左的节点
            leftmost = leftmost.left

        return root
```

递归

```
class Solution:
    def connect(self, root: 'Node') -> 'Node':
        """简单递归"""
        if root:
            if root.left:
                root.left.next = root.right
                if root.next:
                    root.right.next = root.next.left
            self.connect(root.left)
            self.connect(root.right)
        return root
```

41-50

117、填充每个节点的下一个右侧节点指针II

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

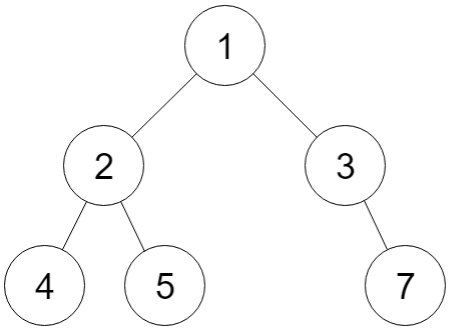


Figure A

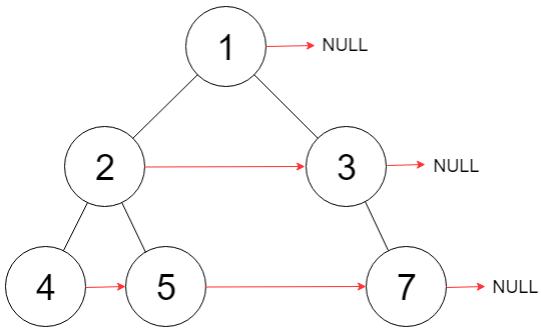


Figure B

```
输入：root = [1,2,3,4,5,null,7]
输出：[1,#,2,3,#,4,5,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，
如图 B 所示。序列化输出按层序遍历顺序（由 next 指针连接），'#' 表示每层的末尾。
```

题解

1、同上解法1

```
class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:return root
        stack = collections.deque()
```

```
        stack.append(root)
        while stack:
            n=len(stack)
            for i in range(n):
                node=stack.popleft()
                if node.left:stack.append(node.left)
                if node.right:stack.append(node.right)
                if i<n-1:
                    node.next=stack[0]

        return root
```

2、利用next指针

1. 利用next指针，将每层节点链接成链表进行遍历；
2. 创建dummyHead可以指向第一个节点；
3. 在遍历当前层的时候，处理下一层的节点；

```
class Solution:
    def connect(self, root):
        if not root:
            return None
        cur = root
        while cur:
            dummyHead = Node(-1)    #为下一行的之前的节点，相当于下一行所有节点链表的头结
点;
            pre = dummyHead
            while cur:
                if cur.left:        #链接下一行的节点
                    pre.next = cur.left
                    pre = pre.next
                if cur.right:
                    pre.next = cur.right
                    pre = pre.next
                cur = cur.next
            cur = dummyHead.next    #此处为换行操作，更新到下一行
        return root
```

120、三角形最小路径和

给定一个三角形 triangle ，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。也就是说，如果正位于当前行的下标 i ，那么下一步可以移动到下一行的下标 i 或 i + 1 。

```
输入：triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
输出：11
解释：如下面简图所示：
   2
  3 4
 6 5 7
4 1 8 3
自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。
```

题解

1、DFS——超时间

向下深度优先搜索，返回最小路径和

deep：深度，i：索引，num：路径和

```python
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        res=0
        def dfs(deep,i,num):
            if deep==len(triangle):
                return num
            num+=triangle[deep][i]
            left=dfs(deep+1,i,num)
            right=dfs(deep+1,i+1,num)

            return min(left,right)

        return dfs(0,0,0)
```

2、动态规划

从低向上

```python
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        dp = triangle
        for i in range(len(dp)-2, -1, -1):
            for j in range(len(dp[i])):
                dp[i][j] += min(dp[i+1][j], dp[i+1][j+1])
        return dp[0][0]
```

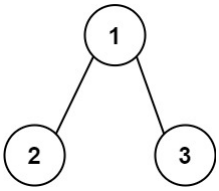
129、求根节点到叶节点数字之和

给你一个二叉树的根节点 root ， 树中每个节点都存放有一个 0 到 9 之间的数字。
每条从根节点到叶节点的路径都代表一个数字：

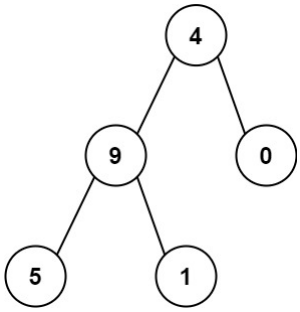
- 例如，从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123 。

计算从根节点到叶节点生成的 所有数字之和 。

叶节点 是指没有子节点的节点。




```
输入：root = [1,2,3]
输出：25
解释：
从根到叶子节点路径 1->2 代表数字 12
从根到叶子节点路径 1->3 代表数字 13
因此，数字总和 = 12 + 13 = 25
```




```
输入：root = [4,9,0,5,1]
输出：1026
解释：
从根到叶子节点路径 4->9->5 代表数字 495
从根到叶子节点路径 4->9->1 代表数字 491
从根到叶子节点路径 4->0 代表数字 40
因此，数字总和 = 495 + 491 + 40 = 1026
```

题解

1、DFS

保存路径字符，

```python
class Solution:
    def sumNumbers(self, root: TreeNode) -> int:
        res=[]
        def dfs(root,s):
            s+=str(root.val)
            if not root.left and not root.right:
                res.append(int(s))
                return
            if root.left:dfs(root.left,s)
            if root.right:dfs(root.right,s)

        dfs(root,'')
        return sum(res)
```

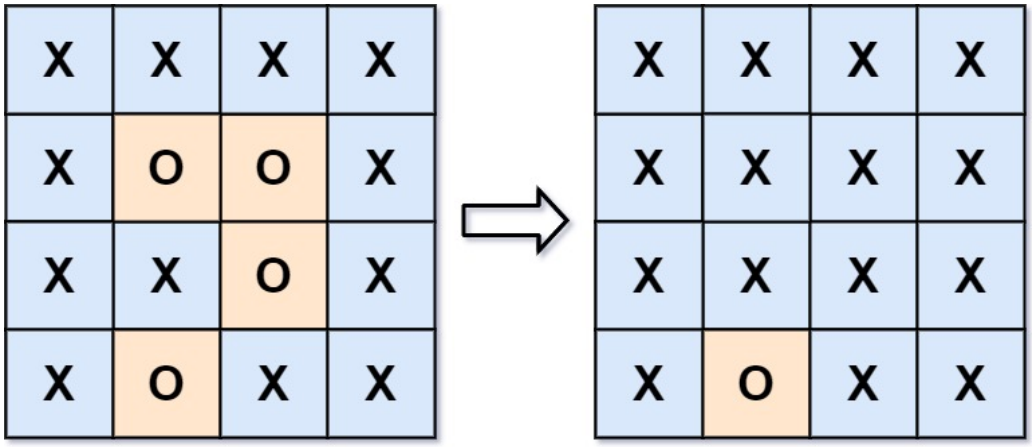
2、BFS

```python
class Solution:
    def sumNumbers(self, root: TreeNode) -> int:
        if not root:
```

```
            return 0
        queue = collections.deque([(root, root.val)])
        res = 0
        while queue:
            node, num = queue.popleft()
            if not node.left and not node.right:
                res += num
            if node.left:
                queue.append((node.left, num * 10 + node.left.val))
            if node.right:
                queue.append((node.right, num * 10 + node.right.val))
        return res
```

130、被围绕的区域

给你一个 `m x n` 的矩阵 `board` ，由若干字符 `'X'` 和 `'O'` ，找到所有被 `'X'` 围绕的区域，并将这些区域里所有的 `'O'` 用 `'X'` 填充。




```
输入：board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],
["X","O","X","X"]]
输出：[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],
["X","O","X","X"]]
解释：被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。
```

题解

1、BFS

任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。

那么，我们只要找到四周边界上存在O且与这些O连接着的O，在搜索时先修改成其他字母，比如“B”。然后遍历二维矩阵，将为O修改为X，将B修改为O即可。

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        if not board or not board[0]:
            return
        row = len(board)
```

```
        col = len(board[0])

        def bfs(i, j):
            from collections import deque
            queue = deque()
            queue.appendleft((i, j))
            while queue:
                i, j = queue.pop()
                if 0 <= i < row and 0 <= j < col and board[i][j] == "O":
                    board[i][j] = "B"
                    for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        queue.appendleft((i + x, j + y))

        for j in range(col):
            # 第一行
            if board[0][j] == "O":
                bfs(0, j)
            # 最后一行
            if board[row - 1][j] == "O":
                bfs(row - 1, j)

        for i in range(row):
            # 第一列
            if board[i][0] == "O":
                bfs(i, 0)
            # 最后一列
            if board[i][col - 1] == "O":
                bfs(i, col - 1)

        for i in range(row):
            for j in range(col):
                if board[i][j] == "O":
                    board[i][j] = "X"
                if board[i][j] == "B":
                    board[i][j] = "O"
```

2、DFS

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        if not board or not board[0]:
            return
        row = len(board)
        col = len(board[0])

        def dfs(i, j):
            board[i][j] = "B"
            for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                tmp_i = i + x
                tmp_j = j + y
                if 1 <= tmp_i < row and 1 <= tmp_j < col and board[tmp_i][tmp_j]
== "O":
                    dfs(tmp_i, tmp_j)

        for j in range(col):
            # 第一行
            if board[0][j] == "O":
```

```
                dfs(0, j)
            # 最后一行
            if board[row - 1][j] == "O":
                dfs(row - 1, j)

        for i in range(row):
            # 第一列
            if board[i][0] == "O":
                dfs(i, 0)
            # 最后一列
            if board[i][col-1] == "O":
                dfs(i, col - 1)

        for i in range(row):
            for j in range(col):
                # O 变成 X
                if board[i][j] == "O":
                    board[i][j] = "X"
                # B 变成 O
                if board[i][j] == "B":
                    board[i][j] = "O"
```

3、并查集

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        f = {}
        def find(x):
            f.setdefault(x, x)
            if f[x] != x:
                f[x] = find(f[x])
            return f[x]
        def union(x, y):
            f[find(y)] = find(x)

        if not board or not board[0]:
            return
        row = len(board)
        col = len(board[0])
        dummy = row * col
        for i in range(row):
            for j in range(col):
                if board[i][j] == "O":
                    if i == 0 or i == row - 1 or j == 0 or j == col - 1:
                        union(i * col + j, dummy)
                    else:
                        for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                            if board[i + x][j + y] == "O":
                                union(i * col + j, (i + x) * col + (j + y))
        for i in range(row):
            for j in range(col):
                if find(dummy) == find(i * col + j):
                    board[i][j] = "O"
                else:
                    board[i][j] = "X"
```

131、分割回文串

给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是 **回文串** 。返回 `s` 所有可能的分割方案。

回文串 是正着读和反着读都一样的字符串。

> 输入：s = "aab"
> 输出：[["a","a","b"],["aa","b"]]

> 输入：s = "a"
> 输出：[["a"]]

题解

1、回溯法

1. 未探索区域：剩余的未搜索的字符串 s；
2. 结束条件：s 为空；
3. 未探索区域当前可能的选择：每次选择可以选取 s 的 1 - length 个字符，cur = s[0...i]；
4. 当前选择符合要求：cur 是回文字符串 isPalindrome(cur)；
5. 新的未探索区域：s 去除掉 cur 的剩余字符串，s[i + 1...N]。

在每次搜索位置区域的时候，使用的是产生一个新数组 path + [s[:i]]，这样好处是方便：不同的路径使用的是不同的 path，因此不需要 path.pop() 操作；而且 res.append(path) 的时候不用深度拷贝一遍 path。

```
class Solution(object):
    def partition(self, s):
        self.isPalindrome = lambda s : s == s[::-1]
        res = []
        self.backtrack(s, res, [])
        return res

    def backtrack(self, s, res, path):
        if not s:
            res.append(path)
            return
        for i in range(1, len(s) + 1): #注意起始和结束位置
            if self.isPalindrome(s[:i]):
                self.backtrack(s[i:], res, path + [s[:i]])
```

2、动态规划

1. 临界分析

```
        if s == "":
            return []
```

2. 初始化，取第一个字母为输入时，所有的回文串

```
ans = [[s[0]] ]
```

3. 状态转移

    1. 直接在回文串组后添加新的字符
    2. 回文串组的最后一项为单字符且与新的字符相同
    3. 回文串组拥有两项以上且倒数第二串与新的字符相同

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        if s == "":
            return []
        ans = [[s[0]] ]
        for i in range(1, len(s)):
            curr = s[i]
            newAns = []
            for item in ans:
                newAns.append(item + [curr])
                if item[-1] == curr:
                    newAns.append(item[0:-1] + [item[-1] + curr])
                if len(item) >= 2 and item[-2] == curr:
                    newAns.append(item[0:-2] + [item[-2] + item[-1] + curr])
            ans = newAns
        return ans
```

134、加油站

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

1. 如果题目有解，该答案即为唯一答案。
2. 输入数组均为非空数组，且长度相同。
3. 输入数组中的元素均为非负数。

```
输入:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出: 3

解释:
从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。
```

```
输入:
gas  = [2,3,4]
cost = [3,4,3]

输出: -1

解释:
你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论怎样，你都不可能绕环路行驶一周。
```

题解

1、遍历判断是否可以走完一圈

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        for i in range(len(gas)):
            if gas[i]<cost[i]:
                continue
            target=i
            t=0
            while t>=0 and target<len(gas):
                t+=gas[target]-cost[target]
                target+=1
            if t<0:
                continue
            target=0
            while t>=0 and target<i:
                 t+=gas[target]-cost[target]
                 target+=1
            if target==i and t>=0:
                return i
            else:
                continue
        return -1
```

2、分析法

1. 然后开完一圈回来，如果耗得油总量比加的总量还多那铁定开不回来。
2. 假设从头开始开一遍，累计下来耗油最多的点的后面那个点开始出发，一定就是跑完以后剩下油最多的。
3. 比如我们跑到第i个点，这时候总消耗减去总加油最大，那说明前面i个点的消耗是整条链上最多的。然后如果总油数比总消耗多，说明可以保证开完一圈。
4. 那么如果想保证能开完一圈，就要从它后面那个点开始开。

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        left = [gas[i] - cost[i] for i in range(len(gas))]
        minValue = 999
        minIndex = 0
```

```
        sumValue = 0
        for j, l in enumerate(left):
            sumValue += l
            if sumValue < minValue:
                minValue = sumValue
                minIndex = j

        if sumValue < 0:
            return -1
        else:
            return (minIndex + 1) % len(gas)
```

优化

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        # total记录可获得的总油量-总油耗，  cur记录当前油耗情况，  ans记录出发位置
        total, cur, ans = 0, 0, 0
        for i in range(len(gas)):
            total += gas[i] - cost[i]
            cur += gas[i] - cost[i]
            if cur < 0:                     # 油不够开到i站
                cur = 0                     # cur置零，在新位置重新开始计算油耗情况
                ans = i + 1                 # 将起始位置改成i+1
        return ans if total >= 0 else -1    # 如果获得的汽油的量小于总油耗，则无法环
                                            # 行一周返回 -1；反之返回ans
```

137、只出现一次的数字II

给你一个整数数组 `nums` ，除某个元素仅出现 **一次** 外，其余每个元素都恰出现 **三次** 。请你找出并返回那个只出现了一次的元素。

> 输入：nums = [2,2,3,2]
> 输出：3

> 输入：nums = [0,1,0,1,0,1,99]
> 输出：99

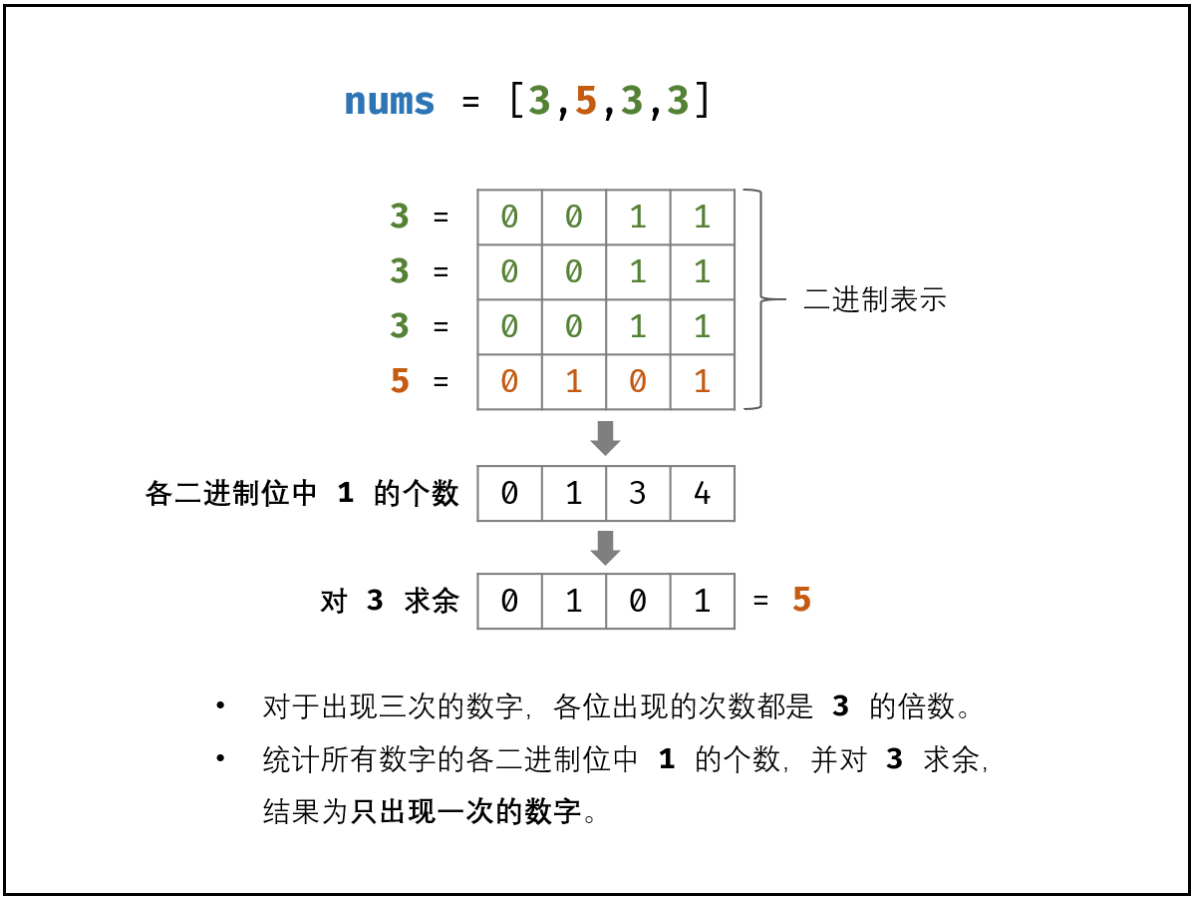
题解

1、数学公式

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

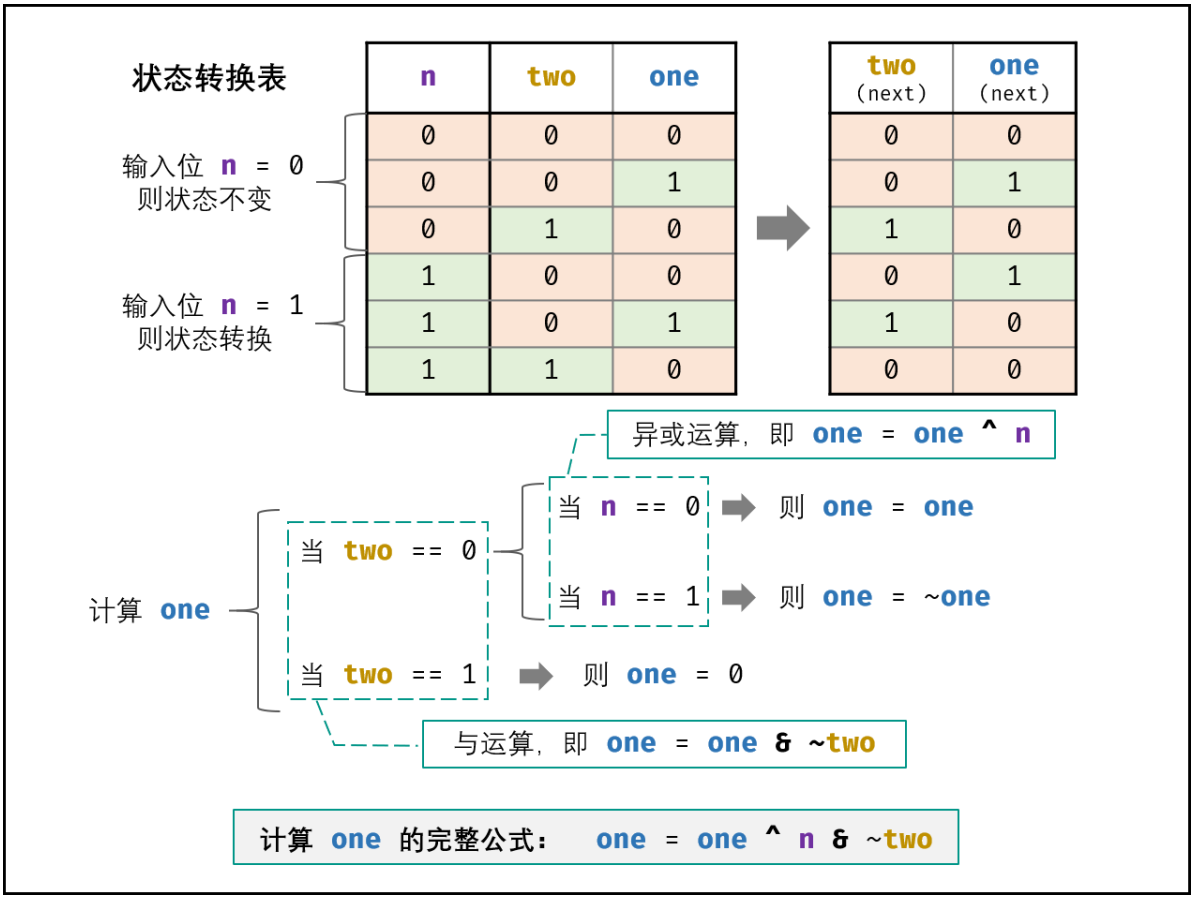
2、有限状态机

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

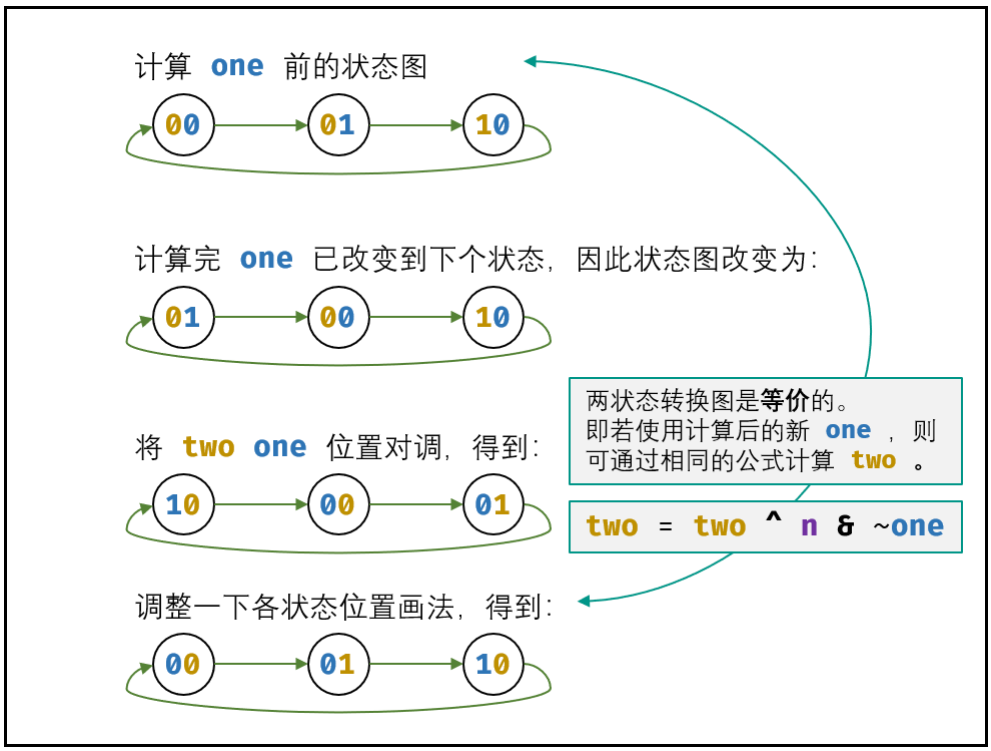


- 对于出现三次的数字，各位出现的次数都是 3 的倍数。
- 统计所有数字的各二进制位中 1 的个数，并对 3 求余，结果为**只出现一次的数字**。

计算 one 方法



计算 two方法




```python
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
```

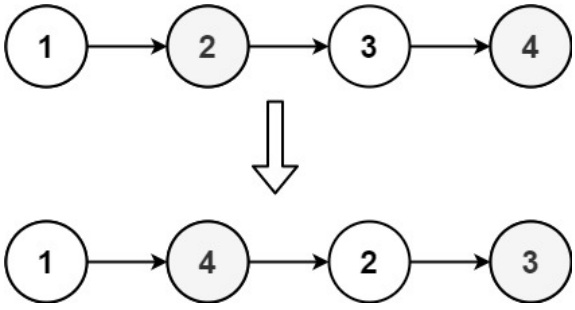
143、重排链表

给定一个单链表 L 的头节点 head ，单链表 L 表示为：

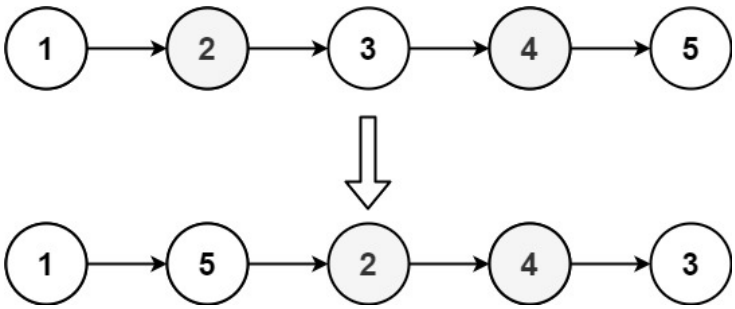
L0 → L1 → … → Ln-1 → Ln
请将其重新排列后变为：

L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → …

不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。




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




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

题解

1、转成数组再变链表

```
class Solution:
    def reorderList(self, head: ListNode) -> None:
        arr=[]
        cur=head
        while cur:
            arr.append(cur.val)
            cur=cur.next
        cur=head
        if len(arr)%2==1:
            for i in range(len(arr)//2):
                cur.next=ListNode(arr[-i-1])
                cur=cur.next
                cur.next=ListNode(arr[i+1])
                cur=cur.next
        else:
            for i in range(len(arr)//2-1):
                cur.next=ListNode(arr[-i-1])
                cur=cur.next
                cur.next=ListNode(arr[i+1])
                cur=cur.next
            cur.next=ListNode(arr[len(arr)//2])
        return head
```

2、指针，反转，拼接

```
class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head or not head.next:
            return head
        # 用快慢指针把链表一分为二，前半部分长度 >= 后半部分长度
        # first 为前半部分的头部，second 为后半部分的头部
        first = low = fast = head
        while fast.next and fast.next.next:
            fast, low = fast.next.next, low.next
        second, node, low.next, second.next = low.next, low.next.next, None,
None
        # 后半部分逆序
        while node:
            node.next, second, node = second, node, node.next
```

```
        # 前后部分交替链接
        while second:
            first.next, second.next, first, second = second, first.next,
first.next, second.next
```

150、逆波兰表达式求值

根据 逆波兰表示法，求表达式的值。

有效的算符包括 +、-、*、/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

- 平常使用的算式则是一种中缀表达式，如 (1 + 2) * (3 + 4) 。
- 该算式的逆波兰表达式写法为 ((1 2 +) (3 4 +) *) 。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 1 2 + 3 4 + * 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

> ```
> 输入：tokens = ["2","1","+","3","*"]
> 输出：9
> 解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9
> ```
>
> ```
> 输入：tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
> 输出：22
> 解释：
> 该算式转化为常见的中缀算术表达式为：
>   ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
> = ((10 * (6 / (12 * -11))) + 17) + 5
> = ((10 * (6 / -132)) + 17) + 5
> = ((10 * 0) + 17) + 5
> = (0 + 17) + 5
> = 17 + 5
> = 22
> ```

题解

1、保存数据，一步一步计算

```
import math
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack=[]
        for i in range(len(tokens)):
```

```
            if tokens[i] in ['+','-','*','/']:
                right=stack.pop()
                left=stack.pop()
                if tokens[i]=='+':
                    ru=left+right
                elif tokens[i]=='-':
                    ru=left-right
                elif tokens[i]=='*':
                    ru=left*right
                else:
                    if left/right<0:
                        ru=math.ceil(left/right)
                    else:
                        ru=math.floor(left/right)
                stack.append(ru)
            else:
                stack.append(int(tokens[i]))

        return int(stack[0])
```

2、eval表达式

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for item in tokens:
            if item not in {"+", "-", "*", "/"}:
                stack.append(item)
            else:
                first_num, second_num = stack.pop(), stack.pop()
                stack.append(
                    int(eval(f'{second_num} {item} {first_num}'))   # 第一个出来的在运算符后面
                )
        return int(stack.pop()) # 如果一开始只有一个数，那么会是字符串形式的
```

151、翻转字符串里的单词

给你一个字符串 s ， 逐个翻转字符串中的所有 单词 。

单词 是由非空格字符组成的字符串。s 中使用至少一个空格将字符串中的 单词 分隔开。

请你返回一个翻转 s 中单词顺序并用单个空格相连的字符串。

说明：

- 输入字符串 s 可以在前面、后面或者单词间包含多余的空格。
- 翻转后单词间应当仅用一个空格分隔。
- 翻转后的字符串中不应包含额外的空格。

> 输入：s = "the sky is blue"
> 输出："blue is sky the"

> 输入：s = " 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，但是翻转后的字符不能包括。

> 输入：s = "a good   example"
> 输出："example good a"
> 解释：如果两个单词间有多余的空格，将翻转后单词间的空格减少到只含一个。

题解

1、python

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

2、双指针

```
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
```