思路新颖

2、思路新颖：最多有多少个点在一条直线上

给出二维平面上的 n 个点，求最多有多少个点在同一条直线上。

我的思路：得求出一元一次方程，但是那样的话就会有多个一元一次方程，再一一比较。

解题思路：分别以数组中的点作为起始点，其后的点相对于起始点求斜率，斜率相等说明在同一条直线上，计数不同斜率的值的数目，取最大值，然后换一个起始点反复计算。这里需要注意斜率不能为 0，所以平行于 x 轴的线段需要另算。

```
def maxPoints(self, points):
    max_count = 0 # 最多有多少个点在一条直线上
    for i in range(len(points)):
        initNode_x = points[i][0]
        initNode_y = points[i][1]
        heng = 0 # 因为求斜率分母不能为0，所以要单独计算平行于x轴的线段
        xielvs = {}  # 计算斜率相等，也就是以该起始点的一条直线上的的点数
        for j in range(i + 1, len(points)):
            dx = points[j][0] - initNode_x
            dy = points[j][1] - initNode_y
            if dx == 0:
                heng += 1
            else:
                xielv = dy // dx
                if xielv not in xielvs:
                    xielvs[xielv] = 1
                else:
                    xielvs[xielv] += 1
        for x in xielvs.values():
            max_count = max(max_count, x, heng)
    return max_count + 1 # 加上起始点
```

3、思路新颖：分糖果

有 N 个小孩站成一列，每个小孩有一个评级。按照要求给小孩分糖果：1、每个小孩至少得到一颗糖果；2、评级越高的小孩比他相邻的两个小孩得到的糖越多。求出最少需要准备的糖果数。

我的思路：初始的时候都是都先给每个小孩一颗糖，然后从左往右遍历，如果右边的小孩比左边的评级高，则给右边的小孩在左边小孩的基础上加一颗糖，反之从右往左遍历亦然。

```python
def candy1(self, ratings):
    candy = [1 for _ in range(len(ratings))]
    for i in range(1, len(ratings)):
        if ratings[i - 1] < ratings[i]:
            candy[i] = candy[i - 1] + 1  # 这里只能在前一者基础上加1，而不能自加1
    for j in range(len(ratings) - 2, -1, -1):
        if ratings[j + 1] < ratings[j]:
            candy[j] = candy[j + 1] + 1
    return sum(candy)
```

4、思路新颖：硬币摆放

有 n 枚硬币，想要摆放成阶梯形状，即第 k 行恰好有 k 枚硬币。给出 n，找到可以形成的完整楼梯行数，n 是一个非负整数，且在 32 位有符号整数范围内。

我的思路：循环减，不满足条件时跳出。

```python
def arrangeCoins1(self, n):
    ca = n
    for i in range(1, n):
        ca -= i
        if ca < 0:
            return i - 1  # 返回上一层的层数
    return -1
```

解题思路：根据等差数列求和， n = (1 + x) * x / 2, 求解 x 的值，并向下取值。

```python
class Solution:
    def arrangeCoins(self, n):
        return math.floor((-1 + math.sqrt(1 + 8*n)) / 2)  # 根据等差数列求和，计算x的大小
```

5、思路新颖：硬币排成线 1

有 n 个硬币排成一条线，两个参赛者轮流从右边依次拿走 1 或 2 个硬币，直到没有硬币为止，拿到最后一枚硬币的人获胜，判断第一个玩家是输还是赢，若第一个玩家赢则返回 True，否则返回 False。

我的思路：当硬币数量是 3 的倍数的时候，后拿的赢，否则先拿的赢。

```python
class Solution:
    def firstWillWin(self, n):
        return bool(n % 3)  # 当n是3的倍数时返回False
```

6、思路新颖：移动 0 问题

给定一个数组，本例实现将 0 移动到数组的最后面，非 0 元素保持原数组的顺序。注意必须在原数组上操作，最小化操作数。

我的思路：利用冒泡排序的思想，将 0 排到最后。

解题思路：采用逆向思维，将所有的非 0 的数往前移动，直到找不到了 0，然后后面还剩几个位置，那么就补几个 0.

思路就是：left 指向非 0 数新位置，right 指向非 0 数的旧位置。

```python
class Solution:
    def moveZeroes(self, nums):
        left, right = 0, 0   # left就是非0数的新位置。right往后找非0数的旧位置。
        while right < len(nums):
            if nums[right] != 0:   # 找到非0数旧位置，赋值给新位置
                nums[left] = nums[right]
                left += 1   # 新位置加1，等待接收下一个非0数旧位置
            right += 1

        while left < len(nums):# 省下的位置补0.
            nums[left] = 0
            left += 1
        return nums
```

```
输入的整数数组是 ：  [0, 1, 0, 3, 12]
移动零后的数组是: [1, 3, 12, 0, 0]
```

7、思路新颖：木材加工

给定一些原木，把它们分割成一些长度相同的小段木头，需要得到小段的数目至少为 k，但是希望得到的小段越长越好，本例将设计能够得到小段木头的最大长度。

实例：三根木头【232、124、456】，k = 7，最大长度为 114

我的思路：将 n 根木头相加，除以 k，得到理想长度，看理想长度是否小于最小原木，然后按照理想长度去裁三根原木，看是否大于等于 k，将理想长度递减，知道满足情况，就是最佳长度。

解题思路：不是按照理想长度递减，而是采用二分法，以理想长度为最大值。

```python
class Solution:
    def woodCut(self, L, k):
        ad_length = sum(L) // len(L)
        while True:
            count = 0
            for x in L:
                count += x // ad_length
            if count >= k:# 如果切分后的段数大于等于k，那么跳出循环
                break
            ad_length -= 1
        return ad_length
```

8、思路新颖：判断平方数

判断一个数是否是完全平方数

```python
class Solution:
    def isPerfectSquare2(self, n):
        return int((n ** 0.5) ** 2) == n
        # 因为对n进行开方，如果不是完全平方数，将返回有限小数，按理来说应该是无限不循环小数，损失了精度，所以不可能相等

if __name__ == '__main__':
    solution = Solution()
    print(solution.isPerfectSquare2(9))
```

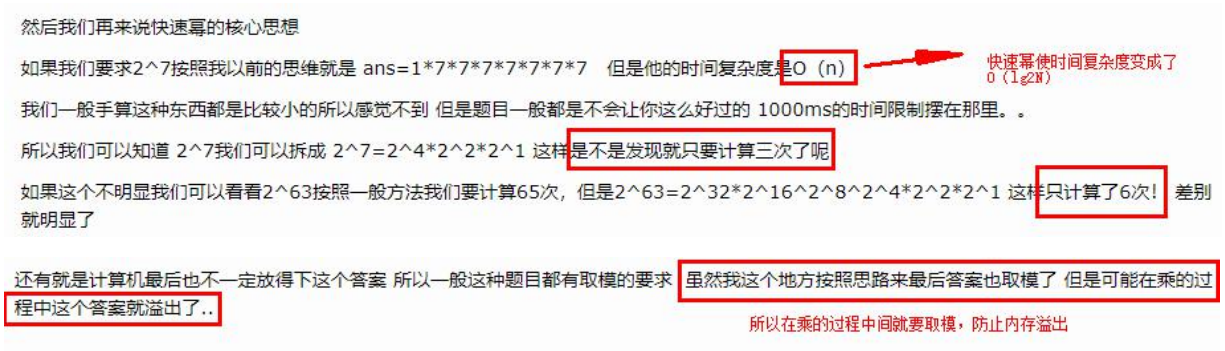
9、思路新颖：检测一个整数是否是 2 的倍数

```python
class Solution:
    def checkPowerOf2(self, n):
        while n % 2 == 0:  # 如果有2的因子，那么结果就会是0
            n = n // 2
        if n == 1:
            return True
        else:
            return False
```

如果n是2的幂次，那么把n中所有的2去除，留下的是1

10、二分回溯：快速幂

a 是底数，N 是指数，b 是取模数，求 a 的 n 次幂，时间复杂度为 O(log2N)




```python
class Solution:
    def fastPower(self, a, n):  # 递归求快速幂，求a的n次幂
        if (n == 1):  # n等于1时，幂为a
            return a
        temp = self.fastPower(a, n / 2)
        return (1 if n % 2 == 0 else a) * temp * temp  # 如果n是奇数，就会舍去一个a，所以要补上

if __name__ == '__main__':
    solution = Solution()
    print(solution.fastPower(2, 64))
```

还是递归简单，简单明了

为了防止内存溢出：

```python
class Solution:
    def fastPower(self, a, n, b):  # 递归求快速幂，求a的n次幂
        if (n == 1):  # n等于1时，幂为a
            return a
        temp = self.fastPower(a, n // 2, b) % b
        return (1 if n % 2 == 0 else a) * temp * temp % 3  # 如果n是奇数，就会舍去一个a，所以要补上

if __name__ == '__main__':
    solution = Solution()
    print(solution.fastPower(2, 31, 3))
```

快速幂之移位解法：

```python
class Solution:
    def Power(self, base, exponent):
        result = 1
        flag = 1 if exponent > 0 else -1
        exponent = exponent if flag == 1 else -exponent
        while exponent != 0:     # 把指数化成二进制，当指数中没有1的时候，结束循环
            if exponent & 1 == 1:# 举例:10^1101 = 10^0001*10^0100*10^1000。
                result *= base
            base = base * base
            exponent >>= 1    # 等价于 exponent //= 2
        return result if flag == 1 else 1 / result
```

11、思路新颖：完美平方

一、题目：

给定正整数 *n*，找到若干个完全平方数（比如 1，4，9，16，...）使得它们的和等于 *n*。你需要让组成和的完全平方数的个数最少。

二、思路：数学想法

- 四平方和定理：每个正整数都可以表示为至多4个正整数的平方和。故该题答案为1，2，3或4.
- 该数 n 若能整除4，则 n /= 4，结果不影响。【化简】
- 如果一个数除以8余7的话，那么肯定是由4个完全平方数组成。return 4
- 将其拆为两个平方数之和，如果拆成功了那么就会返回1或2，因为其中一个平方数可能为0，故判断拆的两个数是否为正整数

剩下的数都会是3

```python
class Solution:
    def numSquares(self, n):
        while n % 4 == 0:  # 4这个数字很特殊，它们多少无论多少个4想乘都是一个完全平方数
            n //= 4  # 向下取整（对于浮点数）
        if n % 8 == 7:
            return 4
        for i in range(n + 1):
            temp = i * i
            if temp <= n:
                if int((n - temp) ** 0.5) ** 2 + temp == n:
                    return 1 + (0 if temp == 0 else 1)
            else:
                break
        return 3


if __name__ == '__main__':
    solution = Solution()
    print(solution.numSquares(13))
```

因为两个完全平方数相乘还会是一个完全平方数，所以当数里面包含4的非0倍数时，都会作为所有其他完全平方数的公因子。
例如 10 = 3 * 3 + 1 * 1 = 9 + 1
40 = (3 * 3 + 1 * 1) * 4 = 36 + 4
结果还是两个，所以应该尽量去除数中的4，只要数里面取余包含4， 那么就要整除4，去除它。

当拆分的第一个数大于n时，拆分结束

当temp == 0， 也就是拆分的另一个数的平方正好等于n时， 返回1，其余情况都返回2

给定一个正整数 n，找到若干个完全平方数（例如 1,4,9），使得它们的和等于 n，完全平方数的个数最少：

我的思路：因为一个数至多有 4 个完全平方数组成。

1、去除数中的 4，因为 4 的任何倍数都是完全平方数，无碍个数的多少

2、如果取 8 的余数是 7，那么就是 4 (1,1,1,2)

3、取 2 和取 1 能够判断出来

4、省下的就是取 3.

方法 1：回溯

```python
def dfs(self, k, sum, ans, result):
    if sum == n:
        result.append(ans[:])
        return
    if sum > n:
        return
    if len(ans) > 4: # 因为可以取重复的数，所以需要限制长度
        return
    for i in range(k):
        sum += i * i
        ans.append(i)
        self.dfs(n, sum, ans, result)
        ans.pop()
        sum -= i * i
```

方法 2：直接写

```python
def numSquares2(self, n):
    while n % 4 == 0:  # 去除4的倍数
        n //= 4
    if n % 8 == 7:
        return 4
    for i in range(int(n ** 0.5) + 1):  # 保证本身可以取到
        temp = i * i
        temp2 = n - i * i
        if int(temp2 ** 0.5) ** 2 == temp2:  # 判断temp2是否是完全平方
            return 1 + (0 if temp == n or temp2 == 0 else 1)  # 如果其中有一个正好等于n，那么就返回1，否则返回2
    return 3
```

栈、队列

12、栈、队列：三元式解析器

给定一个表示任意嵌套三元表达式的字符串 expressions，本例将计算表达式的结果。可以假设给定的表达式是有效的，并且只由数字、T、F 组成（T、F 分别表示 True 和 False）。需要注意的是：

1、给定字符串的长度

2、每个整数都是个位数

3、条件表达式从右到左（跟大多数语言一样）

4、条件永远都是 T 或 F，不会是一个数字

5、表达式的结果总是对 0-9、T 或 F 求值。

我的思路：从右往左，将数字存入栈中，栈中只存两个数，遇到 T 或 F 就选其一，存入栈中，直到栈中只剩下一个数位置。

```python
def parseTernary(self, expression):
    stack = []
    i = len(expression) - 1
    while i >= 0:
        if expression[i] not in ['?', ':']:  # 遇到数字和T和F就加入栈
            stack.append(expression[i])
        if expression[i] == '?':  # 遇到问号就要开始计算
            if len(stack) <= 1:  # 防止出界
                return stack[-1]
            temp1 = stack.pop()
            temp2 = stack.pop()
            if expression[i - 1] == 'T':
                stack.append(temp1)
            if expression[i - 1] == 'F':
                stack.append(temp2)
            i -= 1  # 因为遇到的是问号，所以需要多走一步
        i -= 1
    return stack
```

13、栈、队列：用栈实现队列

使用两个栈实现队列的一些操作，队列应支持 push、pop 和 top 操作，其中 pop 可以弹出队列中的第一个（最前面）元素，pop 和 top 方法都返回第一个元素的值

我的思路：用一个栈实现顺向，用另一个栈实现反向。

```python
class Solution:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []
    def adjust(self):
        if len(self.stack2) == 0:
            while len(self.stack1) != 0:
                self.stack2.append(self.stack1.pop())
    def push(self, element):
        self.stack1.append(element)
    def top(self):
        self.adjust()
        return self.stack2[- 1]
    def pop(self):
        self.adjust()                    # 栈1不能实现，要栈2辅助
        return self.stack2.pop()  # 实现先进先出
```

14、栈、队列：用栈模拟汉诺塔问题

在经典的汉诺塔问题中，有 3 个塔和 N 个可用来堆砌成塔、不同大小的盘子。要求盘子必须按照从小到大的顺序从上往下堆（即任意一个盘子必须堆在比它的盘子上面）。同时，必须满足三个条件：

1、每次只能移动一个盘子

2、每次盘子从堆的顶部被移动后，只能置放于下堆中
3、每个盘子只能放在比它大的盘子上面。
本例写一段程序，将第一堆的盘子移动到最后一堆中。
普通汉诺塔问题：

```
class Solution():
    def move(self, A, B):
        print(A + '->' + B)

    def hanoi(self, n, A, B, C):# 将n个盘子经过B从A移到C
        if n == 1:
            self.move(A, C)
            return
        self.hanoi(n - 1, A, C, B)# 将n - 1个盘子经过C从A移到B
        self.move(A, C)
        self.hanoi(n - 1, B, A, C)# 将n - 1个盘子经过A从B移到C
```

```
a->c
a->b
c->b
a->c
b->a
b->c
a->c
```

当 n == 3，A，B，C = a, b, c

```
class Solution():
    def move(self, A, B):
        B.append(A.pop())

    def hanoi(self, n, A, B, C):# 将n个盘子经过B从A移到C
        if n == 1:
            self.move(A, C)
            return
        self.hanoi(n - 1, A, C, B)# 将n - 1个盘子经过C从A移到B
        self.move(A, C)
        self.hanoi(n - 1, B, A, C)# 将n - 1个盘子经过A从B移到C


if __name__ == "__main__":
    tow1, tow2, tow3 = [], [], []
    n = 3
    for i in range(n - 1, -1, -1):
        tow1.append(i)
    solution = Solution()
    solution.hanoi(n, tow1, tow2, tow3)
    print(tow3)
```

```
[2, 1, 0]
```

，只要改变 move 即可。

15、栈、队列：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]，并且从第 i 个加油站前往第 i + 1 个加油站需要消耗汽油 cost[i]。

有一辆油箱容量无限大的汽车，现在要从某一个加油站出发环绕环路一周，一开始油箱为空。求出可以环绕环路一周时出发的加油站编号，若不存在可行方案，则返回-1.

示例：现在有 4 个加油站，汽油量 gas[i] = [1,1,3,1]，环路旅行时消耗的汽油量 cost[i] = [2,2,1,1]，则出发的加油站编号为 2.

我的思路：用 queue 制作以第 i 个元素开头的队列。

```
def canCompleteCircuit(self, gas, cost):
    size = len(gas)
    result = []
    for i in range(size):
        gasQueue = collections.deque(gas)
        costQueue = collections.deque(cost)
        flag = True  # 刚开始假设都能通过
        for _ in range(i):  # 制作以第i个元素开头的队列
            temp1 = gasQueue.popleft()
            temp2 = costQueue.popleft()
            gasQueue.append(temp1)
            costQueue.append(temp2)
        carOil = 0
        for j in range(len(gas) - 1):
            carOil += gasQueue[j] - costQueue[j]
            if carOil < 0:
                flag = False  # 油不够，置为False
                break
        if flag:
            result.append(i)
    return result
```

16、栈、队列：滑动窗口内唯一元素数量和

给定一个数组和一个滑动窗口的大小，求每一个窗口内唯一元素的个数和。注意，当滑动窗口的大小大于数组长度时，可以认为窗口大小就是数组长度，即窗口不会滑动。

```python
import collections
class Solution:
    def slidingWindowUniqueElementsSum(self, nums, k):
        size = len(nums)
        sum = 0
        queue = collections.deque()
        for i in range(k): # k不到
            queue.append(nums[i])
        sum += self.weiyi(queue) # 第一个窗口额外计算
        for i in range(k, size): # 正好取到k
            queue.popleft()
            queue.append(nums[i])
            sum += self.weiyi(queue)
        return sum

    def weiyi(self, queue):
        sum = 0
        counts = collections.Counter(queue) #没想到我居然会想到用Counter进行计数，记得返回的是字典
        for key, val in counts.items():
            if counts[key] == 1:
                sum += val
        return sum
```

17、栈、队列：逆波兰表达式求值

求逆波兰表达式的值，在逆波兰表达式中，有效的运算符号包括+、-、*、/，每个对象既可以是整数，也可以是另一个逆波兰表达式。

我的思路：肯定需要用到栈，跟一般的还不同。

解题思路：当遇到数字时，往栈里面存入，当遇到运算符号时，弹出两个数字，进行计算，计算后的到的值再存入栈中，循环往复，知道栈中只剩下一个数，就是值。

```python
def evalRPN(self, tokens):
    stack = []
    for i in range(len(tokens)):
        if tokens[i] not in ['+', '-', '*', '/']:
            stack.append(tokens[i]) # 存储非数字
        else:
            if len(stack) == 1: # 因为要弹出两次，最好做一次判断
                return stack[-1]
            temp2 = int(stack.pop())  #两个操作数的顺序不要搞错了
            temp1 = int(stack.pop())
            if tokens[i] == '+':
                stack.append(temp1 + temp2)
            elif tokens[i] == '-':
                stack.append(temp1 - temp2)
            elif tokens[i] == '*':
                stack.append(temp1 * temp2)
            else:
                stack.append(temp1 // temp2)
    return stack[-1] if stack else -1
```

18、栈、队列：有效的括号序列

给定一个字符串所表示的括号序列，包含小括号，中括号和大括号，各种括号要相互配

对，问该字符串是否正确配对。

我的思路：建立一个栈，当遇到左括号（小、中、大）就入栈，当遇到右括号就弹栈，看弹栈出来的左括号是否和右括号匹配。

```
class Solution(object):
    def isValidParentheses(self, s):
        stack = []
        for i in range(len(s)):
            if s[i] == '(' or s[i] == '[' or s[i] == '{':
                stack.append(s[i])
            elif s[i] == ')':
                if stack.pop() != '(':
                    return False
            elif s[i] == ']':
                if stack.pop() != '[':
                    return False
            else:
                if stack.pop() != '{':
                    return False
        return True if len(stack) == 0 else False
```

19、栈、队列：棒球游戏

对于棒球比赛成绩记录，给定一个字符串数组，每一个字符串可以是以下 4 种中的其中一种：

1、整数，直接表示这个回合所得分数

2、‘+’，表叔这个回合的分数是前两个有效分数之和

3、‘D’，表示这个回合得到的分数是上一次获得的有效分数的两倍

4、‘C’，表示上个回合的有效分数是无效的，需要移除。

我的思路：像这种涉及到符号和整数的，一般而言，都要用到栈，用栈存入数字，遇到操作符，就根据操作符的操作进行操作。

```
    def calPoints(self, ops):
        stack = []
        for i in range(len(ops)):
            if ops[i] not in ['+', 'D', 'C']:
                stack.append(int(ops[i]))
            elif ops[i] == '+':
                if len(stack) <= 1:  # 防出界
                    return stack[-1]
                stack.append(stack[-1] + stack[-2])
            elif ops[i] == 'D':
                stack.append(stack[-1] * 2)
            else:
                stack.pop()
        return sum(stack)
```

20、栈、队列：滑动窗口的中位数

给定一个包含 n 个整数的数组和一个大小为 k 的滑动窗口，从左到右在数组中滑动这个窗口，找到数组中每个窗口内的中位数。注意，如果数组格式是偶数，则在该窗口排序数字后，返回第 N//2 个数字。

```
class Solution:
    def medianSlidingWindow(self, nums, k):
        size = len(nums)
        ans = []
        queue = collections.deque()
        for i in range(k):
            queue.append(nums[i])
        ans.append(sorted(queue)[k // 2])
        for i in range(k, size):
            queue.popleft()
            queue.append(nums[i])
            ans.append(sorted(queue)[k // 2])
        return ans
```

21、栈、队列：合并数字

给定 n 个数，将这 n 个数合并成一个数，每次只能选择两个数 a、b 合并，合并需要消耗的能量为 a+b，输出将这 n 个数合并成一个数后消耗的最小能量。

我的思路：明显每次挑选最小的两个数求和，然后再将和放回到列表中，再调出两个最小的值求和，再放回列表，循环反复，当只剩一个数时，就是 n 个数合并的最小能量。

明显即使是最小能量也会大于所有数的和，因为会进行重复的加法。

可以利用 heapq 结构，取出最小的两个数，将和加入 heapq 进行自动堆排序。

```
def mergeNumber1(self, numbers):
    heapq.heapify(numbers)  # 对numbers建立最小堆
    result = 0
    while len(numbers) > 1:
        first = heapq.heappop(numbers)  # 弹出numbers中最小值
        second = heapq.heappop(numbers)
        sum = first + second
        result += sum
        heapq.heappush(numbers, sum)  # 往最小堆numbers中插入sum重新建堆
    return result
```

但heapq里面没有直接提供建立大根堆的方法，可以采取如下方法：每次push时给元素加一个负号（即取相反数），此时最小值变最大值，反之亦然，那么实际上的最大值就可以处于堆顶了，返回时再取负即可。

```
a = []
for i in [1, 5, 20, 18, 10, 200]:
    heapq.heappush(a,-i)
print(list(map(lambda x:-x,a)))
```

复制

输出

```
[200, 18, 20, 1, 10, 5]
```

22、栈、队列：滑动窗口的最大值

给定一个可能包含重复整数的数组和一个大小为 k 的滑动窗口，从左到右在数组中滑动这个窗口，找到数组中每个窗口内的最大值。

我的思路：用 queue 初始化窗口为 k，然后从左边剔除一个元素，从右边加入一个元素，然后比较最大值。

```
class Solution:
    def maxSlidingWindow(self, nums, k):
        if not nums or not k:
            return []
        dq = deque()  # dq用来存储最大元素的索引坐标
        ans = []
        for i in range(k):
            dq.append(nums[i])
        ans.append(max(dq))
        for i in range(k, len(nums), 1):
            dq.popleft()
            dq.append(nums[i])
            ans.append(max(dq))
        return ans
```

23、栈、队列：丑数 1

丑数的定义是：只包含质因子 2、3、5 的正整数，例如 6、8 就是丑数，但 14 就不是丑数，因为它包含质因子 7，本例将检测一个整数是不是丑数。

我的思路：将整数里的所有 2 的倍数、3 的倍数、5 的倍数都去掉，如果剩下 1 就说明是丑数。注意，1、2、3、5 都是丑数。

```
class Solution:
    def isUgly(self, n):
        if n < 0:
            return False
        if n == 1:
            return True
        while n % 2 == 0: # 一直取余，如果等于0，说明还有该数的因子，所以要整除掉它
            n //= 2
        while n % 3 == 0:
            n //= 3
        while n % 5 == 0:
            n //= 5
        return n == 1
```

24、栈、队列：丑数 2

设计一个算法，找出只含素因子 2、3、5 的第 n 小的数，符合条件的数如：1、2、3、4、5、6、8、9、10、12.....

我的思路：按顺序找，但是效率低下。

解题思路：从 1 开始，存入 heapq，依次乘【2、3、5】，都存入 heapq，然后再取出最

小值，再依次成【2、3、5】，当取了 n 次时，就是第 n 小的数。

```
def nthUglyNumber2(self, n):
    minHeap = [1]
    heapq.heapify(minHeap)
    MIN = 0
    for i in range(n):
        MIN = heapq.heappop(minHeap)# 取n次最小值
        print(MIN)
        for x in [2, 3, 5]:
            if MIN * x not in minHeap:  # 一定要去重，因为总会存在公倍数。
                heapq.heappush(minHeap, MIN * x)# 建3 * n 次堆
    return MIN
```

25、栈、队列：超级丑数

超级丑数的定义是：所有质数因子都是给定一个大小为 k 的质数集合的正整数，例如，给出 4 个质数的集合【2,、7、13、19】，那么【1、2、4、7、8、13、14、16、19、26、28、32】是前 12 个超级丑数，本例将找出第 n 个超级丑数。

我的思路：上题的再运用。

```
import heapq
class Solution:
    def nthSuperUglyNumber(self, n, primes):
        minHeap = [1]
        heapq.heapify(minHeap)
        MIN = 0
        for i in range(n):
            MIN = heapq.heappop(minHeap)# 取n次最小值
            for x in primes:
                if MIN * x not in minHeap:  # 一定要去重，因为总会存在公倍数。
                    heapq.heappush(minHeap, MIN * x)# 建 len(minHeap) * n次堆
        return MIN
```

26、栈、队列：直方图中最大的矩阵面积

给出 n 个非负数表示每个直方图的高度，每个直方图的宽均为 1，在直方图中找到最大的矩形面积。

我的思路：首先明确要用到栈，存入栈里面的是直方图的索引，因为既可以通过索引找到高，又可以通过索引确定宽。

只要遇到比栈中最后一个元素高的直方图就将其索引存入栈中，遇到低的就弹栈，然后计算宽，比较最大面积。

```
def largestRectangleArea2(self, heights):
    heights.append(0)# 最后一个元素是最低的元素，那么就会弹出栈中所有的元素
    stack = []
    area = 0
    for i in range(len(heights)):
        while stack and heights[stack[-1]] > heights[i]:# 如果遇到了低的直方图，就要开始计算面积了
            high_index = stack.pop()  # 始终弹出最高的那个直方图的那个高
            high = heights[high_index]
            width = i - high_index   # 宽的话等于最高直方图索引和当前索引的差值
            print(width * high)
            area = max(area, width * high)
        stack.append(i)# 往stack加的永远是最高的那个元素
    return area
```

27、栈、队列：最大矩形

给定一个二维矩阵，元素取值 0 和 1，找到一个最大的矩形，使得其中的值全部为 1，输出它的面积。

我的思路：将数组给矩形化，当数组元素为 0 时，那就是 0；当数组元素是 1 时，那么就等于上一行元素的值加上该行的 1 的和，把前 n 行当做求最大直方图的最大矩形面积。

```
class Solution:
    def maximalRectangle2(self, matrix):
        rowNum, colNum = len(matrix), len(matrix[0])
        for i in range(1, rowNum):  # 从第二行开始
            for j in range(colNum):
                if matrix[i][j] == 1:
                    matrix[i][j] = matrix[i - 1][j] + 1
        area = 0
        for i in range(rowNum):
            area = max(area, self.maxAreaOfRow(matrix[i]))  # 前n行的最大直方图面积
        return area

    def maxAreaOfRow(self, A):
        stack, area = [], 0
        A.append(0)  # 栈的末尾是最小的元素，所以会弹出所有高的索引
        for i in range(len(A)):
            while stack and A[stack[-1]] > A[i]:
                higthest_index = stack.pop()
                higthest = A[higthest_index]
                width = i - higthest_index
                area = max(area, higthest * width)
            stack.append(i)
        return area
```

回溯递归

28、回溯递归：恢复 IP 地址

给定一个由数字组成的字符串，求出其所有可能的 IP 地址

我的思路：毫无疑问该用回溯，当字符串被切割成空时， 如果分割的部分小于 4 或大于 4，返回，并且长度为 4 时，记录并返回。

```python
class Solution:
    def restoreIpAddresses(self, s):
        ans, result, final = [], [], []
        self.dfs(s, ans, result)
        for x in result:
            final.append('.'.join(x))  # '任何字符'.join(字符数组)，那么将会以"任何字符"相连
        return final

    def dfs(self, s, ans, result):
        if s == "":
            if len(ans) == 4 and ans not in result:
                result.append(ans[:])
                return
            if len(ans) > 4 or len(ans) < 4:
                return

        for i in range(1, 4):
            integer = int(s[:i])
            if integer > 255:  # 大于了最大值，直接返回就好
                return
            if i > len(s):  # 如果剩下的字符串长度小于要截取的长度，直接返回
                return
            ans.append(s[:i])
            self.dfs(s[i:], ans, result)
            ans.pop()

# 主函数
if __name__ == '__main__':
    solution = Solution()
    S = "25525511135"
    print("字符串S是：", S)
    print(solution.restoreIpAddresses(S))
```

```
字符串S是： 25525511135
[['255', '255', '11', '135'], ['255', '255', '111', '35']]
['255.255.11.135', '255.255.111.35']
```

29、回溯递归：生成括号

给出 n 对括号，将这些括号任意组合，生成新的括号组合，并返回所有可能组合结果

我的思路：利用 permutation 函数对其进行全排列，看满足条件的有多少种。其中判断满足条件的函数如下：

```python
def isVaild(self, string):  # 诀窍就是遇到顺括号，就入栈，如果遇到反括号就出栈，最后如果栈为0，表示配对成功。
    stack = []
    for x in string:
        if x == ')':
            if len(stack) == 0:  # 如果第一个括号就是反括号，直接返回False
                return False
            else:
                stack.pop()
        else:
            stack.append(x)
    if len(stack) == 0:
        return True
    else:
        return False
```

```python
def generateParenthesis(self, n):
    List, ans = [], []
    for i in range(n):
        List.append('(')
        List.append(')')
    possible = permutations(List)
    for x in possible:
        if self.isVaild(x) and ''.join(x) not in ans:
            ans.append(''.join(x))
    return ans

def isVaild(self, string): # 诀窍就是遇到顺括号，就入栈，如果遇到反括号就出栈，最后如果栈为0，表示配对成功。
    stack = []
    for x in string:
        if x == ')':
            if len(stack) == 0: # 如果第一个括号就是反括号，直接返回False
                return False
            else:
                stack.pop()
        else:
            stack.append(x)
    if len(stack) == 0:
        return True
    else:
        return False
```

解题思路：回溯先放左括号还是先放右括号

1、左括号和右括号最多可放 n 个

2、当放置的左括号数小于右括号数，不可放置右括号。

```python
class Solution:
    def helpler(self, l, r, item, res):
        if r < l: # 放置的右括号数大于放置的左括号数，进行回溯
            return
        if r == 0 and l == 0: # 出口
            res.append(item[:])
        if l > 0:
            self.helpler(l - 1, r, item + '(', res) # 先放左括号
        if r > 0:
            self.helpler(l, r - 1, item + ')', res) # 先放右括号

    这里没有用循环回溯，而是枚举回溯。

    def generateParenthesis1(self, n):
        if n == 0:
            return []
        res = []
        self.helpler(n, n, '', res)
        return res
```

30、回溯递归：解码方法

有一个消息包含 A-Z,并通过以下规则编码，A：1，B：2，....，Z：26.现在给定一个加密后的消息，求出所有编码方式的数量。

```python
def numDecodings(self, s):
    ans, result = [], []
    self.dfs(s, ans, result)
    return result

def dfs(self, s, ans, result):
    if s == "":
        result.append(ans[:])
        return
    for i in range(1, 3):  # 回溯长度为1和长度为2的字符，因为最大为26是两位。
        if i > len(s):
            continue
        tempStr = s[: i]
        if int(tempStr) > 26 or int(tempStr) <= 0:
            return
        ans.append(tempStr)
        self.dfs(s[i:], ans, result)
        ans.pop()
```

```
输入: 12345678
输出: [['1', '2', '3', '4', '5', '6', '7', '8'], ['1', '23', '4', '5', '6', '7', '8'], ['12', '3', '4', '5', '6', '7', '8']]
输入: 23
输出: [['2', '3'], ['23']]
```

31、回溯递归：子集 2

给定一个含不同整数的集合，返回其所有的子集。

我的思路：回溯的点是整数的位置，从不同的位置取值，求出所有不同长度的子集。

但是最后是使用 combinations 函数进行取值。

```python
from itertools import combinations

class Solution:
    def subsets2(self, nums):
        result = []
        for i in range(len(nums) + 1):
            temp = list(combinations(nums, i))
            for x in temp:
                result.append(list(x))  # 将combinations的元组转化为列表。
        print(result)

#主函数
if __name__ == '__main__':
    nums = [1, 2, 3]
    print("整数集合是: ", nums)
    solution = Solution()
    print("包含的所有子集有: ", solution.subsets2(nums))
```

```
"D:\Program Files\Python37\python.exe" F:/PycharmProject
整数集合是:  [1, 2, 3]
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
包含的所有子集有:  None
```

32、回溯递归：数字组合 1

给定两个整数 n 和 1，返回【1，n】中 k 个数字所有可能组合。

```
def dfs1(self, n, k, m, p, tmp):  # p用来计数k个数字，m用来实现递推
    if k == p:
        self.res.append(tmp[:])
        return
    for i in range(m, n + 1):
        tmp.append(i)
        self.dfs(n, k, i + 1, p + 1, tmp)
        tmp.pop()
```

m是情髓

但毫无疑问 combinations（arr，k）明显更快

33、回溯递归：数字组合 2

给出一组候选数字 C 和目标数字 T，本例将找出 C 中所有的组合，使组合中的数字和为 T，C 中每个数字在每个组合中只能使用一次。

```
class Solution:
    def combinationSum2(self, candidates, target):
        candidates.sort()
        self.ans, tmp= [], []
        self.dfs(candidates, target, 0, tmp)
        return self.ans
    def dfs(self, can, target, p, tmp):
        if sum(tmp) == target:
            tmp = tmp[:]
            tmp = sorted(tmp)
            if tmp not in self.ans:  # 因为有重复数字，所以可能会存在重复情况。
                self.ans.append(tmp)
            return
        if p == len(can):  # 取到了末尾数必须返回
            return
        for i in range(p, len(can)):  # 从p开始保证只取一次
            tmp.append(can[i])
            self.dfs(can, target, p + 1, tmp)
            tmp.pop()
        return
#主函数
if __name__ == '__main__':
    candidates = [10, 1, 6, 7, 2, 1, 5]
    target = 8
    print("候选数字: ", candidates)
    print("目标数字: ", target)
    solution = Solution()
    print("结果是: ", solution.combinationSum2(candidates, target))
```

```
候选数字:  [10, 1, 6, 7, 2, 1, 5]
目标数字:  8
结果是:  [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]
```

34、回溯递归：数字组合 3

相对上题，改一个条件，就是 C 中的数字可以无限制的取用。

```
class Solution:
    def combinationSum2(self, candidates, target):
        candidates.sort()
        self.ans, tmp= [], []
        self.dfs(candidates, target, tmp)
        return self.ans
    def dfs(self, can, target, tmp):
        if sum(tmp) == target:
            tmp = tmp[:]
            tmp = sorted(tmp)
            if tmp not in self.ans:
                self.ans.append(tmp)
            return
        if sum(tmp) > target:
            return
        for i in range(len(can)):
            tmp.append(can[i])
            self.dfs(can, target, tmp)
            tmp.pop()
```

代替了上面的取到了末尾

```
候选数字:  [2, 3, 6, 7]
目标数字:  7
结果是:  [[2, 2, 3], [7]]
```

35、回溯递归：单词拆分 2

给定字符串 s 和单词字典 dict，在字符串中增加空格来构建一个句子，并且所有单词来自字典，返回所有可能的句子。

我的思路：毫无疑问要进行回溯，回溯的就是在 s 中并且在 dict 内的单词。那么出口呢就是 s 中的字符被完全截取了，就存取该情况，那么什么情况下返回呢

```
class Solution:
    def wordBreak(self, s, wordDict):
        List = []
        self.dfs(s, wordDict, "", List)
        return List

    def dfs(self, s, wordDict, string, List):
        if len(s) == 0:
            List.append(string)
        for i in range(1, len(s) + 1):# 要取到空的话，i必须要等于len
            if s[:i] not in wordDict:
                continue
            else:
                self.dfs(s[i:], wordDict, string + " " + s[:i], List)
```

36、回溯递归：单词接龙 1

给出两个单词（start 和 end）和一个字典，找出所有从 start 到 end 的最短转换序列。转换规则为：

1、每次只能改变一个字母；

2、转换过程中的中间单词必须在字典中出现。

返回所有可能的转换情况。

我的思路：回溯数组中能够被一个单词转化的所有情况，并删除该情况，直到转化了的字符串和 end 只相差一个字母，记录该情况，插回删除的元素，继续回溯，知道所有情况被找出。

```
class Solution:
    def findLadders(self, start, end, dict, ans, results):
        if self.canChange(start, end):
            if not results or len(ans) + 1 <= len(results[-1]):# 求最短长度路径
                results.append(ans[:] + [end])
            return
        for i in range(len(dict)):
            if not self.canChange(start, dict[i]):
                continue
            else:
                x = dict[i]
                dict.remove(x)  # 删掉第i个元素
                ans.append(x)
                self.findLadders(x, end, dict, ans, results)
                ans.pop()
                dict.insert(i, x)  # 将第i个元素插回原位置

    def canChange(self, string1, string2):# 相差一个字母返回True
        if len(string1) != len(string2):
            return False
        count = 0
        for i in range(len(string1)):
            if string1[i] != string2[i]:
                count += 1
        if count == 1:
            return True
        else:
            return False
```

```
# 主函数
if __name__ == '__main__':
    start = "hit"
    end = "cog"
    dict = ["hot", "dot", "dog", "lot", "log"]
    print("start是: ", start)
    print("end是: ", end)
    print("dict是: ", dict)
    solution = Solution()
    ans = [start]
    results = []
    solution.findLadders(start, end, dict, ans, results)
    print(results)
```

```
start是:  hit
end是:  cog
dict是:  ['hot', 'dot', 'dog', 'lot', 'log']
[['hit', 'hot', 'dot', 'dog', 'cog'], ['hit', 'hot', 'lot', 'log', 'cog']]
```

37、回溯递归：单词矩阵

给出一系列不重复的单词，本例将找出所有可能构成的单词矩阵。一个有效的单词矩阵是指，如果从第 k 行读出来的单词和第 k 列读出来的单词相同（0 <=k<max(numRows,numColumns）,那么就是一个单词矩阵。

```
单词序列是:  ['area', 'lead', 'wall', 'lady', 'ball', 'ssss']
['wall', 'area', 'lead', 'lady']
['ball', 'area', 'lead', 'lady']
```

```python
class Solution:
    def wordSquares(self, words):
        size = len(words)
        ans = []
        for i in range(size):
            results = []
            tempWords = words.copy()
            self.quanpailie(0, tempWords, results)
            for x in results:
                if self.is_dancijuzhen(x) and x not in ans:
                    ans.append(x)
        return ans

    def quanpailie(self, start, words, result):  # 对word[start]进行全排列
        size = len(words)
        if start >= size:
            # 只需要和字符串长度相同的部分
            result.append(words[:len(words[0])])  # 当start == size时，word[start]不需要全排列，直接列出。
        for i in range(start, size):
            words[start], words[i] = words[i], words[start]
            self.quanpailie(start + 1, words, result)
            words[start], words[i] = words[i], words[start]

    def is_dancijuzhen(self, words):
        size = len(words)
        for i in range(size):
            for j in range(size):
                if words[i][j] != words[j][i]:  # 判断是否是单词矩阵
                    return False
        return True
```

利用 Python 中的全排列函数进行计算。

```python
from itertools import combinations, permutations


class Solution:
    def wordSquares(self, words):
        ans = []
        results = list(permutations(words, 4))  # 从words中取出4个数进行全排列
        for x in results:
            if self.is_dancijuzhen(x) and x not in ans:
                ans.append(x)
        return ans

    def is_dancijuzhen(self, words):
        size = len(words)
        for i in range(size):
            for j in range(size):
                if words[i][j] != words[j][i]:  # 判断是否是单词矩阵
                    return False
        return True
```

38、回溯递归：包含所有连接的子串

给定一个字符串 s 和一个单词列表 words，在 s 中查找子串，正好是所有单词列表的连接，输出查找到子串的其实索引。

```
from itertools import permutations

class Solution(object):
    def findSubstring(self, s, words):
        ans = []
        List = list(permutations(words))# 全排列的每一种情况都是用元组存储的，且记得用list强制转换
        for i in range(len(List)):
            string = ''
            for j in range(len(List[i])):
                string += List[i][j]
            ans.append(s.index(string))
        return ans
```

39、回溯递归：翻转游戏

给定一个只包含两种字符+和-的字符串，两个人轮流翻转“++”变成“--”。当一个人无法采取行动时游戏结束，另一个人将是赢家，本例将判断能否保证先手胜利。

我的思路：先遍历的是不同的‘++’的起始位置，然后换成‘--’后，再当做一个新字符串翻转“++”，所以回溯的是不同‘++’的位置，定义一个 count，翻转一次，自加一次，当没有“++”后，如果 count 是奇数，说明能保证先手胜利。

```
class Solution:
    def canWin(self, s, count):
        flag = False
        if not s.__contains__("++"):
            if count % 2 == 1:
                return True
            else:
                return False
        for i in range(len(s) - 1):
            tempS = s
            if s[i: i + 2] == "++":
                s = s[0: i] + "--" + s[i + 2:] # 替换字符串操作
                flag = self.canWin(s, count + 1)
                if flag == True: # 回溯 return 布尔值关键操作
                    return True
                s = tempS
        return flag

#主函数
if __name__ == '__main__':
    s = "++++++"
    print("s是: ", s)
    solution = Solution()
    flags = []
    print(solution.canWin(s, 0))
```

```
"D:\Program F
s是:  +++++
False
s2是:  ++++
True
```

40、回溯递归：电话号码的字母组合

传统的电话拨号盘如图 1 所示，2-9 数字键盘上的每个数字可以代表 3 个字母之一。例如数字 2 代表 abc。输入任何 2-9 的数字组合，本例将返回所有的字母组合。

我的思路：肯定要用到回溯的方法，采用字符串截取的方法，回溯每一个字符的所有可能取值情况，出口是，当字符串被截取到空时。

```
def letterCombinations(self, digits, ans, results):
    jianpan = ["", " ", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]
    if len(digits) == 0:
        results.append(list(ans))
        return
    num = int(digits[0])# 获得第一个
    size = len(jianpan[num])
    for i in range(size):
        ans.append(jianpan[num][i])
        self.letterCombinations(digits[1:], ans, results)# 抛弃一个
        ans.remove(jianpan[num][i])
```

41、回溯递归：整数排序

```
class Solution():
    def quickSort(self, start, end, List):
        left, right = start, end # 需要重新定义首尾指针，以便下一层遍历
        if left >= right:   # 当首尾指针重叠，表明只剩下一个元素，程序结束
            return
        pivot = List[start]
        while left < right:
            while left < right and List[right] > pivot:# 先进行右边指针的移动，不然left指针会移动到最右边
                right -=1
            while left < right and List[left] <= pivot:# 那么中间元素就会变成最后一个元素
                left += 1
            if left < right:                           # 也就是说中间元素允许是第一个元素，不允许是最后一个元素
                List[left], List[right] = List[right], List[left]
        List[left], List[start] = List[start], List[left]  # 首元素为哨兵，需要交换和中间元素的位置
        self.quickSort(start, right - 1, List)# left - 1, right + 1正好切分了数组
        self.quickSort(left + 1, end, List)
```

```python
class Solutioin:
    def quickSort(self, A, start, end):  # 快速排序需要一个开始位置和一个结束位置
        if start >= end:
            return
        pivot = A[start]
        left = start
        right = end
        while left < right:
            while A[right] > pivot and left < right:  # right遇到相等的还是要交换到左边，所以不能跳过
                right -= 1
            while A[left] <= pivot and left < right:  # left遇到相等的可以跳过，因为从自身开始
                left += 1
            if left < right:
                A[left], A[right] = A[right], A[left]
        A[start], A[left] = A[left], A[start]
        self.quickSort(A, start, left - 1)  # left == right跳出循环，中间元素不需要再比较
        self.quickSort(A, right + 1, end)  # 如果这里不进行加1减1操作，将超出递归次数
```

42、回溯递归：整数替换

给定一个正整数 n，如果 n 为偶数，将 n 替换为 n/2，如果 n 为奇数， 将 n 替换为 n + 1 或 n - 1， 那么将 n 转化为 1，最少的替换次数是多少？

解题思路：

这道题拿到手上肯定是一道递归的题目，函数的意义是求 n 转化为 1，最少的替换次数是多少，那么递归就是 n / 2, n + 1, n - 1 转化为 1，最少的替换次数是多少，而当为奇数时，有两种情况，毫无疑问要用到回溯的思想，那么出口就是，当 n = 1 时，把路径存起来， 那么沿途就需要存路径点，为了节省空间复杂度，出口时做出判断，此处用栈。

```python
class Solution:
    def integeReplacement(self, n):
        ans = []
        temp = [n]
        self.dfs(n, temp, ans)
        return ans

    def dfs(self, n, temp, ans):  # temp存储沿途路径点，ans存储所有可能情况
        if n == 1:
            if len(ans) == 0:
                ans.append(temp[:])  # 只赋予值，如果没有后面的[:]，将指向同一个地址
            else:
                tempList = ans.pop()
                ans.append(temp[:] if len(temp) < len(tempList) else tempList)
            return
        if n % 2 == 0:
            temp.append(n // 2)
            self.dfs(n // 2, temp, ans)  # 一种情况，不存在回溯
        else:
            for x in [-1, 1]:
                temp.append(n + x)
                self.dfs(n + x, temp, ans)
                temp.pop()  # 两种情况， 需要回溯
```

判断长度

43、回溯递归：寻找丢失的数

给一个由 1 - n 的整数随机组成的一个字符串序列， 其中丢失了一个整数，本例将找到它。

比如：n = 20, str = "19201234567891011121314151618" 丢失的数是 17

解题思路：该题拿到手的时候明显该使用回溯的方法，因为你无法判断你从左往右找的数字是一位还是两位，所以要根据一位两位进行回溯， 那么出口的话，一是找到了 0，二是找到了重复的数字，三是到底了。那么该函数的意思就是从第几位开始往后填充存在的值。

```python
class Solution:
    def findMissing2(self, n, str):
        used = [False for _ in range(n + 1)]
        return self.find(n, str, 0, used)

    def find(self, n, str, index, used):  # 从index处开始找丢失的整数
        result = []
        if index >= len(str):  # 到了尾部，注意这里不是n
            for i in range(1, len(used)):
                if used[i] == False:
                    result.append(i)
            return result if len(result) == 1 else -1
        for i in range(1, 3):
            temp = int(str[index: index + i])
            if temp == 0:
                return # 如果有0出现， 那么直接返回就好
            elif temp > n:
                continue
            elif used[temp] == True:
                continue
            else:
                used[temp] = True
                result= self.find(n, str, index + i, used)
                used[temp] = False
        return result
```

回溯不满足条件的也可以用continue

44、分割回文串 1

给定一个字符串 s，将 s 分割成一些子字符串，使每个子字符串都是回文，返回 s 符合要求的最少分割次数。

我的思维：肯定是能够把字符串分割成子字符串都是回文的，大不了都分割成单个的就好，就是要尽可能找到长的回文串。

其实可以用回溯的方法进行解决，找到回文长度分别为（n ~~1）的子串，知道回文长度为 1 是一种解法。

```python
class Solution:
    def minCut(self, s):...

    def minCut2(self, s):
        ans, temp = [], []
        self.dfs(s, temp, ans)
        return ans

    def isHuiWen(self, s):
        if s == s[::-1]:
            return True
        else:
            return False

    def dfs(self, s, temp, ans):
        size = len(s)
        if size == 1:  # 出口是当只剩一个元素时即是回文
            temp.append(s)
            ans.append(temp[:])
            temp.pop()  # 注意，因为最后一个元素是在这里添加的，所以也要进行回溯。
            return
        for i in range(1, size):
            if self.isHuiWen(s[:i]):
                temp.append(s[:i])
                self.dfs(s[i:], temp, ans)
                temp.pop()
if __name__ == '__main__':
    s = "aab"
    print("初始字符串: ", s)
    solution = Solution()
    print("分割次数: ", solution.minCut2(s))
```

45、回溯递归：分割回文串 2

给定一个字符串 s，将 s 分割成一些子字符串，使每个子字符串都是回文串，返回 s 所有可能的回文串分割方案：

我的思维：肯定是能够把字符串分割成子字符串都是回文的，大不了都分割成单个的就好，就是要尽可能找到长的回文串。

其实可以用回溯的方法进行解决，找到回文长度分别为（n ~~1）的子串，知道回文长度为 1 是一种解法。

```python
class Solution:
    def minCut(self, s):...

    def minCut2(self, s):
        ans, temp = [], []
        self.dfs(s, temp, ans)
        return ans

    def isHuiWen(self, s):
        if s == s[::-1]:
            return True
        else:
            return False

    def dfs(self, s, temp, ans):
        size = len(s)
        if size == 1:  # 出口是当只剩一个元素时即是回文
            temp.append(s)
            ans.append(temp[:])
            temp.pop()  # 注意，因为最后一个元素是在这里添加的，所以也要进行回溯。
            return
        for i in range(1, size):
            if self.isHuiWen(s[:i]):
                temp.append(s[:i])
                self.dfs(s[i:], temp, ans)
                temp.pop()
if __name__ == '__main__':
    s = "aab"
    print("初始字符串：", s)
    solution = Solution()
    print("分割次数：", solution.minCut2(s))
```

46、回溯递归：回文排列 1

给定一个字符串，判断字符串是否存在回文排列，若存在则返回 True，否则返回 False

解题思路：其实是问用该字符串中的字符能够组成回文串。

用一个字典，键用来存储字符串中存在的字符，值用来计数字符的个数

然后取出该字典的 values 数组，如果存在两个奇数那么就不可能组成回文串，**只能允许有一个奇数**，放中间组成回文串。

```python
class Solution:
    def canPermutePalindrome(self, s):
        lookup = {}
        count = 0
        for x in s:  # 遍历字符串
            if x not in lookup:
                lookup[x] = 1  # 初始化
            else:
                lookup[x] += 1
        for x in lookup.values():
            count += x % 2  # 计数奇数的个数
        return count <= 1
```

47、回溯递归：回文排列 2

给定一个字符串 s,返回所有的回文排列（不重复）；如果没有回文排列，返回空列表

解题思路：用全排列方便一点

```
class Solution:
    def quanpailie(self, s, start, results):
        size = len(s)
        s = list(s)  # 字符串转字符串数组
        if start >= size:
            string = "".join(s)  # 字符串数组转字符串
            if s == s[::-1] and string not in results:# 字符串转化为列表用join，用str只能转换数字
                results.append(string)
        for i in range(start, size):
            s[start], s[i] = s[i], s[start]
            self.quanpailie(s, start + 1, results)
            s[start], s[i] = s[i], s[start]
```

数据结构

48、k 组翻转链表

给定链表及整数 k，将这个链表从头指针开始，每 k 个元素翻转一下。链表元素个数不是 k 的倍数，最后剩余的元素不用翻转。

我的思路：首先需要两个函数：

1、翻转指定范围的链表：reverse（start，end）；其中 start 和 end 分别是起始节点和结束节点，返回的是翻转后链表的头节点和下一个翻转链表的开头。

2、找到整个链表的各个范围的头结点和尾节点

其中 reverse 链表利用的是 start 指针一直往后移，而 start 经过的节点都放到头节点，等 start 到了 end 后，自然而然就完成了翻转了。

```
class Solution:
    def reverse(self, start, end):
        newHead = ListNode(0)# 做链表题时首先定义一个头节点，指向链表
        newHead.next = start
        while newHead.next != end:
            temp = start.next
            start.next = temp.next
            temp.next = newHead.next
            newHead.next = temp
        return [newHead.next, start]# 返回翻转后的链表的头结点和下一个范围的起始点的前驱

    def reverseKGroup(self, head, k):
        newHead = ListNode(0)
        newHead.next = head
        start = newHead  # 因为要接着下一个翻转链表
        while start.next:
            end = start # end是基于start开始往后算的，除了第一个start是头结点，后续的头结点都是程序过程中记录的
            for i in range(k):
                end = end.next
                if end == None:# 当不满足k个数时，结束翻转
                    return newHead.next
            print(start.next.val, end.val)
            doneHead, nextStart = self.reverse(start.next, end)# 注意，你这里应该传入的是start.next，也就是翻转链表的头
            start.next = doneHead
            start = nextStart  # 下一个翻转链表的前驱
        return newHead.next
```

所以不能将指针指向head

```
#主函数
if __name__ == '__main__':
    node1 = ListNode(1)
    node2 = ListNode(2)
    node3 = ListNode(3)
    node4 = ListNode(4)
    node5 = ListNode(5)
    node1.next = node2
    node2.next = node3
    node3.next = node4
    node4.next = node5
    k = 2
    list1 = []
    #创建对象
    solution = Solution()
    newlist = solution.reverseKGroup(node1, k)
    while (newlist):
        list1.append(newlist.val)
        newlist = newlist.next
    print("初始化的链表是：", [node1.val, node2.val, node3.val, node4.val, node5.val])
    print(" 翻转后的结果是:", list1)
```

```
初始化的链表是： [1, 2, 3, 4, 5]
 翻转后的结果是: [2, 1, 4, 3, 5]
```

49、线段树的修改

修改线段树的末尾节点，确保修改后，线段树的每个节点 max 属性仍然具有正确的值。

我的思路：查找二叉树，找到该点后，进行修改，然后对 max 重新构造树。

如何重新构造树？

采用后续遍历的方法，根节点取左右子树的更大值，return 根节点。

解题思路：不用将查找和修改分开，后续遍历途中就可以进行值的更改。

```
class Solution:
#参数root、index、value是线段树的根，并使用[index, index]将节点的值更改为新的给定值
#返回列表
    def modify(self, root, index, value):
        pos = root
        if pos.start == index and pos.end == index:  # 后续遍历途中进行值的修改，且因为修改的是叶子节点，所以要返回max
            pos.max = value
            return pos.max
        elif pos.left == None and pos.right ==None:  # 这种要返回值的，不要遍历到空节点，遍历到叶子结点就好
            return pos.max
        leftMax = self.modify(pos.left, index, value)
        rightMax = self.modify(pos.right, index, value)
        pos.max = max(leftMax, rightMax)
        return pos.max
```

50、线段树的构造 1

根据列表构造线段树，线段树的叶子节点的 start==end==列表索引，max=列表中的值

我的思路：毫无疑问应该采用后序遍历进行树的构造。

解题思路：上述思路错误，应该采取前序遍历构造，先构造父节点，然后构造子节点，直到列表中的元素不再支持构造子节点，返回构造好的叶子节点。

因为这里构造的是线段树，所有叶子结点加起来就是列表中的所有元素，所以这里构造很特殊，非叶子节点的 max 其实都是 0，等到叶子节点构造完成后，在重新更改值，列表中的值在遍历到叶子节点是赋值。

```python
class Solution:
    def build(self, A):
        return self.buildTree2(0, len(A) - 1, A)

    def buildTree2(self, start, end, A):
        if start == end:
            return SegmentTreeNode(start, end, A[start])  # A中的元素直到遍历到叶子结点才赋值
        node = SegmentTreeNode(start, end, 0)
        node.left = self.buildTree2(start, (start + end)// 2, A)  # 采取这样的方法，确定构造多少节点
        node.right = self.buildTree2((start + end) // 2 + 1, end, A)
        node.max = max(node.left.max, node.right.max)  # 重新赋值
        return node
```

如果是构造普通的二叉树

51、线段树的构造 2

本例将实现一个 build 方法，接收 start 和 end 作为参数，然后构造一个代表区间【start，end】的线段树，返回这棵线段树的根。这棵线段树不是叶节点全是列表中的点，而是将区间中的数不断二分，分到为一个数作为叶子节点位置。

我的思路：肯定还是用前序遍历，先建立父节点，再建立子节点。

相比于上一题其实就是少了一个 max。

```python
def build2(self, start, end):
    if start == end:
        return SegmentTreeNode(start, end)  # 当不可再分，建立叶子节点
    node = SegmentTreeNode(start, end)
    node.left = self.build(start, (start + end) // 2)
    node.right = self.build((start + end) // 2 + 1, end)
    return node
```

52、线段树查询 1

找到线段树中指定区间【start，end】内的元素个数

我的思路：其实元素总个数就是叶子节点的 count 值相加，那么其实只要遍历到叶子节点，看该节点的范围是否符合【start， end】，如果符合，那么相加就好了。

```python
def query2(self, root, start, end):  # 节点root下[start, end]的元素个数
    # if start > root.end or end < root.start:  其实该行代替下行更好，想想为什么
    if root.left == None and root.right == None and start > root.end or end < root.start:
        return 0  # 当叶子节点不满足范围时，返回0就好
    if root.left == None and root.right == None and start <= root.start and root.end <= end:
        return root.count
    left_count = self.query2(root.left, start, end)
    right_count = self.query2(root.right, start, end)
    return left_count + right_count
```

53、线段树查询 2

跟上题相比，找的不是区间【start，end】内的值之和，而是区间内的最大值。

我的思路：其实就是遍历到叶子节点就直接 return max，而后记录 left_max,right_max，

最后返回其中更大值，一直返回到根节点。

```
def query(self, root, start, end):  # 找该节点下存在于区间[start, end]的最大值
    if root.left == None and root.right == None:
        if end < root.start or start > root.end:
            return -65536  # 超出范围时返回一个超小值
        if start <= root.start and root.end <= end:
            return root.max
    leftMax = self.query(root.left, start, end)
    rightMax = self.query(root.right, start, end)
    return max(leftMax, rightMax)
```

54、是否为子树

有两个大小不同的二叉树，判断一个是否是另一棵的子树：

我的思路：用同一种遍历，求出二者的遍历序列，看其中是否存在一者包含另一者的情况。

解题思路：我的思路是错误的，因为不同结构的树也可能遍历序列相同。

由于先序遍历（后序遍历） 再加上中序遍历能唯一确定一棵二叉树（必须知道中序遍历，因为只有中序遍历才能知道根节点的左右各有多少个节点）

拓展：

一，问题介绍

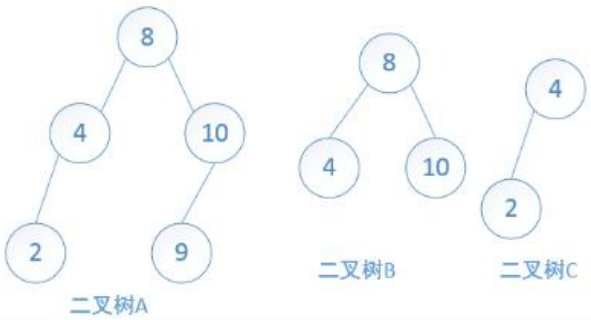
本文章讨论两个问题：

①如何判断两棵二叉树的结构是一样的、对应的每个结点都有着相同的值。--即判断两棵二叉树是一样的

②给定两棵二叉树，如何判断一棵二叉树是另一棵二叉树的子结构

③给定两棵二叉树，如何判断一棵二叉树是另一棵二叉树的**子树**

注意，子结点与子树有那么一点点不同。



上面的二叉树B 是二叉树A 的子结构，但是不能说是二叉树A的子树。但是二叉树C 是 二叉树A的子树。

二叉树的子树和子结构

子树的意思是只要包含了一个结点，就得包含这个结点下的所有节点.

子结构的意思是包含了一个结点，可以只取左子树或者右子树，或者都不取。

55、最小子树

给定一棵二叉树，找到和为最小的子树，返回其根节点：

我的思路：1、定义一个求树和的函数 2、遍历树，记录最小和及其根节点

```python
class Solution:
    def findSubtree(self, root):
        result = []
        self.pre_order(root, result)
        return result

    def pre_order(self, root, result):
        if root is None:
            return
        if len(result) == 0 or self.sumTree(root) < result[-1][1]:# 为空或者和更小时，追加到末尾
            result.append([root.val, self.sumTree(root)])
        self.pre_order(root.left, result)
        self.pre_order(root.right, result)

    def sumTree(self, root):# 求树的和
        if root is None:
            return 0
        leftSum = self.sumTree(root.left)
        rightSum = self.sumTree(root.right)
        return leftSum + rightSum + root.val# 以叶子节点为例，0 + 0 + 叶子值
```

56、具有最大平均数的子树

给定一棵二叉树，找到具有最大平均值的子树，返回子树的根节点，并输出子树
我的思路：1、定义一个函数求树的平均值 2、遍历该树，比较最大值。

```python
class Solution:
    def findSubtree2(self, root):
        result = []
        self.pre_order(root, result)
        return result[-1][0]# 返回最小平均值的树根节点

    def pre_order(self, root, result):
        if root is None:
            return
        self.temp = [0, 0]# 定义全局数组
        self.avgTree(root)
        sum = self.temp[0]
        size = self.temp[1]
        if len(result) == 0 or result[-1][1] < (sum/size):
            result.append([root, sum/size])
        self.pre_order(root.left, result)
        self.pre_order(root.right, result)

    def avgTree(self, root):# 不能传递数字， 而应传递数组，因为数组存的是地址。
        if root is None:
            return
        self.temp[0] += root.val # 利用全局变量记录和与节点的数量。
        self.temp[1] += 1
        self.avgTree(root.left)
        self.avgTree(root.right)
```

57、二叉搜索树中最接近的值

给定一棵非空二叉搜索树（BST）以及一个 target 值，找到其中最接近给定值的 k 个数。
注意，1、给出的 targer 值是浮点数；2、可以假设 k 总是合理的，即 k 小于等于总结点数；
3、可以保证在给出 BST 中只有唯一一个最接近给定值的 k 个值的集合。

我的思路：利用中序遍历序列化得到排序的数组，然后按照找到 target，从中间往两边发散找到 k 个值。

解题思路：思路误区，只是要你找出 k 个最接近 target 的数，并没有说要连续。所以只需要遍历树，记录树节点值和 target 的差和树节点，按差从小到大排序取 k 个数。

```python
class Solution:
    def closestKValues(self, root, target, k):
        self.allNodes = []
        self.pre_order(root, target)
        self.allNodes = sorted(self.allNodes, key = lambda x: x[0]) # 按差值排序
        result = []
        for i in range(k):
            result.append(self.allNodes[i][1]) # 记录k个最接近的值
        return result

    def pre_order(self, root, target):
        if root is None:
            return
        self.allNodes.append([abs(root.val - target), root.val]) # 记录差值和节点值
        self.pre_order(root.left, target)
        self.pre_order(root.right, target)
```

58、二叉搜索树中插入节点

给定一棵二叉搜索树和一个新的树节点，将节点插入树中，需要保证该树仍然是一颗二叉搜索树。

我的思路：无

解题思路：陷入思维误区，往二叉搜索树中插入节点，肯定是作为叶子节点插入的，所以不需要遍历，直接循环就好，大于节点值往右，小于节点值往左，如果向左向右为空了，那就直接插入，跳出循环，返回根节点。

```python
def insertNode2(self, root, node):
    cur = root # 从根节点开始查找
    while True: # 等到指针指向插入节点为止
        if cur.val > node.val: # 当前值大于节点值往左
            if cur.left == None:  # 如果左边是空，直接插入
                cur.left = node
                break  # 插入后，就跳出循环就好
            cur = cur.left
        else:
            if cur.right == None:
                cur.right = node
                break
            cur = cur.right
    return root
```

59、二叉搜索树中删除节点

给定一棵具有不同节点值的二叉搜索树，删除树中与给定值相同的节点。如果没有，就不做任何处理，如果有，保证处理之后树仍然是一棵二叉搜索树。

我的思路：利用循环找到要删除节点，将节点替换为该节点的右节点，然后将该节点的

左节点变成右节点的左节点。

解题思路：记录非删除节点，对非删除节点构造树

```
class Solution:
    ans = []  # 全局变量的使用

    def inorder(self, root, value):
        if root is None:
            return
        if root.val != value:
            self.ans.append(root.val) # 将非删除元素记录
        self.inorder(root.left, value)  # 调用成员变量或方法要用self
        self.inorder(root.right, value)

    def build(self, l, r):  # 利用二分法建立平衡二叉树
        if l == r:
            return TreeNode(self.ans[l])  # 建立叶子结点
        if l > r:
            return
        mid = (l + r) // 2
        node = TreeNode(self.ans[mid])  # 建立根节点和非叶子节点
        node.left = self.build(l, mid - 1)
        node.right = self.build(mid + 1, r)
        return node  # 返回一根子树

    def removeNode(self, root, value):
        self.inorder(root, value)
        self.ans = sorted(self.ans) # 建立平衡二叉树，首先得对数组排序
        return self.build(0, len(self.ans) - 1)
```

60、二叉搜索树转化成更大的树

给定二叉搜索树，将其转换为更大的树，使原始 BST 上每个节点的值都更改为在原始数中大于等于该节点值之和（包括该节点）

我的思路：没理解题目意思

解题思路：就是每个节点值应该等于大于该节点值的其他节点值之和加上该节点的值

变相使用中序遍历，中序遍历是左中右，改为右中左的顺序。进行遍历，传递一个 sum 值，将经过的 sum 值加到经过的节点的 val 中。

因为比根节点大的就是其右子树，比左节点大的是根节点和兄弟右子树。

```
class Solution:
    def convertBST(self, root):
        self.sum = 0
        self.helper(root)
        return root
    def helper(self, root):
        if root is None:
            return
        self.helper(root.right) # 先访问右边
        self.sum += root.val
        root.val = self.sum
        self.helper(root.left)
```

61、二叉搜索树的搜索区间

给定两个值 k1、k2（k1 < k2）和一个二叉搜索树的根节点。找到树中所有值在 k1 到 k2 范围内的节点，并打印所有 x，其中 x 是二叉搜索树的节点值，返回所有升序的节点值。

我的思路：中序遍历所有节点，记录在范围内的值

```
def searchRange2(self, root, k1, k2): # 返回一颗子树满足条件的数
    res = []
    if root is None:
        return []    # 完美利用局部列表完成任务
    res.extend(self.searchRange2(root.left, k1, k2))
    if k1 <= root.val <= k2:
        res.append(root.val)
    res.extend(self.searchRange2(root.right, k1, k2))
    return res
```

62、二叉搜索树的中序后继

给定一棵二叉搜索树以及一个节点，采用中序遍历的方法，求给定后继节点，如果没有后继节点则返回 null。

我的思路：利用中序遍历将二叉搜索树序列化。

```
def inorderSuccessor2(self, root, p):
    stack = []
    pos = root
    flag = False
    while pos is not None or stack:
        if pos is not None:
            stack.append(pos)
            pos = pos.left
        else:
            pos = stack.pop()  # 因为这是中序遍历的打印口，该方法适合求前序和中序的后继
            if flag:
                return pos
            if pos.val == p.val:
                flag = True
            pos = pos.right
```

从中序遍历的特性去寻找：左-根-右。中序遍历一个结点时，下一个结点有三种情况：

1. 如果当前结点有右结点，则下一个遍历的是右子树的最左结点；
2. 如果当前结点无右结点，若它是父节点的左儿子，则下一遍历的是父节点；
3. 如果当前结点无右结点，且它是父节点的右儿子，则所在子树遍历完了。向上寻找一个作为左儿子的祖先结点，那么下一遍历的就是该祖先结点的父节点；（一直找到根节点为止）
4. 如果上面三种情况都没找到，则该节点是树的最后一个结点，无后继结点

```python
def inorderSuccessor(self, root, p):
    successor = None
    while root and root.val != p.val:
        if p.val < root.val:  # // 只有往左查找时 successor 才需要不断更新
            successor = root
            root = root.left
        else:
            root = root.right
    return successor
```

63、二叉搜索树两数之和

给定一个整数 n，在二叉搜索树中找到和为 n 的两个数字返回
我的思路：这种情况是不是有很多种？岂不是需要回溯
该题代码有问题，只能返回一种情况

```python
class Solution:
    def twoSum(self, root, n):
        if not root:
            return
        stack, check = [], set()
        while stack or root:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            if root.val == n:
                return root.val
            elif n - root.val in check:  # 在check中表示有这样的数
                return [root.val, n - root.val]
            if root.val not in check:  # 存入check
                check.add(root.val)
            root = root.right
        return False
```

64、裁剪二叉搜索树

给定一棵有根的二叉搜索树和两个数字 min、max，将这个数超出范围的数裁掉，依然保证该数是一颗合法的二叉搜索树。

我的思路：肯定利用构建二叉树的思想，进行前序遍历，然后看是否符合范围，不符合范围，看大于还是小于，如果大于，那么舍弃掉根节点和右子树，反之亦然。直到为 None

```python
def trimBST2(self, root, mininum, maximum):  # 对一颗一颗子树进行裁剪， 然后返回。
    if root is None:
        return
    if root.val > maximum:  # 当根节点值比最大值大，返回裁剪好的左子树。
        return self.trimBST2(root.left, mininum, maximum)
    if root.val < mininum:
        return self.trimBST2(root.right, mininum, maximum)
    root.left = self.trimBST2(root.left, mininum, maximum)
    root.right = self.trimBST2(root.right, mininum, maximum)
    return root  # 当在区间内时， 返回一颗完全的数。
```

当不符合范围时，根节点和其中一棵子树已经没了，所以不需要返回root，直接返回其中一棵还在的子树即可。

65、统计完全二叉树节点树

给定一棵完全二叉树，计算它的节点数

我的思路：为空时返回 0，之后返回左子树的节点数+右子树的节点数+1

```python
def countNodes2(self, root):
    if root == None:
        return 0
    left = self.countNodes(root.left)
    right = self.countNodes(root.right)
    return left + right + 1
```

66、二叉搜索树迭代器

本例将实现一个带有下列属性的二叉查找树迭代器：

1、next（）返回二叉树中下一个最小的元素；

2、元素按照递增的顺序被访问（例如中序遍历）

3、Next（）和 hasNext（）的询问操作要求平均时间复杂度为 O（1）

我的思路：如何利用中序遍历实现时间复杂度为 O（1）？

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left, self.right = None, None
class BSTIterator:
#参数root是二叉树的根节点
    def __init__(self, root):
        self.stack = []
        self.pos = root

    def hasNext1(self):
        return self.pos is not None or len(self.stack) > 0  # 只有栈为空和指针指向空节点才会返回False

    def next1(self):  # 非递归中序遍历的分解应用，调用一次该函数表示将求出一个中序后继
        while self.pos is not None:
            self.stack.append(self.pos)
            self.pos = self.pos.left
        self.pos = self.stack.pop()
        nxt = self.pos
        self.pos = self.pos.right
        return nxt
```

67、翻转二叉树

翻转一棵二叉树，即所有非叶子节点的左右子树交换。

我的思路：从根节点开始，重新构造树，左右节点互换。

```
def dfs1(self, node):
    if node == None:
        return
    left = self.dfs1(node.left)  # 必须赋临时值，不然会左子树会覆盖右子树
    right = self.dfs1(node.right)
    node.left = right
    node.right = left
    return node
```

68、相同二叉树

检查两颗二叉树是否等价，即两颗二叉树必须拥有相同的结构，并且每个对应位置节点上的数都相等。

我的思路：因为前序遍历（后序遍历）和中序遍历能唯一确定一棵树。

解题思路：两棵树同时从根节点开始遍历，值不相等就返回 False，或者其中一个为 None 另一个不为 None，当然同时为 None 也可以返回 True，接下来再比较两个子树是否相等，最后返回两颗子树的布尔值相与，都为真是为真。

```
def isIdentical2(self, a, b):
    if a == None and b == None:
        return True
    if a == None or b == None:  # 防止a、b中出现没有val值的情况，因为前面已经排除二者皆为None的情况，
                                # 所以只要二者其中有None，则不相等
        return False
    if a.val != b.val:
        return False
    flaga = self.isIdentical2(a.left, b.left)
    flagb = self.isIdentical2(a.right, b.right)
    return flaga and flagb  # 只要两颗子树为真，那么这棵树就为真。
```

69、前序遍历树和中序遍历树构造二叉树

根据前序遍历和中序遍历的结果构造二叉树

我的思路：其实就是要把前序遍历和中序遍历的相同的部分分在一起，知道只剩一个元素时，那么前序遍历和中序遍历是相等的。那么这就是出口。

解题思路：构造二叉树肯定是前序遍历的，先构造根节点，在构造两个子节点，最后返回根节点。

```python
def buildTree1(self, preorder, inorder):# 根据前序和后序序列构造二叉树
    if not preorder:  # 出口在让中序遍历和后续遍历相等，也就是只剩下一个元素时，返回。
        return
    root_val = preorder[0]  # 按照中序遍历数组找第一个元素也是可以的。
    root_inorder_pos = inorder.index(root_val)# 找到中序遍历根节点的位置
    root = TreeNode(root_val)# 构造根节点
    root.left = self.buildTree1(preorder[1: root_inorder_pos + 1], inorder[:root_inorder_pos])
    root.right = self.buildTree1(preorder[root_inorder_pos + 1:], inorder[root_inorder_pos + 1:])
    return root# 左右子树根据前序和中序建立了二叉树， 返回这棵子树
```

70、二叉树的后续遍历

将二叉树后序遍历序列化

```python
class Solution:
    def postorderTraversal2(self, root):
        stack1, stack = [], []
        stack1.append(root)
        while stack1:
            pos = stack1.pop()  # 就这个地方与层次遍历不同
            stack.append(pos.val)
            if pos.left:
                stack1.append(pos.left)
            if pos.right:
                stack1.append(pos.right)
        return stack[::-1]# 返回倒序
```

71、二叉树的所有路径

给出一棵二叉树，找出从根节点到叶子节点的所有路径

我的思路：传递一个字符串，进行遍历，到了叶子节点，记录该字符串，也就是记录一条路径。

```python
class Solution:
    def binaryTreePaths(self, root):
        if root is None:
            return []
        result = set([])
        self.dfs2(root, [], result)
        return result

    def dfs2(self, node, path, result):
        path.append(str(node.val))  # 如果你只判断到叶子结点，那么应该放在这
        if node.left is None and node.right is None:
            result.append('->'.join(path))
            path.pop()
            return
        self.dfs(node.left, path, result)
        self.dfs(node.right, path, result)
        path.pop()

    def dfs(self, node, path, result): # 找到从node到叶子节点的所有路径
        if node == None: # 不能用这个，不然左右子点都为空的话，那么路径会出现两遍，但这里用的集合，所以可以
            result.add('->'.join(path))
            return
        path.append(str(node.val))
        self.dfs(node.left, path, result)
        self.dfs(node.right, path, result)
        path.pop()
```

72、中序遍历树和后续遍历树构造二叉树

根据中序遍历树和后序遍历序列构造二叉树

```python
def buildTree2(self, inorder, postorder):
    if not inorder: # 当其中为空时，也就是当只剩一个元素时
        return
    root_val = postorder[-1]
    root_inorder_pos = inorder.index(root_val)
    root = TreeNode(root_val)
    root.left = self.buildTree2(inorder[:root_inorder_pos], postorder[:root_inorder_pos])
    root.right = self.buildTree2(inorder[root_inorder_pos + 1:], postorder[root_inorder_pos: len(postorder) - 1])
    return root
```

73、二叉树的序列化和反序列化

将数序列化成字符串，遇到空则返回‘#’，再根据该字符串反序列化为二叉树

我的思路：如果空节点用‘#’表示出来了，那么是可以根据序列化为字符串的遍历书序进行反序列化的。例如你用前序遍历序列化，那就可以用前序遍历反序列化。

```python
class Solution:
    def serialize(self, root):
        if not root:
            return ['#'] # 遇到叶子结点，初始化列表
        ans = []
        ans.append(str(root.val))
        ans += self.serialize(root.left) # 列表相连可以用加号
        ans += self.serialize(root.right)
        return ans

    def deserialize(self, data):
        ch = data.pop(0)  # 根据列表建立树，该种情况只在该情况下可以构建树，不然的话需从中间开始，或者采用2 * n + 1/2 * n + 2。
        if ch == '#':
            return None
        else:
            root = TreeNode(int(ch))
        root.left = self.deserialize(data)
        root.right = self.deserialize(data)
        return root
```

74、二叉树的层次遍历 1

真正的层次遍历

```python
class Solution:
    def levelOrder(self, root):
        if root is None:
            return []
        queue = deque([root])
        result = []
        while queue:
            level = [] # 记录一层的节点
            for _ in range(len(queue)): # 该层的个数由queue中的个数决定
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)
        return result
```

75、二叉树的层次遍历 2

从下往上进行层次遍历
我的思路：应用上面的真正层次遍历思想

```python
class Solution:
    def levelOrderBottom(self, root):
        if root is None:
            return []
        queue = deque([root])
        result = []
        while queue:
            level = []
            size = len(queue)
            for _ in range(size): # 这个是精髓
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)
        return list(reversed(result))
```

76、二叉树的锯齿形层次遍历

给出一棵二叉树，返回其节点值的锯齿形层次遍历，即先从左往右，下一层再从右往左，层与层之间交替进行。

```python
class Solution:
    def zigzagLevelOrder2(self, root):
        import collections
        queue, result = collections.deque(), []
        queue.append(root)
        count = 0
        while queue:
            level = []
            for _ in range(len(queue)):
                pos = queue.popleft()
                level.append(pos.val)
                if pos.left:
                    queue.append(pos.left)
                if pos.right:
                    queue.append(pos.right)
            if count % 2 == 0:  # 顺着遍历
                result.append(level)
            else:               # 反着遍历
                result.append(level[::-1])
            count += 1
        return result
```

77、动态规划：攀爬字符串

给定一个字符串 S1，将其递归地分割成两个非空子字符串,从而将其表示为二叉树。

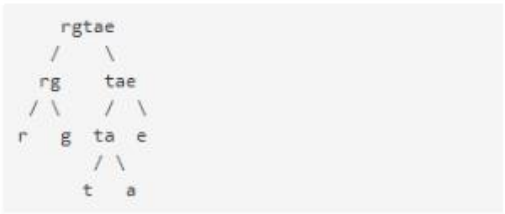
下面是s1 = “great”的一个可能表达：

在攀爬字符串的过程中，我们可以选择其中任意一个非叶节点，然后交换该节点的两个儿子。

例如，我们选择了 “gr” 节点，并将该节点的两个儿子进行交换，从而产生了攀爬字符串 “rgeat”。

我们认为，“rgeat” 是 “great” 的一个攀爬字符串.

类似地，如果我们继续将其节点 “eat” 和 “at” 进行交换，就会产生新的攀爬字符串 “rgtae”。



同样地，” rgtae” 也是 “great” 的一个攀爬字符串。

给定两个相同长度的字符串s1 和 s2，判定 s2 是否为 s1 的攀爬字符串。

我的思路：求出 s1 所有可能的攀爬字符串，看 s2 是否是。但这样太复杂。

解题思路：其实判断二者是否互为攀爬字串：

1、二者是否长度相等，二者排序之后是否字母相同，否则，return False

2、对截断位置进行遍历，假如将 s1 截断为 s11,s12，将 s2 截断为 s21,s22，如果 s11 == s21 and s12 == s22 or s11 == s22 and s12 == s21，那么返回 True

3、那么出口是什么呢？出口应该是当 s1 = s2 时，返回 return，也就是如果是互为攀爬字符串时，那么一定满足 s11 == s21 and s12 == s22 or s11 == s22 and s12 == s21。

```python
def isScramble(self, s1, s2):# 判断s1和s2是否互为攀爬字符串
    tempS1 = sorted(s1)
    tempS2 = sorted(s2)
    if s1 == s2:
        return True
    if len(s1) != len(s2):
        return False
    if tempS1 != tempS2:
        return False
    length = len(s1)
    for i in range(1, len(s1)):#判断二者的子串是否互为攀爬字符串，可以以最后一个元素为折分线，后一部分就是最后一个元素
        if self.isScramble(s1[:i], s2[:i]) and self.isScramble(s1[i:], s2[i:]):
            return True
        if self.isScramble(s1[:i], s2[i:]) and self.isScramble(s1[i:], s2[:i]):
            return True
    return False
```

集合、数组、字符串

78、集合、列表、字符串：最接近的三数之和

给出一个包含 n 个整数的数组 s，找到与给定整数 target 最接近的三元组，返回这三元组的和。

我的思路：将数组中的所有数都减去 target，然后取绝对值，存入列表，一个绝对值，对应着一个下标，排序列表，求出最接近的三个数的下标，然后进行求和。

```python
class Solution:
    def threeSumClosest(self, numbers, target):
        hash = []# 这里不能用集合，因为集合不支持sorted。
        for i in range(len(numbers)):
            temp = abs(target - numbers[i])
            hash.append([temp, i])
        hash = sorted(hash, key=lambda x: x[0])
        SUM = 0
        for i in range(3):  # 如果求最接近的n个数之和，那么这里换成n就好了
            SUM += numbers[hash[i][1]]
        return SUM
```

79、集合、列表、字符串：三数之和为 0

给出一个有 n 个整数的数组 S，在 S 中找到三个整数 a、b、c，找到所有 a + b + c = 0 的元组。

我的思路：先遍历第一个整数，然后将剩余的两个数之和等于 target - 第一个整数来做。

```python
class Solution:
    def threeSum(self, nums):
        nums.sort()# 需排序
        results = []
        length = len(nums)
        for i in range(length - 2):  # 后面留两个数
            target = -nums[i]        # 因为三数之和等于0，相当于0 - nums[i]
            if i and nums[i] == nums[i - 1]:  # 负数会索引到后面，所以不用担心
                continue
            left, right = i + 1, length - 1# 因为是排序的，所以可以这么做。
            while left < right:
                sum = nums[left] + nums[right]
                if sum < target:
                    left += 1
                elif sum > target:
                    right -= 1
                else:
                    results.append([nums[i], nums[left], nums[right]])
                    left += 1
                    right -= 1
        return results
```

可以用 combinations 来做：

```python
def threeSum2(self, nums):
    tempList = list(combinations(nums, 3))
    ans = []
    for x in tempList:
        if sum(x) == 0:
            ans.append(list(x))# 将元组转化为列表
    return ans
```

80、集合、列表、字符串：四数之和为定值

给一个包含 n 个数的整数数组，在 s 中找到所有使得和为给定整数 target 的四元组（a,b,c,d）

我的思路：直接用 combinations 就好

给出的代码过于复杂， 如下：

```python
def fourSum(self, nums, target):
    nums.sort()
    ans = []
    length = len(nums)
    for i in range(length):  # 第一个数
        if nums[i] == nums[i - 1]:  # 去重
            continue
        result1 = target - nums[i]
        for j in range(i + 1, length):# 第二个数
            if j != i + 1 and nums[j] == nums[j - 1]:# 去重
                continue
            result2 = result1 - nums[j]
            left, right = j + 1, length - 1
            while left < right:   # 两数之和为定值
                sum = nums[left] + nums[right]
                if sum < result2:
                    left += 1
                elif sum > result2:
                    right -= 1
                else:
                    ans.append([nums[i], nums[j], nums[left], nums[right]]) # 四个数
                    left += 1
                    right -= 1
    return ans
```

81、奇偶分割数组

分割一个整数数组，使得奇数在前，偶数在后。

我的思路：快速排序的部分应用。左边找偶数，右边找奇数，交换。

```python
def partitionArray2(self, nums):
    size = len(nums)
    start, end = 0, size - 1
    while start < end:
        while nums[start] % 2 == 1:
            start += 1
        while nums[end] % 2 == 0:
            end -= 1
        nums[start], nums[end] = nums[end], nums[start]
        start += 1
        end -= 1
    return nums
```

82、单词拆分 1

给定字符串 s 和单词字典 dict，本例将判断 s 是否可以分成一个或多个以空格分割的子字符串，并且这些子字符串都在字典中存在。

我的思路：截取 0-i 个字符作为新的字符串，如果字符串再字典中，那么就将后续未截取的字符串作为新的字符串进行重新的截取，知道字符串都空，返回 True

```python
class Solution:
    def wordBreak(self, s, dict):
        i = 1
        while(len(s) >= 0): # 因为s长度可变，所以应避免使用for循环
            if s[0: i] in dict:
                s = s[i:]
                i = 1  # 重新初始化
            if s == "":
                return True
            i += 1
        return False
```

```python
class Solution:
    def numDistinct2(self, S, T):  # https://www.jianshu.com/p/3d061dbe0c99
        dp = [[0 for j in range(len(T) + 1)] for i in range(len(S) + 1)]
        for i in range(len(S) + 1):
            dp[i][0] = 1# 空集是所有集合的子集，所以当T为空集时，S中包含T的个数都为1
        for i in range(len(S)):
            for j in range(len(T)):
                if S[i] == T[j]:
                    dp[i + 1][j + 1] = dp[i][j + 1] + dp[i][j]# 而相等的时候还要加上S字符串减1包含T字符串减1的个数
                else:
                    dp[i + 1][j + 1] = dp[i][j + 1]# 当不等的时候毫无疑问，等于S字符串减1的包含T的个数
        for x in dp:
            print(x)
        return dp[len(S)][len(T)]
```

83、统计比给定整数小的数

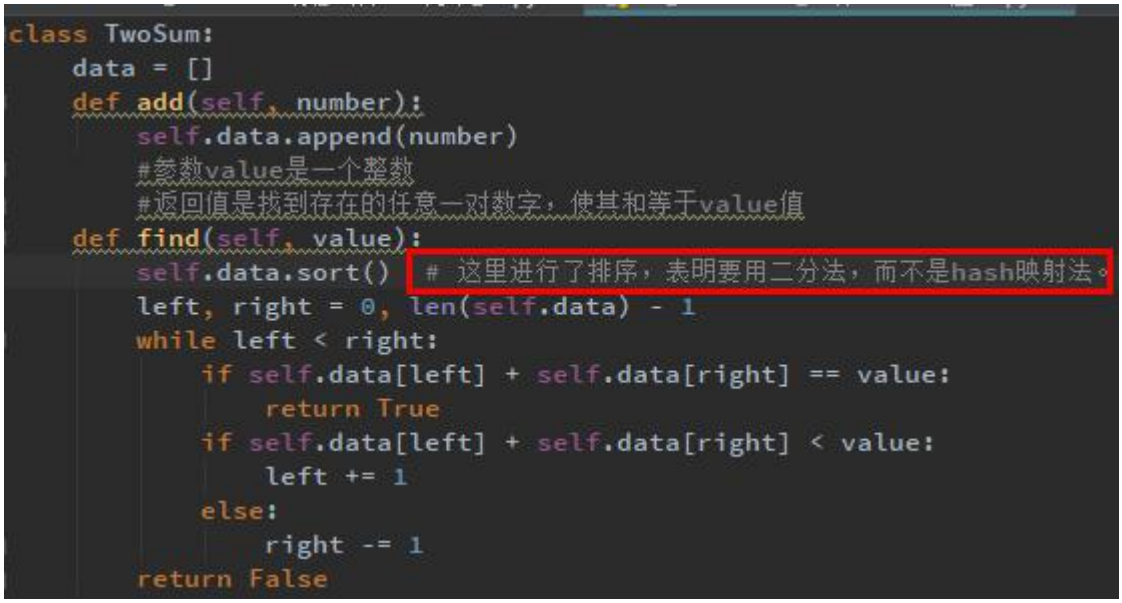
给定一个整数数组（数组长度为 n，元素的取值范围为 0-10000），以及一个查询列表。每一个查询都会给出一个整数，本例将返回数组中小于该给定整数的元素数量。

```python
class Sollution:
    def countSmaller(self, A, q):  # 在有序列表中找到一个数
        left, right = 0, len(A) - 1
        while left + 1 < right:
            mid = (left + right) // 2
            if A[mid] < q:
                left = mid
            else:
                right = mid
        if left == 0:  # 只有这种情况，left才可能等于要查找的数，不然要查找的数一定是大于这个位置的值的
            return 0
        return left + 1

    def countOfSmallerNumber(self, A, queries):
        ans = []
        for i in queries:
            ans.append(self.countSmaller(A, i))
        return ans
```

84、两数之和 3

设计并实现一个 TwoSum 类，需要支持 add 和 find 操作，add 操作把这个数添加到内部的数据结构，find 操作判断是否存在任意一对数字之和等于这个值。



```python
class TwoSum:
    data = []
    def add(self, number):
        self.data.append(number)
        #参数value是一个整数
        #返回值是找到存在的任意一对数字，使其和等于value值
    def find(self, value):
        self.data.sort()  # 这里进行了排序，表明要用二分法，而不是hash映射法。
        left, right = 0, len(self.data) - 1
        while left < right:
            if self.data[left] + self.data[right] == value:
                return True
            if self.data[left] + self.data[right] < value:
                left += 1
            else:
                right -= 1
        return False
```