1、两数之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那两个整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍

> 给定 nums = [2, 7, 11, 15], target = 9
>
> 因为 nums[0] + nums[1] = 2 + 7 = 9
> 所以返回 [0, 1]

- **第一想法**：暴力解法，第一个循环遍历列表所有数，然后第二个循环从它后面开始循环，选择相加等于target的下标。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        a=len(nums)
        for i in range(a):
            for j in range(a-i-1):
                if nums[i]+nums[i+j+1]==target:
                    return [i,i+j+1]
```

- **高分解答**：

1. 解题关键主要是想找到 num2 = target - num1，是否也在 list 中，那么就需要运用以下两个方法：

   num2 in nums，返回 True 说明有戏
   nums.index(num2)，查找 num2 的索引

```
def twoSum(nums, target):
    lens = len(nums)
    j=-1
    for i in range(lens):
        if (target - nums[i]) in nums:
            if (nums.count(target - nums[i]) == 1)&(target - nums[i] ==
nums[i]):
                #如果num2=num1,且nums中只出现了一次，说明找到是num1本身。
                continue
            else:
                j = nums.index(target - nums[i],i+1) #index(x,i+1)是从
num1后的序列后找num2
                break
    if j>0:
        return [i,j]
    else:
        return []       #执行通过，不过耗时较长，共 1636ms。
```

2. 解题思路是在方法一的基础上，优化解法。想着，num2 的查找并不需要每次从 nums 查找一遍，只需要从 num1 位置之前或之后查找即可。但为了方便 index 这里选择从 num1 位置之前查找：

```
def twoSum(nums, target):
    for i in range(1, len(nums)):
        temp = nums[:i]
        num1 = target - nums[i]
        if num1 in temp:
            j = temp.index(num1)
            return [j,i]
```

3. 通过哈希来求解，这里通过字典来模拟哈希查询的过程。
个人理解这种办法相较于方法一其实就是字典记录了 num1 和 num2 的值和位置，而省了再查找 num2 索引的步骤。

```
def two_sum(nums, target):
    """这样写更直观，遍历列表同时查字典"""
    dct = {}
    for i, n in enumerate(nums):
        cp = target - n
        if cp in dct:
            return [dct[cp], i]
        else:
            dct[n] = i
```

4. 类似方法二，不需要 `mun2` 不需要在整个 `dict` 中去查找。可以在 `num1` 之前的 `dict` 中查找，因此就只需要一次循环可解决。

```
def two_sum(nums, target):
    dct = {}
    for i, n in enumerate(nums):
        if target - n in dct:
            return [dct[target - n], i]
        dct[n] = i
```

20、有效的括号

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。
注意空字符串可被认为是有效字符串。

> 输入: "()"
> 输出: true
>
> 输入: "()[]{}"
> 输出: true

> 输入: "(]"
> 输出: false
>
> 输入: "([)]"
> 输出: false
>
> 输入: "{[]}"
> 输出: true

- **第一想法**：通过计算左括号右括号数量返回。没有考虑括号的出现顺序，导致运行错误

```
class Solution:
    def isValid(self, s: str) -> bool:
        a=0
        b=0
        c=0
        for i in s:
            if i=='(':
                a+=1
            elif i=='[':
                b+=1
            elif i=='{':
                c+=1
            elif i==')':
                a-=1
            elif i==']':
                b-=1
            elif i=='}':
                c-=1
        if a==0 and b==0 and c==0:
            return True
        else:
            return False
```

- **高分解答：**
  - 算法原理
    - 栈先入后出特点恰好与本题括号排序特点一致，即若遇到左括号入栈，遇到右括号时将对应栈顶左括号出栈，则遍历完所有括号后 stack 仍然为空；
    - 建立哈希表 dic 构建左右括号对应关系：key 左括号，value 右括号；这样查询 2个括号是否对应只需 O(1) 时间复杂度；建立栈 stack，遍历字符串 s 并按照算法流程一一判断。
  - 算法流程

1. 如果 c 是左括号，则入栈 push；
2. 否则通过哈希表判断括号对应关系，若 stack 栈顶出栈括号 stack.pop() 与当前遍历括号 c 不对应，则提前返回 false。

  - 提前返回 false
  - 提前返回优点： 在迭代过程中，提前发现不符合的括号并且返回，提升算法效率。
    - 解决边界问题：
      - 栈 stack 为空： 此时 stack.pop() 操作会报错；因此，我们采用一个取巧方法，给 stack 赋初值？，并在哈希表 dic 中建立 key: '?'，value:'?'的对应关系予以配合。此时当 stack 为空且 c 为右括号时，可以正常提前返回 false；

        - 字符串 s 以左括号结尾： 此情况下可以正常遍历完整个 s，但 stack 中遗留未出栈的左括号；因此，最后需返回 len(stack) == 1，以判断是否是有效的括号组合。
    - 复杂度分析
        - 时间复杂度 O(N)O(N)：正确的括号组合需要遍历 1 遍 s；
        - 空间复杂度 O(N)O(N)：哈希表和栈使用线性的空间大小。

```
class Solution:
    def isValid(self, s: str) -> bool:
        dic = {'{': '}',  '[': ']', '(': ')', '?': '?'}
        stack = ['?']
        for c in s:
            if c in dic: stack.append(c)        #左括号入栈
            elif dic[stack.pop()] != c: return False        #右括号与出栈的符号不一样则输出False
        return len(stack) == 1
```

    - 五行代码：

```
if len(s)%2 != 0:
            return False
        while '()' in s or '[]' in s or '{}' in s:
            s = s.replace('[]','').replace('()','').replace('{}','')
        return True if s == '' else False
```

21、合并两个有序链表

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

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

- **第一想法：**

```

```

- **高分解答：**

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

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

2. 提前截止运算

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

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

53、最大子序和

给定一个整数数组 `nums` ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

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

- **第一想法：**：暴力破解，枚举所有可能性，但是超出时间限制

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        max_n=min(nums)
        len_n=len(nums)
        if len_n==1:
            return nums[0]
        for i in range(len_n):
            for j in range(len_n-i):
```

```
            a=nums[j]
            for k in range(i):
                a+=nums[j+k+1]
            if a>max_n:
                max_n=a
        return max_n
```

- **高分解答：**

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

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        tmp = nums[0]
        max_ = tmp
        n = len(nums)
        for i in range(1,n):
            # 当当前序列加上此时的元素的值大于tmp的值，说明最大序列和可能出现在后续序列中，记录此时的最大值
            if tmp + nums[i]>nums[i]:
                max_ = max(max_, tmp+nums[i])
                tmp = tmp + nums[i]
            else:
            #当tmp(当前和)小于下一个元素时，当前最长序列到此为止。以该元素为起点继续找最大子序列，
            # 并记录此时的最大值
                max_ = max(max_, tmp, tmp+nums[i], nums[i])
                tmp = nums[i]
        return max_
```

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

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

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

```
            max_r = max(tmp, max_r)
        #返回三个中的最大值
        return max(max_right,max_left,max_l+max_r)
```

617、合并二叉树

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点

> 输入:
>
> ```
>   Tree 1            Tree 2
>    1                2
>   / \              / \
>  3   2            1   3
> /                  \   \
> 5                   4   7
> ```
>
> 输出:
> 合并后的树:
>
> ```
>      3
>     / \
>    4   5
>   / \   \
>  5   4   7
> ```

- **高分解答：**

  1. 递归：如果当前两个树都为非空，则值相加，若有一个为空，则返回另外一个。

     ```
     class Solution:
         def mergeTrees(self, t1: TreeNode, t2: TreeNode) -> TreeNode:
             if not t1 and t2:
                 return t2
             elif t1 and t2:
                 t1.val = t2.val+t1.val
                 t1.left = self.mergeTrees(t1.left,t2.left)
                 t1.right = self.mergeTrees(t1.right,t2.right)
             return t1
     ```

  2. 迭代：

     1. 首先判断边界条件，两个树有一个为空则返回另一个
     2. 初始化栈，根结点成对入栈
     3. pop 栈顶，对该节点进行合并
     4. 处理左右子树，这里需要注意的是，入栈的节点对必须都是有值的，否则下一轮合并遇到 None 值，相当于 None += val，对应的子树就断开了，所以这里需要处理左右子节点为空的情况，都为非空才入栈
     5. 迭代终止条件是栈为空
     6. 返回和存储的树

     ```
     
     ```

```
def mergeTrees(self, t1: TreeNode, t2: TreeNode) -> TreeNode:
    if not t1 or not t2:
        return t1 or t2
    stack = [(t1, t2)]
    while stack:
        node1, node2 = stack.pop()
        node1.val += node2.val
        if node1.left:
            if node2.left:
                stack.append((node1.left, node2.left))
        else:
            if node2.left:
                node1.left = node2.left

        if node1.right:
            if node2.right:
                stack.append((node1.right, node2.right))
        else:
            node1.right = node2.right
    return t1
```

461、汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 `x` 和 `y`，计算它们之间的汉明距离。

> 输入: x = 1, y = 4
>
> 输出: 2
>
> 解释:
> 1   (0 0 0 1)
> 4   (0 1 0 0)
>   ↑   ↑

- **第一想法**：转换为二进制，然后从最后一位开始比较，不同的话s加1，超过最短数之后，有1的话s加1.

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        x1=str(bin(x))[2:]
        y1=str(bin(y))[2:]
        s=0
        n=max(len(x1),len(y1))
        if n==len(x1):
            x1,y1=y1,x1          #让短的永远是x1
        for i in range(n):
            if i<len(x1):
                if x1[-i-1]!=y1[-i-1]:       #能对比时，查不同
                    s+=1
            else:
```

```
                if y1[-i-1]=='1':            #超出之后，判断1的数量
                    s+=1

        return s
```

- **高分回答：**

1. 使用异或运算

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        return bin(x^y).count('1')
```

2. 列表操作

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        num = 0
        res = []
        while x != 0 or y != 0:
            res.append((x%2, y%2))
            x //= 2
            y //= 2
        for xii, yii in res:
            if xii != yii:
                num += 1
        return num
```

3. 递归运算：先求异或，在送入func递归计算1的个数

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        def func(n):
            return 0 if n <= 0 else 1 + func(n & (n-1))
        return func(x ^ y)
```

4. 异或之后求1的个数

    1. 题目可转变为求异或后1的个数， x ^ y
    2. 将异或后的值与1相与，(x ^ y) & 1，可得出该值最末位是否为1
    3. 将该值右移一位，再次进行步骤2

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        s = x ^ y
        res = 0
        while s > 0:
            res += s & 1
            s = s >> 1
        return res
```

226、翻转二叉树

翻转一棵二叉树。

> 输入：
>
> ```
>    4
>
>  / \
> 2   7
> /\  / \
> 1 3 6  9
> ```
>
> 输出：
>
> ```
>    4
>
>  /  \
> 7    2
> /\  / \
> 9 6 3  1
> ```

- **第一想法：**使用递归算法，交换左树右树的数据，直到为空

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

- **高分解法：**同样是递归，下面详解递归函数
  - 分析递归函数
    1. 函数的定义是什么?
       该函数可以翻转一棵二叉树，即将二叉树中的每个节点的左右孩子都进行互换。
    2. 函数的输入是什么?
       函数的输入是要被翻转的二叉树。
    3. 函数的输出是什么?
       返回的结果就是已经翻转后的二叉树。
  - 递归函数的写法
    1. 递归终止的条件
       当要翻转的节点是空，停止翻转，返回空节点。
    2. 返回值
       虽然对 root 的左右子树都进行了翻转，但是翻转后的二叉树的根节点不变，故返回 root 节点。
    3. 函数内容
       root 节点的新的左子树：是翻转了的 root.right => 即 root.left = invert(root.right);
       root 节点的新的右子树：是翻转了的 root.left => 即 root.right = invert(root.left);

```
class Solution(object):
    def invertTree(self, root):
        if not root:
            return
        root.left, root.right = self.invertTree(root.right),
self.invertTree(root.left)
        return root
```

104、二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

> 给定二叉树 [3,9,20,null,null,15,7],
>
> 3
>
> / \
> 9 20
> / \
> 15 7
> 返回它的最大深度 3 。

- **第一想法：**使用递归，根节点的最大深度=子节点的最大深度+1，不断嵌套，

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

- **高分解答：**

1. BFS：广度优先搜索，一层一层搜索，先搜索距离近的，从上到下遍历一次

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

2. DFS：深度优先搜索，从某个状态开始，不断地转移状态，直到无法转移状态，然后回退到前一步的状态，继续转移到其他状态，如此不断重复，直到找到最终的解。
    - 最后得到的深度不一定是最大深度，所以要用max判断
    - DFS（先序遍历）节点右孩子先入栈，左孩子再入栈`

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        # DFS
        if root is None:
            return 0
        stack = [(1, root)]
        depth = 0
        while stack:
            cur_dep, node = stack.pop()
            depth = max(depth, cur_dep)
            if node.right:
                stack.append((cur_dep+1,node.right))
            if node.left:
                stack.append((cur_dep+1,node.left))
        return depth
```

206、反转链表

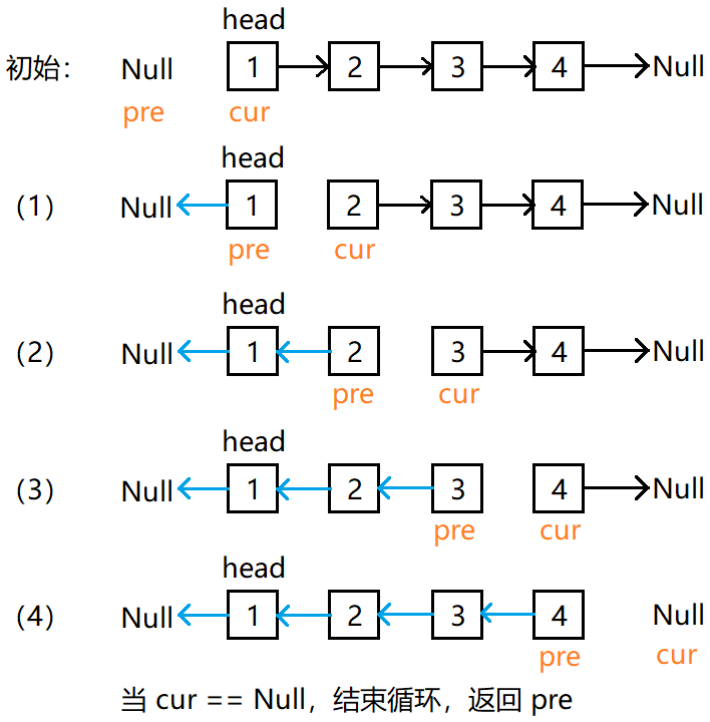
反转一个单链表。

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

- **第一想法：**递归，从后往前输出

```
```

- **高分解答：**
    1. 迭代：需要一个 `cur` 指针表示当前遍历到的节点；一个 `pre` 指针表示当前节点的前驱节点；在循环中还需要一个中间变量 `temp` 来保存当前节点的后驱节点。
        - 算法流程：
            - 首先 pre 指针指向 Null，cur 指针指向 head；
            - 当 cur != Null，执行循环。
                - 先将 cur.next 保存在 temp 中防止链表丢失：temp = cur.next
                - 接着把 cur.next 指向前驱节点 pre：cur.next = pre
                - 然后将 pre 往后移一位也就是移到当前 cur 的位置：pre = cur
                - 最后把 cur 也往后移一位也就是 temp 的位置：cur = temp
            - 当 cur == Null，结束循环，返回 pre。




```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre = None
        cur = head
        while cur:
            temp = cur.next    # 先把原来cur.next位置存起来
            cur.next = pre
            pre = cur
            cur = temp
        return pre
```

```
#迭代链表，把当前节点的指针指向上一个节点，直到调整完成
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        curr, prev = head, None
        while curr:
            prev, curr.next, curr = curr, prev, curr.next
        return prev
```

2、递归法：使用递归找到最后一个节点，然后出栈过程中调整节点的指向。

先把以head为头结点的链表记为L1，以head.next为头结点的链表记为L2

```
if not (head and head.next):
 return head
```

前两句代码是终止条件，head为空或只有一个节点返回head

```
newHead = self.reverseList(head.next)
```

这是递归语句，可以理解为已经将L2链表反转好了，假设反转得到 rL2

接下来要实现反转L1只剩下head这个节点了，于是

```
head.next.next = head
head.next = None
```

要知道head.next已经是 rL2 的末节点了，那么我们将head.next的next指针指向head，再将head的next指针指向None，就完成了 L1的反转

```python
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        ## 递归
        if not (head and head.next):
            return head

        newHead = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return newHead
```

136、只出现一次的数字

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

> ```
> 输入: [2,2,1]
> 输出: 1
> ```
>
> **示例 2:**
>
> ```
> 输入: [4,1,2,1,2]
> 输出: 4
> ```

- **第一想法:**
  - 暴力破解，从第一个数开始搜索，找到没有相同的数字——超出时间

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        for i in range(len(nums)):
            a=0
            for j in range(len(nums)):
                if nums[i]==nums[j]:
                    a+=1
            if a==1:
                return nums[i]
```

  - 从第一个开始，搜索列表中是否有相同的数字，有则将两个删除，

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        while 1:
            a=0
            b=nums[0]
            for i in range(1,len(nums)):
                if nums[0]==nums[i]:
                    nums.remove(b)
                    nums.remove(b)
                    a=1
                    break
            if a==0:
                return b
```

- **高分解答：**

1. 先排序，然后相同元素则会相邻。循环一次判断前一个元素是否等于后一个元素，间隔一个判断一个即可，如果不相等就是想要的结果。

```
class Solution:
    def singleNumber(nums):
        if len(nums)==1:    #如果数组长度为1，则直接返回即可
            return nums[0]
        nums.sort()      #对数组进行排序，使其相同元素靠在一起
        for i in range(1,len(nums),2):    #循环数组，验证前后是否相同，由于原始出现两次，因此可跳跃判断
            if nums[i-1] != nums[i]:
                return nums[i-1]
            if (i+2) == len(nums):    #判断单一元素在排序后数组的最后面
                return nums[-1]
```

2. 由于目标元素只有一次，其他元素有多次，因此，依次删除列表的元素，同一个元素删除两次，报错则为目标值。

```
class Solution:
    def singleNumber(nums):
        while True:
            d = nums[0]
            nums.remove(d)
            try:
                nums.remove(d)
            except:
                return d
```

3. 一个元素出现一次、其他出现多次，那么数组求和，与去重后的和相差的就是目标值。

```
class Solution:
    def singleNumber(nums):
        return sum(set(nums))*2-sum(nums)
```

4. 位运算

```python
from functools import reduce
class Solution:
    def singleNumber(nums):
        return reduce(lambda x, y: x ^ y, nums)      #异或运算
```

169、多数元素

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数 大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

> 输入：[3,2,3]
> 输出：3
>
> **示例 2:**
>
> 输入：[2,2,1,1,1,2,2]
> 输出：2

- **第一想法：**首先获得set后的列表，删除元素，删除次数大于len/2的数为多数元素

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        c=len(nums)
        b=set(nums)
        for i in b:
            a=0
            while 1:
                try:
                    nums.remove(i)
                    a+=1
                except:
                    if a>c/2:
                        return i
                    break
```

- **高分解答：**

1. 出现次数最多的数一定超过长度的一半，排序之后取中间的数，如果满足题意则该数一定是出现次数最多的

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

2. 一次遍历；
   记录当前值和次数；
   如果遍历值与当前值相同，times自增，不同，times自减；
   通过增减抵消的方式，最终达到剩下的数字是结果的效果，时间复杂度为 O(n)。

```
class Solution:
    def majorityElement(self, nums):
        res = nums[0]
        times = 1
        for i in range(1, len(nums)):
            if times == 0:
                res = nums[i]
                times = 1
            elif nums[i] == res:
                times += 1
            else:
                times -= 1
        return res
```

3. 用字典保存每个元素在列表中出现的次数，返回最大值。

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        #defaultdict的作用是在于，当字典里的key不存在但被查找时，
        #返回的不是keyError而是一个默认值
        dct = defaultdict(int)
        for i in nums:
            dct[i] += 1
        return max(dct.keys(), key=lambda x: dct[x])
```

283、移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

> **示例:**
>
> ```
> 输入: [0,1,0,3,12]
> 输出: [1,3,12,0,0]
> ```
>
> **说明**:
>
> 1. 必须在原数组上操作，不能拷贝额外的数组。
> 2. 尽量减少操作次数

- **第一想法：**
  1. 删除其中所有的0，删除几个在后面加几个0

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        a=0
        while 1:
            try:
                nums.remove(0)
                a+=1
            except:
                break
        for i in range(a):
            nums.append(0)
```

2. 遍历列表，如果等于0，删除，然后后面加一位0

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        for i in nums:
            if i == 0:
                nums.remove(0)
                nums.append(0)
```

- **高分解答：**

1. 判断那个数为0，然后pop，后面补0

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        zero_num = 0
        for i in range(len(nums)):
            idx = i-zero_num
            if nums[idx] == 0:
                nums.pop(idx)
                nums.append(0)
                zero_num+=1
```

2. 双指针，

```
class Solution:
    def moveZeroes(self, n: List[int]) -> None:
        j=0
        for i in range(len(n)):
            if n[i] != 0 :
                n[j] ,n[i] = n[i], n[j]
                j+=1
```

448、找到所有数组中消失的数字

给定一个范围在 1 ≤ a[i] ≤ n (n = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为O(n)的情况下完成这个任务吗? 你可以假定返回的数组不算在额外空间内。

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

- **第一想法**：获得长度，从1开始遍历，看该数是否存在数组中——超出时间

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

- **高分解答**：

1. 利用set一行解决

```
class Solution:
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        return list(set(range(1, len(nums) + 1)) - set(nums))
```

2. 对输入数组的每个值进行Counter计数

```
class Solution:
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        res=[] #返回值空间不算额外空间
        # 根据题目提示需要用到Counter函数
        c = collections.Counter(nums)
        for i in range(1, len(nums) + 1):
            if c[i] == 0:
                res.append(i)
        return res
```

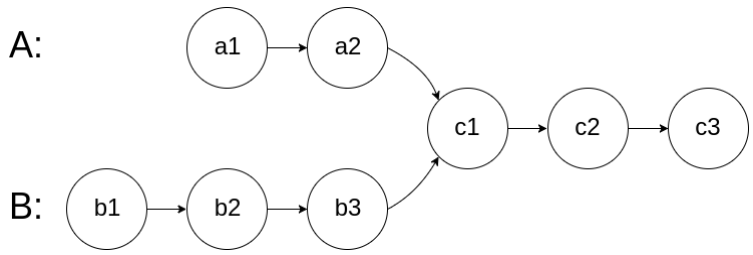
3. 将 nums 中所有正数作为索引i，置 nums[i] 为负值。仍为正数的位置即为（未出现过）消失的数字

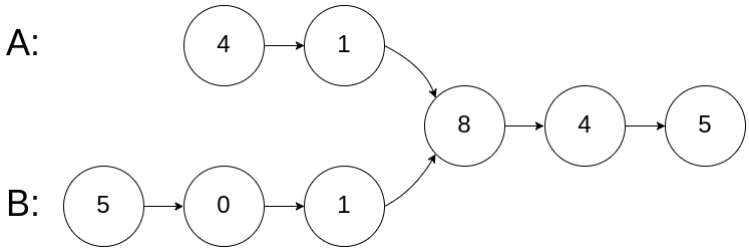
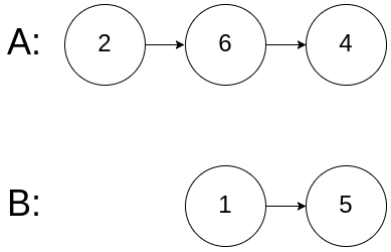
   原始数组：[4,3,2,7,8,2,3,1]
   重置后为：[-4,-3,-2,-7,8,2,-3,-1]
   结论：[8,2] 分别对应的 index 为 [5,6]（消失的数字）

```
class Solution:
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        for num in nums:
            nums[abs(num) - 1] = -abs(nums[abs(num) - 1])
        return [idx + 1 for idx, num in enumerate(nums) if num > 0]
```

160、相交链表

编写一个程序，找到两个单链表相交的起始节点。



> 示例 1：
>
> 
>
>
> 输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
> 输出：Reference of the node with value = 8
> 输入解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
>
> 示例 3：
>
> 
>
>
> 输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
> 输出：null
> 输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
> 解释：这两个链表不相交，因此返回 null。

- **高分解答：**

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

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

2. 存链表节点进行查询

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

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

3. 算出链表距离差挨个对比

    1. 分别遍历两个链表，并记录两链表的长度差n，将出现两种情况，
    (1)让长链表先走n步
    (2)再同时开始走，并对比两个链表的当前节点，节点相等时即为交点
    (3)若没有交点，则在最后的Null处相交

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

```
            cur1 = cur1.next
        while cur1 != cur2:
            cur1 = cur1.next
            cur2 = cur2.next
        return cur1
```

155、最小栈

设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。
pop() —— 删除栈顶的元素。
top() —— 获取栈顶元素。
getMin() —— 检索栈中的最小元素。

> 输入：
> ["MinStack","push","push","push","getMin","pop","top","getMin"]
> [[],[-2],[0],[-3],[],[],[],[]]
>
> 输出：
> [null,null,null,null,-3,null,0,-2]
>
> 解释：
> MinStack minStack = new MinStack();
> minStack.push(-2);
> minStack.push(0);
> minStack.push(-3);
> minStack.getMin(); --> 返回 -3.
> minStack.pop();
> minStack.top();   --> 返回 0.
> minStack.getMin(); --> 返回 -2.

- **高分解答：**

  1. 官方解答：

     - 当一个元素要入栈时，我们取当前辅助栈的栈顶存储的最小值，与当前元素比较得出最小值，将这个最小值插入辅助栈中；
     - 当一个元素要出栈时，我们把辅助栈的栈顶元素也一并弹出；
     - 在任意一个时刻，栈内元素的最小值就存储在辅助栈的栈顶元素中。

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = [math.inf]

    def push(self, x: int) -> None:
        self.stack.append(x)
        self.min_stack.append(min(x, self.min_stack[-1]))

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

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

    def getMin(self) -> int:
        return self.min_stack[-1]
```

2. 可以用一个栈，这个栈同时保存的是每个数字 x 进栈的时候的值 与 插入该值后的栈内最小值。即每次新元素 x 入栈的时候保存一个元组：（当前值 x，栈内最小值）。

   这个元组是一个整体，同时进栈和出栈。即栈顶同时有值和栈内最小值，top()函数是获取栈顶的当前值，即栈顶元组的第一个值； getMin() 函数是获取栈内最小值，即栈顶元组的第二个值；pop() 函数时删除栈顶的元组。

   每次新元素入栈时，要求新的栈内最小值：比较当前新插入元素 x 和 当前栈内最小值（即栈顶元组的第二个值）的大小。

   ```
   class MinStack(object):

       def __init__(self):
           self.stack = []

       def push(self, x):
           if not self.stack:
               self.stack.append((x, x))
           else:
               self.stack.append((x, min(x, self.stack[-1][1])))

       def pop(self):
           self.stack.pop()

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

       def getMin(self):
           return self.stack[-1][1]
   ```

121、买股票的最佳时机

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

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

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

- **高分解答：**

1. 官方：一次遍历——遍历一遍数组，计算每次 **到当天为止** 的最小股票价格和最大利润。

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

2. 官方：动态规划

    1. 动态规划做题步骤

    明确 dp(i)应该表示什么（二维情况：dp(i)(j)）；
    根据 dp(i)和 dp(i-1)的关系得出状态转移方程；
    确定初始条件，如 dp(0)。

    dp[i] 表示前 i 天的最大利润，因为我们始终要使利润最大化，则：

    dp[i] = max(dp[i-1], prices[i]-minprice)

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

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

        return dp[-1]
```

3. 类似方法1，一次遍历；最小值更新；最大收益更新；

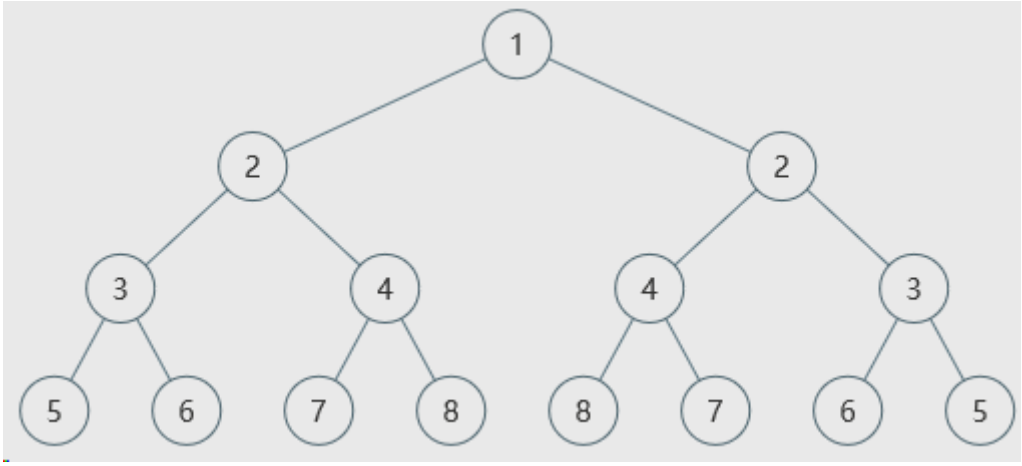
```
class Solution:
    def maxProfit(self, prices):
        res = 0
        minValue = float("inf")
        for i in range(len(prices)):
            if prices[i] < minValue:        #更新最小值
                minValue = prices[i]
            if prices[i] - minValue > res:  #更新最大收益
                res = prices[i]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

101、对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

> 例如，二叉树 [1,2,2,3,4,4,3] 是对称的。
>
> ```
>  1
> / \
> 2   2
> / \  / \
> 3  4  4  3
> ```
>
> 但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:
>
> ```
>  1
> / \
> 2   2
> \   \
> 3    3
> ```

- **高分解答：**

1. 

   首先我们对这棵树根节点的左子树进行前序遍历: pre_order = [2,3,5,6,4,7,8]

   接着我们对这棵树根节点的右子树进行后序遍历：post_order = [8,7,4,6,5,3,2]

   根据两次遍历我们不难发现 post_order 就是 pre_order的逆序，其实这也是对称二叉树的一个性质，

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

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

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

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

2. 在队列中同时取出两个节点left, right，判断这两个节点的值是否相等，然后把他们的孩子中按照(left.left, right.right) 一组，(left.right, right.left)一组放入队列中。

   BFS做法需要把所有的节点都检查完才能确定返回结果True，除非提前遇到不同的节点值而终止返回False。

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        queue = collections.deque()
        queue.append((root, root))
        while queue:
            left, right = queue.popleft()
            if not left and not right:
                continue
            if not left or not right:
                return False
            if left.val != right.val:
                return False
            queue.append((left.left, right.right))
            queue.append((left.right, right.left))
        return True
```

3. 递归：根据镜像二叉树的特点，即左右子树是对称的

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

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

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

543、二叉树的直径

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

> 给定二叉树
>
> ```
>    1
>  / \
> 2   3
> / \
> ```
>
> 4 5
> 返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

- **第一想法**：最大直径如果路线经过根节点，那就是左子树的最大深度+右子树的最大深度，同时再考虑不经过根节点的最大直径，使用max判断

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

        if root is None:
            return 0

        return max(maxDepth(root.left)+maxDepth(root.right),
                self.diameterOfBinaryTree(root.left),
                self.diameterOfBinaryTree(root.right))
```

- **高分解答**：

  1.官方：与上面类似，但是在计算最大深度的同时比较最大直径，

```python
class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.ans = 1
        def depth(root):
            if not root: return 0
            L = depth(root.left)
            R = depth(root.right)
            self.ans = max(self.ans, L + R + 1)
            return max(L, R) + 1
        depth(root)
        return self.ans - 1
```

2. 对第一思路的改进：`traverse`函数返回tuple (当前树的直径，当前树的深度)。

```python
class Solution:
    def diameterOfBinaryTree(self, root):
        def traverse(root):
            if not root:
                return 0, 0
            ldiameter, ldepth = traverse(root.left)
            rdiameter, rdepth = traverse(root.right)
            return max(ldiameter, rdiameter, ldepth + rdepth),
max(ldepth, rdepth) + 1
        return traverse(root)[0]
```

70、爬楼梯

假设你正在爬楼梯。需要 *n* 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 *n* 是一个正整数

> 示例 1：
>
> 输入： 2
> 输出： 2
> 解释： 有两种方法可以爬到楼顶。
>
> 1. 1 阶 + 1 阶
> 2. 2 阶
>
> 示例 2：
>
> 输入： 3
> 输出： 3
> 解释： 有三种方法可以爬到楼顶。
>
> 1. 1 阶 + 1 阶 + 1 阶
> 2. 1 阶 + 2 阶
> 3. 2 阶 + 1 阶

- **第一思路：**总结规律，n的输出等于（n-1的输出）+（n-2的输出），可以使用迭代或递归

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        if n==1:return 1
        if n==2:return 2
        a=1
        b=2
        for i in range(3,n+1):
            a,b=b,a+b
        return b
```

- **高分解答：**

1. 递归解法：容易超时，python可以加个缓存装饰器，这样也算是将递归转换成迭代的形式了

   除了这种方式，还有增加步长来递归，变相的减少了重复计算

   还有一种方法，在递归的同时，用数组记忆之前得到的结果，也是减少重复计算

   ```python
   class Solution:
       @functools.lru_cache(100)  # 缓存装饰器
       def climbStairs(self, n: int) -> int:
           if n == 1: return 1
           if n == 2: return 2
           return self.climbStairs(n-1) + self.climbStairs(n-2)
   ```

2. 直接DP，新建一个字典或者数组来存储以前的变量，空间复杂度O(n)

   ```python
   class Solution:
       def climbStairs(self, n: int) -> int:
           dp = {}
           dp[1] = 1
           dp[2] = 2
           for i in range(3,n+1):
               dp[i] = dp[i-1] + dp[i-2]
           return dp[n]
   ```

3. 还是DP，只不过是只存储前两个元素，减少了空间，空间复杂度O(1)，同我的第一想法
4. 直接斐波那契数列的计算公式喽

   ```python
   class Solution:
       def climbStairs(self, n: int) -> int:
           import math
           sqrt5=5**0.5
           fibin=math.pow((1+sqrt5)/2,n+1)-math.pow((1-sqrt5)/2,n+1)
           return int(fibin/sqrt5)
   ```

5. 面向测试用例编程——纯搞笑

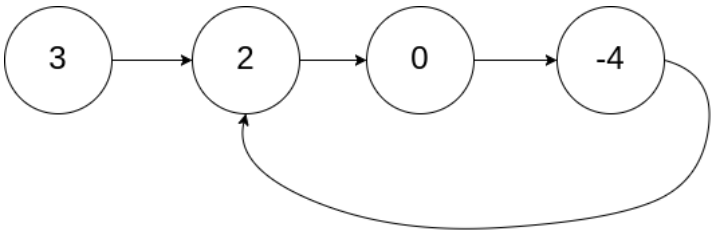
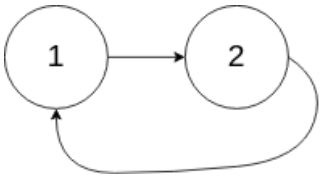
```python
class Solution:
    def climbStairs(self, n: int) -> int:
        a =
[1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,
17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269,21783
09,3524578,5702887,9227465,14930352,24157817,39088169,63245986,102334155
,165580141,267914296,433494437,701408733,1134903170,1836311903]
        return a[n-1]
```

141、环形链表

给定一个链表，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true 。 否则，返回 false 。

> 示例 1：
>
> 
>
>
> 输入：head = [3,2,0,-4], pos = 1
> 输出：true
> 解释：链表中有一个环，其尾部连接到第二个节点。
> 示例 2：
>
> 
>
>
> 输入：head = [1,2], pos = 0
> 输出：true
> 解释：链表中有一个环，其尾部连接到第一个节点。

- **第一想法：**将链表数据保存在数组中，查询链表元素是在数组中是否存在，与下面的哈希表相比，查询速度慢

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if  head  is None:
            return False
        if head.next is None:return False
        a=[]
        while head:
            a.append(head)
            if head.next in a:
                return True
            head=head.next
```

- **高分解答：**

1. hash表：使用hash表记录已经访问过的节点

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        m = {}
        while head:
            if m.get(head):
                return True
            m[head] = 1
            head = head.next
        return False
```

2. 快慢指针：枚举时添加一个慢一拍的指针，如果有环最终两个指针会相遇

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        sp, fp, si = head, head, False
        while fp:
            if si:
                sp = sp.next
                si = False
            else:
                si = True
            fp = fp.next
            if sp == fp:
                return True
        return False
```

3. 链表计数：链表中最多10000个节点，超过10000就是有环

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        count = 0
        while head and count <= 10000:
            count, head = count + 1, head.next
        return count > 10000
```

4. 链表反转：按顺序反转链表，由于环之前的部分已经反转，最后会返回到head节点

```python
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if not head or not head.next:
            return False
        p, q = None, head
        while q:
            u, p, q = p, q, q.next
            p.next = u
        return head == q
```

5. 标记val值：将原链表val值变为特殊值，若访问的节点val已经被变更过就是有环 python中可以给节点添加新属性不改变原链表，

```python
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        while head:
            if head.val == '1':
                return True
            head.val = '1'
            head = head.next
        return False
```

198、打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

> 示例 1：
>
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
> 偷窃到的最高金额 = 1 + 3 = 4 。
>
> 示例 2：
>
> 输入：[2,7,9,3,1]
> 输出：12
> 解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
> 偷窃到的最高金额 = 2 + 9 + 1 = 12 。

- **高分解答：**

1. 动态规划

   k 个房子中最后一个房子是Hk−1，如果不偷这个房子，那么问题就变成在前 k−1 个房子中偷到最大的金额，也就是子问题 f(k-1)。如果偷这个房子，那么前一个房子H k−2显然不能偷，其他房子不受影响。那么问题就变成在前 k−2 个房子中偷到的最大的金额。两种情况中，选择金额较大的一种结果。

f(k) = max { f(k-1), H{k-1} + f(k-2) }

在写递推关系的时候，要注意写上 k=0 和 k=1 的基本情况：

- 当k=0 时，没有房子，所以 f(0) = 0。
- 当k=1 时，只有一个房子，偷这个房子即可，所以 f(1) = H_0

  这样才能构成完整的递推关系，后面写代码也不容易在边界条件上出错。

```
def rob(self, nums: List[int]) -> int:
    if len(nums) == 0:
        return 0

    N = len(nums)
    dp = [0] * (N+1)
    dp[0] = 0
    dp[1] = nums[0]
    for k in range(2, N+1):
        dp[k] = max(dp[k-1], nums[k-1] + dp[k-2])
    return dp[N]
```

空间优化后：

```
def rob(self, nums: List[int]) -> int:
    prev = 0
    curr = 0
    # 每次循环，计算“偷到当前房子为止的最大金额”
    for i in nums:
        # 循环开始时，curr 表示 dp[k-1]，prev 表示 dp[k-2]
        # dp[k] = max{ dp[k-1], dp[k-2] + i }
        prev, curr = curr, max(curr, prev + i)
        # 循环结束时，curr 表示 dp[k]，prev 表示 dp[k-1]

    return curr
```

234、回文链表

请判断一个链表是否为回文链表。

> **示例 1:**
>
> ```
> 输入: 1->2
> 输出: false
> ```
>
> **示例 2:**
>
> ```
> 输入: 1->2->2->1
> 输出: true
> ```

- **高分解答：**

  1. 将值复制到数组中后用双指针法——官方

1. 复制链表值到数组列表中。
2. 使用双指针法判断是否为回文。

```
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        vals = []
        current_node = head
        while current_node is not None:
            vals.append(current_node.val)
            current_node = current_node.next
        return vals == vals[::-1]
```

2. 递归：如果使用递归反向迭代节点，同时使用递归函数外的变量向前迭代，就可以判断链表是否为回文。

   currentNode 指针是先到尾节点，由于递归的特性再从后往前进行比较。frontPointer 是递归函数外的指针。若 currentNode.val != frontPointer.val 则返回 false。反之，frontPointer 向前移动并返回 true。

```
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:

        self.front_pointer = head

        def recursively_check(current_node=head):
            if current_node is not None:
                if not recursively_check(current_node.next):
                    return False
                if self.front_pointer.val != current_node.val:
                    return False
                self.front_pointer = self.front_pointer.next
            return True

        return recursively_check()
```

3. 快慢指针：我们可以将链表的后半部分反转（修改链表结构），然后将前半部分和后半部分进行比较。比较完成后我们应该将链表恢复原样。虽然不需要恢复也能通过测试用例，但是使用该函数的人通常不希望链表结构被更改。

   1. 找到前半部分链表的尾节点。
   2. 反转后半部分链表。
   3. 判断是否回文。
   4. 恢复链表。
   5. 返回结果。

```
class Solution:

    def isPalindrome(self, head: ListNode) -> bool:
        if head is None:
            return True

        # 找到前半部分链表的尾节点并反转后半部分链表
        first_half_end = self.end_of_first_half(head)
        second_half_start = self.reverse_list(first_half_end.next)

        # 判断是否回文
```

```
        result = True
        first_position = head
        second_position = second_half_start
        while result and second_position is not None:
            if first_position.val != second_position.val:
                result = False
            first_position = first_position.next
            second_position = second_position.next

        # 还原链表并返回结果
        first_half_end.next = self.reverse_list(second_half_start)
        return result

    def end_of_first_half(self, head):
        fast = head
        slow = head
        while fast.next is not None and fast.next.next is not None:
            fast = fast.next.next
            slow = slow.next
        return slow

    def reverse_list(self, head):
        previous = None
        current = head
        while current is not None:
            next_node = current.next
            current.next = previous
            previous = current
            current = next_node
        return previous
```

4. 堆栈：遍历链表，把每个节点都push到stack中，再次遍历列表，同时节点依次出栈

```
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        stack = []
        # step1: push
        curr = head
        while(curr):
            stack.append(curr)
            curr = curr.next
        # step2: pop and compare
        node1 = head
        while(stack):
            node2 = stack.pop()
            if node1.val != node2.val:
                return False
            node1 = node1.next
        return True
```

改进：一半堆栈

```
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        stack = []
        slow = head
```

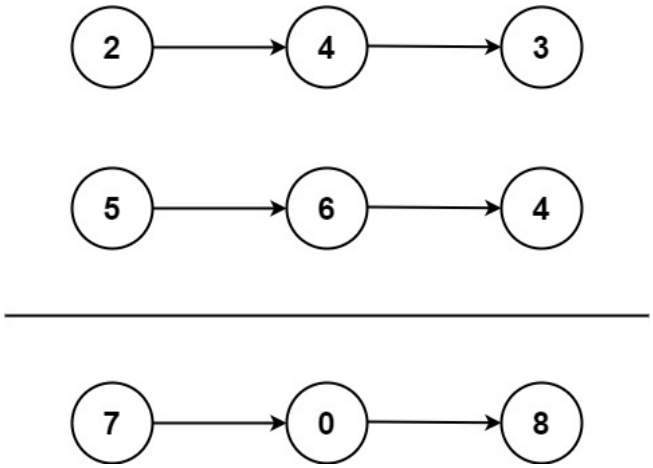
```
        fast = head
        # step1: push
        while(fast and fast.next):
            stack.append(slow)
            slow = slow.next
            fast = fast.next.next
        if fast: # 奇数
            slow = slow.next
        # step2: pop and compare
        while(stack):
            curr = stack.pop()
            if curr.val != slow.val:
                return False
            slow = slow.next
        return True
```

2、两数相加

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

> 示例 1：
>
> 
>
>
> 输入：l1 = [2,4,3], l2 = [5,6,4]
> 输出：[7,0,8]
> 解释：342 + 465 = 807.
> 示例 2：
>
> 输入：l1 = [0], l2 = [0]
> 输出：[0]
> 示例 3：
>
> 输入：l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
> 输出：[8,9,9,9,0,0,0,1]

- **第一想法：**将两个链表保存在数组，然后相加保存在新的链表中

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        a=[]
        b=[]
        s=0
        ru=ListNode()
        head=ru
        while l1:
            a.append(l1.val)
            l1=l1.next
        while l2:
            b.append(l2.val)
            l2=l2.next
        if len(a)>len(b):
            a,b=b,a
        for i in range(len(b)):
            if i<len(a):
                x=(a[i]+b[i]+s)%10
                s=(a[i]+b[i]+s)//10
                ru.next=ListNode(int(x))
                ru=ru.next
            else:
                x=(b[i]+s)%10
                s=(b[i]+s)/10
                ru.next=ListNode(int(x))
                ru=ru.next
        if s==1:
            ru.next=ListNode(1)
        return head.next
```

- **高分解答：**

1. 遍历，对上面解答的改进

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        # 创建一个结点值为 None 的头结点，dummy 和 p 指向头结点，
        # dummy 用来最后返回，p 用来遍历
        dummy = p = ListNode(None)
        s = 0               # 初始化进位 s 为 0
        while l1 or l2 or s:
        # 如果 l1 或 l2 存在，则取l1的值 + l2的值 + s(s初始为0，如果下面有进位
1，下次加上)
            s += (l1.val if l1 else 0) + (l2.val if l2 else 0)
            p.next = ListNode(s % 10)       # p.next 指向新链表，用来创建一
个新的链表
            p = p.next                      # p 向后遍历
            s //= 10                    # 有进位情况则取模，eg. s = 18，18
// 10 = 1
            l1 = l1.next if l1 else None    # 如果l1存在，则向后遍历，否则为
None
            l2 = l2.next if l2 else None    # 如果l2存在，则向后遍历，否则为
None
        return dummy.next   # 返回 dummy 的下一个节点，因为 dummy 指向的是空
的头结点，
                            # 下一个节点才是新建链表的后序节点
```

2. 递归：

    1. 因为两个数字相加会产生进位，所以使用i来保存进位。
    2. 则当前位的值为(l1.val + l2.val + i) % 10
    3. 则进位值为(l1.val + l2.val + i) / 10
    4. 建立新node，然后将进位传入下一层。

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        def dfs(l, r, i):
            if not l and not r and not i: return None
            s = (l.val if l else 0) + (r.val if r else 0) + i
            node = ListNode(s % 10)
            node.next = dfs(l.next if l else None, r.next if r else
None, s // 10)
            return node
        return dfs(l1, l2, 0)
```

3、无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

> 示例 1:
>
> 输入: s = "abcabcbb"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
> 示例 2:
>
> 输入: s = "bbbbb"
> 输出: 1
> 解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
> 示例 3:
>
> 输入: s = "pwwkew"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
> 请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
> 示例 4:
>
> 输入: s = ""
> 输出: 0

- **第一想法：**

    1. 暴力破解，选取不同长度窗口，滑动窗口，与set后对比，长度相等就意味着没有重复字符——超时间

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        if len(s)==1:
            return 1
        x=0
        for i in range(1,len(s)+1):
            for j in range(len(s)+1-i):
                a=s[j:j+i]
                b=set(a)
                if len(a)==len(b):
                    x=max(i,x)
        print(len(s))
        return x
```

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

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

- **高分解答：**

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

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # 哈希集合，记录每个字符是否出现过
        occ = set()
        n = len(s)
        # 右指针，初始值为 -1，相当于我们在字符串的左边界的左侧，还没有开始移动
        rk, ans = -1, 0
        for i in range(n):
            if i != 0:
                # 左指针向右移动一格，移除一个字符
```

```
                occ.remove(s[i - 1])
            while rk + 1 < n and s[rk + 1] not in occ:
                # 不断地移动右指针
                occ.add(s[rk + 1])
                rk += 1
            # 第 i 到 rk 个字符是一个极长的无重复字符子串
            ans = max(ans, rk - i + 1)
        return ans
```

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

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

5、最长回文子串

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

> 示例 1：
>
> 输入：s = "babad"
> 输出："bab"
> 解释："aba" 同样是符合题意的答案。
> 示例 2：
>
> 输入：s = "cbbd"
> 输出："bb"
> 示例 3：
>
> 输入：s = "a"
> 输出："a"
> 示例 4：
>
> 输入：s = "ac"
> 输出："a"

- **第一想法**：双指针，一个指针从前开始遍历，另一个指针往后指定数组范围a，判断是否为回文子串，re保存最长的回文子串

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        re=''
        if len(s)==1:
            return s
        if len(s)==2:
            if s[0]==s[1]:
                return s
            else:
                return s[0]
        for i in range(len(s)):
            a=''
            for j in range(i,len(s)):
                a+=s[j]
                if a==a[::-1] and len(re)<len(a):
                    re=a
        return re
```

- **高分解答：**

1. 动态规划：对于一个子串而言，如果它是回文串，并且长度大于 22，那么将它首尾的两个字母去除之后，它仍然是个回文串。

   我们用 P(i,j) 表示字符串 s** 的第 i 到 j 个字母组成的串（下文表示成 s[i:j]]）是否为回文串：

$$P(i,j)=\begin{cases}\text{true}, & \text{如果子串 } S_i \dots S_j \text{ 是回文串}\\ \text{false}, & \text{其它情况}\end{cases}$$

   这里的「其它情况」包含两种可能性：

   - s[i, j] 本身不是一个回文串；
   - i > j，此时 s[i, j] 本身不合法。

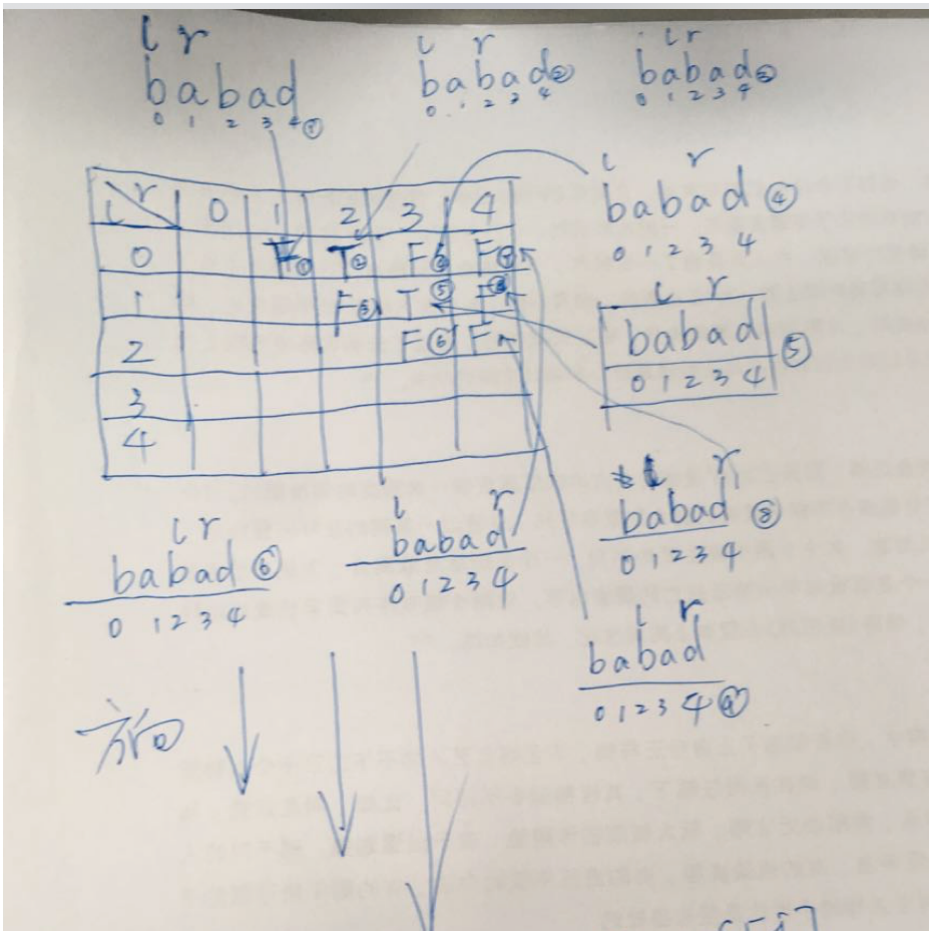
   那么我们就可以写出动态规划的状态转移方程：

$$P(i,j)=P(i+1,j-1)\wedge(S_i==S_j)$$

   上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$\begin{cases}P(i,i)=\text{true}\\ P(i,i+1)=(S_i==S_{i+1})\end{cases}$$

   最终的答案即为所有 P(i, j) = true 中 j−i+1（即子串长度）的最大值。注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序




```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        dp = [[False] * n for _ in range(n)]
        ans = ""
        # 枚举子串的长度 l+1
        for l in range(n):
            # 枚举子串的起始位置 i，这样可以通过 j=i+l 得到子串的结束位置
            for i in range(n):
                j = i + l
                if j >= len(s):
                    break
                if l == 0:
                    dp[i][j] = True
                elif l == 1:
                    dp[i][j] = (s[i] == s[j])
                else:
                    dp[i][j] = (dp[i + 1][j - 1] and s[i] == s[j])
                if dp[i][j] and l + 1 > len(ans):
                    ans = s[i:j+1]
        return ans
```

2. 中心扩展算法：**所有的状态在转移的时候的可能性都是唯一的。**也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

   我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

```python
class Solution:
    def expandAroundCenter(self, s, left, right):
```

```
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    def longestPalindrome(self, s: str) -> str:
        start, end = 0, 0
        for i in range(len(s)):
            left1, right1 = self.expandAroundCenter(s, i, i)
            left2, right2 = self.expandAroundCenter(s, i, i + 1)
            if right1 - left1 > end - start:
                start, end = left1, right1
            if right2 - left2 > end - start:
                start, end = left2, right2
        return s[start: end + 1]
```

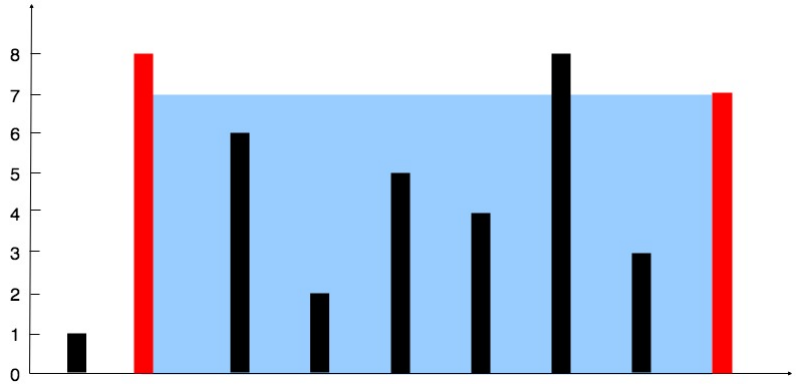
3. Manacher 算法
4. 因为题目要的最长回文，所以不用验证更短的子串了。或者说，就是以尾座标为标志进行验证，也不用担心错过更长的结果。效果是只需遍历一次。时间空间都是双百。

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        if not s: return ""
        length = len(s)
        if length == 1 or s == s[::-1]: return s
        max_len,start = 1,0

        for i in range(1, length):
            even = s[i-max_len:i+1]
            odd = s[i-max_len-1:i+1]
            if i - max_len - 1 >= 0 and odd == odd[::-1]:
                start = i - max_len - 1
                max_len += 2
                continue
            if i - max_len >= 0 and even == even[::-1]:
                start = i - max_len
                max_len += 1
                continue
        return s[start:start + max_len]
```

> **abccbad**
>
> | i | maxlen | start | even | odd |
> |---|---|---|---|---|
> | 1 | 1 | 0 | ab | dab |
> | 2 | 1 | 0 | bc | abc |
> | 3 | 1 | 0 | cc | bcc |
> | 4 | 2 | 2 | ccb | bccb |
> | 5 | 4 | 1 | bccba | abccba |
> | 6 | 6 | 0 | abccbad | abccbad |

11、盛最多水的容器

给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0) 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

> 示例 1：
>
> 
>
>
> 输入：[1,8,6,2,5,4,8,3,7]
> 输出：49
> 解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
> 示例 2：
>
> 输入：height = [1,1]
> 输出：1
> 示例 3：
>
> 输入：height = [4,3,2,1,4]
> 输出：16
> 示例 4：
>
> 输入：height = [1,2,1]
> 输出：2

- **第一想法：**
  1. 最容易理解的就是枚举所有可能性，使用两个循环，找出最大值——超时间

```
class Solution:
    def maxArea(self, height: List[int]) -> int:
        maxre=0
        for i in range(len(height)):
            for j in range(i+1,len(height)):
                s=(j-i)*(height[i] if height[i]<height[j] else
height[j])
                if s>maxre:
                    maxre=s

        return maxre
```

  2. 双指针，指针在数组两端，向里收缩，直至指针指到同一个数。

向内收窄短板可以获取面积最大值

```
class Solution:
    def maxArea(self, height: List[int]) -> int:
        sta,end,maxre=0,len(height)-1,0
        while sta<end:
            if height[sta]<height[end]:
                s=(end-sta)*height[sta]
                sta+=1
            else:
                s=(end-sta)*height[end]
                end-=1
            if s>maxre:
                maxre=s

        return maxre
```

15、三数之和

给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？请你找出所有和为 0 且不重复的三元组。

> 示例 1：
>
> 输入：nums = [-1,0,1,2,-1,-4]
> 输出：[[-1,-1,2],[-1,0,1]]
> 示例 2：
>
> 输入：nums = []
> 输出：[]
> 示例 3：
>
> 输入：nums = [0]
> 输出：[]

- **第一想法：**把数组分成正集、负集、0的字典，分别考虑不同情况，得到最终数组
  - 字典的值应该是该数出现的次数
  - 0有3个以上时，[0,0,0]；
  - 0的个数大于0，分析一正一负+0的情况
  - 两个相同正数+一个负数、一个正数+两个相同负数
  - 两个不同正数+一个负数、一个正数+两个相同负数

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        negative = dict()
        positive = dict()
        zero = 0
        for num in nums:
            if num > 0:
                if num in positive.keys():
                    positive[num] += 1
                else:
```

```
                positive[num] = 1
            elif num < 0:
                if num in negative.keys():
                    negative[num] += 1
                else:
                    negative[num] = 1
            else:
                zero += 1
        result = []
        if zero >= 3:
            result.append([0, 0, 0])
        if zero > 0:
            for num in negative.keys():
                if (-num) in positive.keys():
                    result.append([num, 0, -num])
        for num in negative.keys():
            if negative[num] >= 2 and (-num) * 2 in positive.keys():
                result.append([num, num, - num * 2])
        for num in positive.keys():
            if positive[num] >= 2 and (-num) * 2 in negative.keys():
                result.append([-num * 2, num, num])
        keysOfNegative = list(negative.keys())
        for num1Index in range(len(keysOfNegative) - 1):
            num1 = keysOfNegative[num1Index]
            for num2 in keysOfNegative[num1Index + 1:]:
                if -(num1 + num2) in positive.keys():
                    result.append([num1, num2, -(num1 + num2)])
        keysOfPositive  = list(positive.keys())
        for num1Index in range(len(keysOfPositive) - 1):
            num1 = keysOfPositive[num1Index]
            for num2 in keysOfPositive[num1Index + 1:]:
                if -(num1 + num2) in negative.keys():
                    result.append([-(num1 + num2), num1, num2])
        return result
```

- **高分解答：**

1. 排序+双指针

    1. 特判，对于数组长度 n，如果数组为 null 或者数组长度小于 3，返回 []。
    2. 对数组进行排序。
    3. 遍历排序后数组：
        - 若 nums[i]>0：因为已经排序好，所以后面不可能有三个数加和等于 0，直接返回结果。
        - 对于重复元素：跳过，避免出现重复解
        - 令左指针 L=i+1，右指针 R=n−1，当 L<R 时，执行循环：
            - 当 nums[i]+nums[L]+nums[R]==0，执行循环，判断左界和右界是否和下一位置重复，去除重复解。并同时将 L,R 移到下一位置，寻找新的解
            - 若和大于 0，说明 nums[R] 太大，R 左移
            - 若和小于 0，说明 nums[L] 太小，L 右移
              复杂度分析

```
class Solution:
```

```
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n=len(nums)
        res=[]
        if(not nums or n<3):
            return []
        nums.sort()
        for i in range(n):
            if(nums[i]>0):
                return res
            if(i>0 and nums[i]==nums[i-1]):
                continue
            L=i+1
            R=n-1
            while(L<R):
                if(nums[i]+nums[L]+nums[R]==0):
                    res.append([nums[i],nums[L],nums[R]])
                    while(L<R and nums[L]==nums[L+1]):
                        L=L+1
                    while(L<R and nums[R]==nums[R-1]):
                        R=R-1
                    L=L+1
                    R=R-1
                elif(nums[i]+nums[L]+nums[R]>0):
                    R=R-1
                else:
                    L=L+1
        return res
```

优化：使用set（）去重

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        if not nums or len(nums) <= 2:
            return[]
        nums.sort()
        res = set()
        for k in range(len(nums) - 2):
            if nums[k] > 0: break
            if k > 0 and nums[k] == nums[k - 1]: continue
            i, j = k + 1 , len(nums) - 1
            while i < j:
                sum = nums[i] + nums[j] + nums[k]
                if sum < 0: i += 1
                elif sum > 0: j -= 1
                else:
                    res.add((nums[i], nums[j], nums[k]))
                    i += 1
                    j -= 1

        return list(res)
```

17、电话号码的字母组合

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

> 输入："23"
> 输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

- **第一想法：**创建字母为键，数字为值的字典，使用回溯法找到所有组合

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        s_n={}
        result=[]
        s='abcdefghijklmnopqrstuvwxyz'
        n='22233344455566677778889999'
        for i in range(26):
            s_n[s[i]]=n[i]
        #print(s_n)
        if len(digits) ==0:
            return result

        def fn(d,digits):
            if len(digits)==0:
                result.append(d)
            else:
                for st in [k for k,v in s_n.items() if v==digits[0]]:
                    fn(d+st,digits[1:])

        fn('',digits)
        return result
```

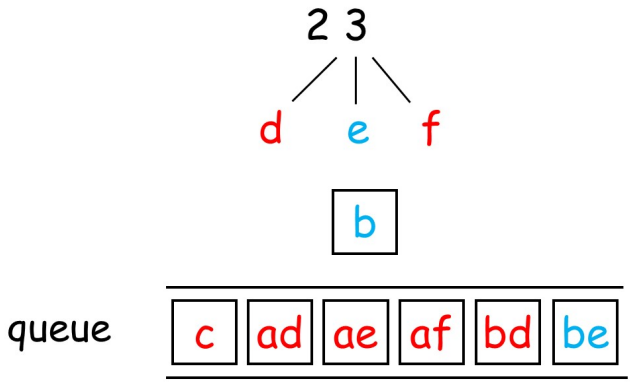
- **高分解答：**

1. 回溯法，对字典进行优化，键是数字，值是数组

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits: return []
```

```
        phone = {'2':['a','b','c'],
                 '3':['d','e','f'],
                 '4':['g','h','i'],
                 '5':['j','k','l'],
                 '6':['m','n','o'],
                 '7':['p','q','r','s'],
                 '8':['t','u','v'],
                 '9':['w','x','y','z']}
        def backtrack(conbination,nextdigit):
            if len(nextdigit) == 0:
                res.append(conbination)
            else:
                for letter in phone[nextdigit[0]]:
                    backtrack(conbination + letter,nextdigit[1:])
        res = []
        backtrack('',digits)
        return res
```

2. 队列：先将输入的 digits 中第一个数字对应的每一个字母入队，然后将出队的元素与第二个数字对应的每一个字母组合后入队...直到遍历到 digits 的结尾。最后队列中的元素就是所求结果。




```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits: return []
        phone = ['abc','def','ghi','jkl','mno','pqrs','tuv','wxyz']
        queue = ['']  # 初始化队列
        for digit in digits:
            for _ in range(len(queue)):
                tmp = queue.pop(0)
                for letter in phone[ord(digit)-50]:# 这里我们不使用 int()转换字符串，使用ASCII码
                    queue.append(tmp + letter)
        return queue
```

19、删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

> 给定一个链表: 1->2->3->4->5, 和 n = 2.
>
> 当删除了倒数第二个节点后，链表变为 1->2->3->5.

- **第一想法：**将链表保存至数组，删除数组[-n]，重新保存到链表

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        a=[]
        while head:
            a.append(head.val)
            head=head.next
        del a[-n]
        re=ListNode(None)
        reh=re
        for i in a:
            reh.next=ListNode(int(i))
            reh=reh.next

        return re.next
```

- **高分解答：**

1. 循环迭代 -- 找到 length -n 个节点

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0)
        dummy.next = head
        #step1: 获取链表长度
        cur, length = head, 0
        while cur:
            length += 1
            cur = cur.next
        #step2: 找到倒数第N个节点的前面一个节点
        cur = dummy
        for _ in range(length - n):
            cur = cur.next
        #step3: 删除节点，并重新连接
        cur.next = cur.next.next
        return dummy.next
```

2. 快慢指针 -- 找倒数第N个节点的前一个节点

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0)
        dummy.next = head
        #step1: 快指针先走n步
        slow, fast = dummy, dummy
        for _ in range(n):
```

```
            fast = fast.next
        #step2: 快慢指针同时走，直到fast指针到达尾部节点，
        #此时slow到达倒数第N个节点的前一个节点
        while fast and fast.next:
            slow, fast = slow.next, fast.next
        #step3: 删除节点，并重新连接
        slow.next = slow.next.next
        return dummy.next
```

3. 递归迭代 -- 回溯时，进行节点计数

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        if not head:
            self.count = 0
            return head
        head.next = self.removeNthFromEnd(head.next, n) # 递归调用
        self.count += 1 # 回溯时进行节点计数
        return head.next if self.count == n else head
```

22、括号生成

数字 *n* 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

> 输入：n = 3
> 输出：[
> "((()))",
> "(()())",
> "(())()",
> "()(())",
> "()()()"
>]

- **高分解答：**

  1.暴力法：我们可以生成所有 2^{2n} 个 `'('` 和 `')'` 字符构成的序列，然后我们检查每一个是否有效即可。

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        def generate(A):
            if len(A) == 2*n:
                if valid(A):
                    ans.append("".join(A))
            else:
                A.append('(')
                generate(A)
                A.pop()
                A.append(')')
                generate(A)
                A.pop()
        def valid(A):
```

```
            bal = 0
            for c in A:
                if c == '(': bal += 1
                else: bal -= 1
                if bal < 0: return False
            return bal == 0
        ans = []
        generate([])
        return ans
```

2. 回溯法：我们可以只在序列仍然保持有效时才添加 `'('` or `')'`。

   如果左括号数量不大于 n*n*，我们可以放一个左括号。如果右括号数量小于左括号的数量，我们可以放一个右括号。

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        ans = []
        def backtrack(S, left, right):
            if len(S) == 2 * n:
                ans.append(''.join(S))
                return
            if left < n:
                S.append('(')
                backtrack(S, left+1, right)
                S.pop()
            if right < left:
                S.append(')')
                backtrack(S, left, right+1)
                S.pop()

        backtrack([], 0, 0)
        return ans
```

3. 按括号序列的长度递归：任何一个括号序列都一定是由 `(` 开头，并且第一个 `(` 一定有一个唯一与之对应的 `)`。这样一来，每一个括号序列可以用 `(a)b` 来表示，其中 a 与 b 分别是一个合法的括号序列（可以为空）。

   那么，要生成所有长度为 2 * n 的括号序列，我们定义一个函数 generate(n) 来返回所有可能的括号序列。那么在函数 generate(n) 的过程中：

   - 我们需要枚举与第一个 (对应的) 的位置 2 * i + 1；
   - 递归调用 generate(i) 即可计算 a 的所有可能性；
   - 递归调用 generate(n - i - 1) 即可计算 b 的所有可能性；
   - 遍历 a 与 b 的所有可能性并拼接，即可得到所有长度为 2 * n 的括号序列。

```
class Solution:
    @lru_cache(None)
    def generateParenthesis(self, n: int) -> List[str]:
        if n == 0:
            return ['']
        ans = []
        for c in range(n):
            for left in self.generateParenthesis(c):
                for right in self.generateParenthesis(n-1-c):
                    ans.append('({}){}'.format(left, right))
        return ans
```

4. 动态规划：当我们清楚所有 i 时括号的可能生成排列后，对与i=n的情况，我们考虑整个括号排列中最左边的括号。

   剩下的括号要么在这一组新增的括号内部，要么在这一组新增括号的外部（右侧）。

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        dp = [[] for _ in range(n+1)]          # dp[i]存放i组括号的所有有效组合
        dp[0] = [""]                           # 初始化dp[0]
        for i in range(1, n+1):                # 计算dp[i]
            for p in range(i):                 # 遍历p
                l1 = dp[p]                     # 得到dp[p]的所有有效组合
                l2 = dp[i-1-p]                 # 得到dp[q]的所有有效组合
                for k1 in l1:
                    for k2 in l2:
                        dp[i].append("({0}){1}".format(k1, k2))

        return dp[n]
```

5. 递推公式
   - n=0 []
   - n=1 ['()']
   - n=2 ['()()','(())']
   - n=3 ["((()))", "(()())", "(())()","()(())","()()()"]

     可知递推公式为：f(n) = f(n-1) 的基础上在不同结果中不同位置插入'()' ,然后去重

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        if n==0:
            return []

        pre = ['()',]
        for i in range(n-1):     #不断迭代到n
            now = set()
            for st in pre:       #n-1的所有结果
                n = len(st)
                for index in range(n):       #在其中一个结果的不同位置加入（）
                    now.add(st[:index]+'()'+st[index:])
            pre = now
        return list(pre)
```

31、下一个排列

实现获取 下一个排列 的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须 原地 修改，只允许使用额外常数空间。

> 输入：nums = [1,2,3]
> 输出：[1,3,2]

> 输入：nums = [3,2,1]
> 输出：[1,2,3]
>
> 输入：nums = [1,1,5]
> 输出：[1,5,1]
>
> 输入：nums = [1]
> 输出：[1]

- **高分解答：**

1. 倒序遍历+反转函数

   下一个更大的排列首先需要一个邻近的更大的数字替换自身，但同时还需要保证后面的数字组合达到最小才符合邻近标准，因此我们反其道而行之，先保证后面的数字组合达到最小，然后从标记位置向后搜索到一个大于自己的数进行对调，这样就能获取下一个排列。

```
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        index = 0
        #从后遍历，找到nums[i] > nums[i-1]的索引
        for i in range(len(nums)-1, 0, -1):
            if nums[i] > nums[i-1]:
                index = i
                break
        if index == len(nums)-1:         #如果是最后两位，直接替换
            nums[index], nums[index-1] = nums[index-1], nums[index]
        elif index == 0:         #如果是第一位，倒序输出
            self.Reverse(nums, 0, len(nums)-1)
        else:
            self.Reverse(nums, index, len(nums)-1)       #索引点后倒序
            for s in range(index, len(nums)):        #找到大于index-1的值，
替换
                if nums[s] > nums[index-1]:
                    nums[index-1], nums[s] = nums[s], nums[index-1]
                    break
    def Reverse(self, nums, start, end):
        mid = (start+end+1) // 2
        k = 0
        for j in range(start, mid):
            nums[j], nums[end-k] = nums[end-k], nums[j]
            k += 1
```

2. 字典序：

   1、从右到左，找到第一个左侧小于右侧的下标值i
   2、再次从右到左，找到第一个大于nums[i]的数以及下标j，然后i 和 j对应的数进行互换
   3、i下标后面的数按从小到大排序

```
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """

        if len(nums) < 2: return
        for i in range(len(nums) - 2, -1, -1):
```

```
            if nums[i] < nums[i + 1]:
                for j in range(len(nums) - 1, i, -1):
                    if nums[j] > nums[i]:
                        nums[i], nums[j] = nums[j], nums[i]
                        break
                nums[i + 1:] = sorted(nums[i + 1:])
                return
        return nums.sort()
```

33、搜索旋转排序数组

整数数组 nums 按升序排列，数组中的值 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， [0,1,2,4,5,6,7] 在下标 3 处经旋转后可能变为 [4,5,6,7,0,1,2] 。

给你 旋转后 的数组 nums 和一个整数 target ， 如果 nums 中存在这个目标值 target ， 则返回它的下标，否则返回 -1 。

> 输入：nums = [4,5,6,7,0,1,2], target = 0
> 输出：4

> 输入：nums = [4,5,6,7,0,1,2], target = 3
> 输出：-1

> 输入：nums = [1], target = 0
> 输出：-1

- **第一想法**

  1、搜索target是否存在于nums中，输出他的下标

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        if target in nums:
            return nums.index(target)
        else:
            return -1
```

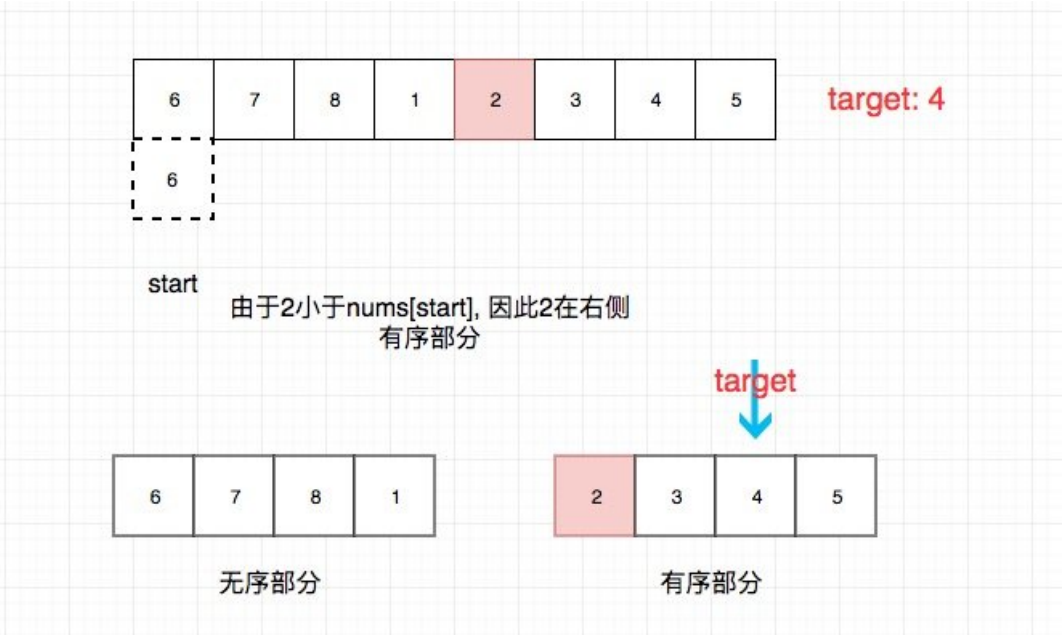
  2、遍历一次，查找target

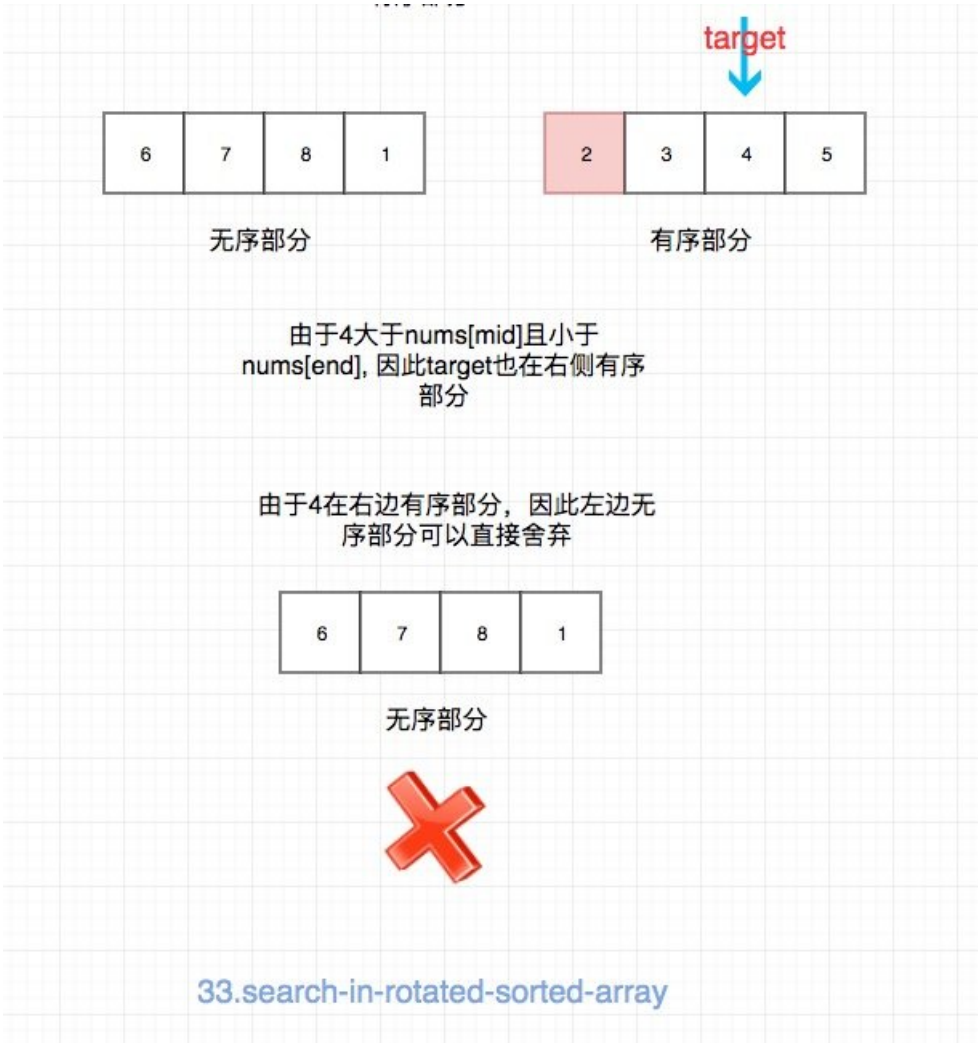
```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        for i in range(len(nums)):
            if target==nums[i]:
                return i

        return -1
```

- **高分解答**

1、二分法

1. 我们可以先找出mid，然后根据mid来判断，mid是在有序的部分还是无序的部分
    - 假如mid小于start，则mid一定在右边有序部分。
    - 假如mid大于等于start， 则mid一定在左边有序部分。
    - 然后我们继续判断target在哪一部分， 我们就可以舍弃另一部分了
2. 只需要比较target和有序部分的边界关系就行了。 比如mid在右侧有序部分，即[mid, end]
   那么我们只需要判断 target >= mid && target <= end 就能知道target在右侧有序部分，我们就
   可以舍弃左边部分了(start = mid + 1)， 反之亦然。





33.search-in-rotated-sorted-array

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        """用二分法，先判断左右两边哪一边是有序的，再判断是否在有序的列表之内"""
        if len(nums) <= 0:
            return -1

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

            # 如果中间的值大于最左边的值，说明左边有序
            if nums[mid] > nums[left]:
                if nums[left] <= target <= nums[mid]:
                    right = mid
                else:
                    # 这里 +1，因为上面是 <= 符号
                    left = mid + 1
            # 否则右边有序
            else:
                # 注意：这里必须是 mid+1，因为根据我们的比较方式，mid属于左边的序列
                if nums[mid+1] <= target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid

        return left if nums[left] == target else -1
```

34、在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 target，返回 [-1, -1]。

> 输入：nums = [5,7,7,8,8,10], target = 8
> 输出：[3,4]
>
> 输入：nums = [5,7,7,8,8,10], target = 6
> 输出：[-1,-1]
>
> 输入：nums = [], target = 0
> 输出：[-1,-1]

- **第一想法：**
    1. 使用index（）函数查找元素，首先正着查找第一个元素，然后数组倒过来查找第一个元素。使用try检测异常。

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

        return result
```

2. 使用for循环，从前遍历，获取第一个之后，从后往前遍历，获取倒数第一个

```python
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        result=[-1,-1]
        if len(nums)==0:
            return result
        for i in range(len(nums)):
            if nums[i]==target:
                result[0]=i
                break
        for i in range(len(nums)-1,result[0]-1,-1):
            if nums[i]==target:
                result[1]=i
                break
        return result
```

- **高分解答：**

1. 二分法：二分查找两边逼近的方式（时间复杂度为 O(log n)，只有使用二分法。）

```python
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        size = len(nums)
        if size == 0:
            return [-1, -1]

        first_position = self.__find_first_position(nums, size, target)
        if first_position == -1:
            return [-1, -1]
        last_position = self.__find_last_position(nums, size, target)
        return [first_position, last_position]

    def __find_first_position(self, nums, size, target):
        left = 0
        right = size - 1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] < target:
                left = mid + 1
            elif nums[mid] == target:
                right = mid
            else:
                # nums[mid] > target
                right = mid - 1
```

```
        if nums[left] == target:
            return left
        else:
            return -1

    def __find_last_position(self, nums, size, target):
        left = 0
        right = size - 1
        while left < right:
            mid = (left + right + 1) // 2
            if nums[mid] > target:
                right = mid - 1
            elif nums[mid] == target:
                left = mid
            else:
                # nums[mid] < target
                left = mid + 1

        # 由于能走到这里，说明在数组中一定找得到目标元素，因此这里不用再做一次判断
        return left
```

39、组合总和

给定一个无重复元素的数组 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。

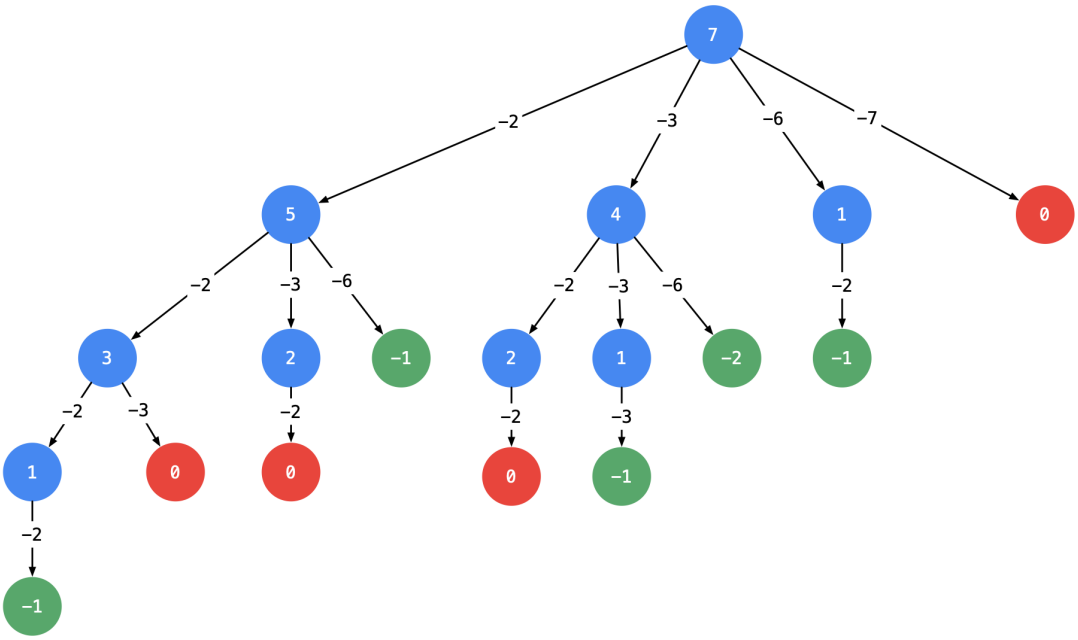
candidates 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 target）都是正整数。
- 解集不能包含重复的组合。

> 输入：candidates = [2,3,6,7], target = 7,
> 所求解集为：
> [
> [7],
> [2,2,3]]
>
> 输入：candidates = [2,3,5], target = 8,
> 所求解集为：
> [
> [2,2,2,2],
> [2,3,3],
> [3,5]]

- **高分解答：**
    1. 回溯算法+剪枝



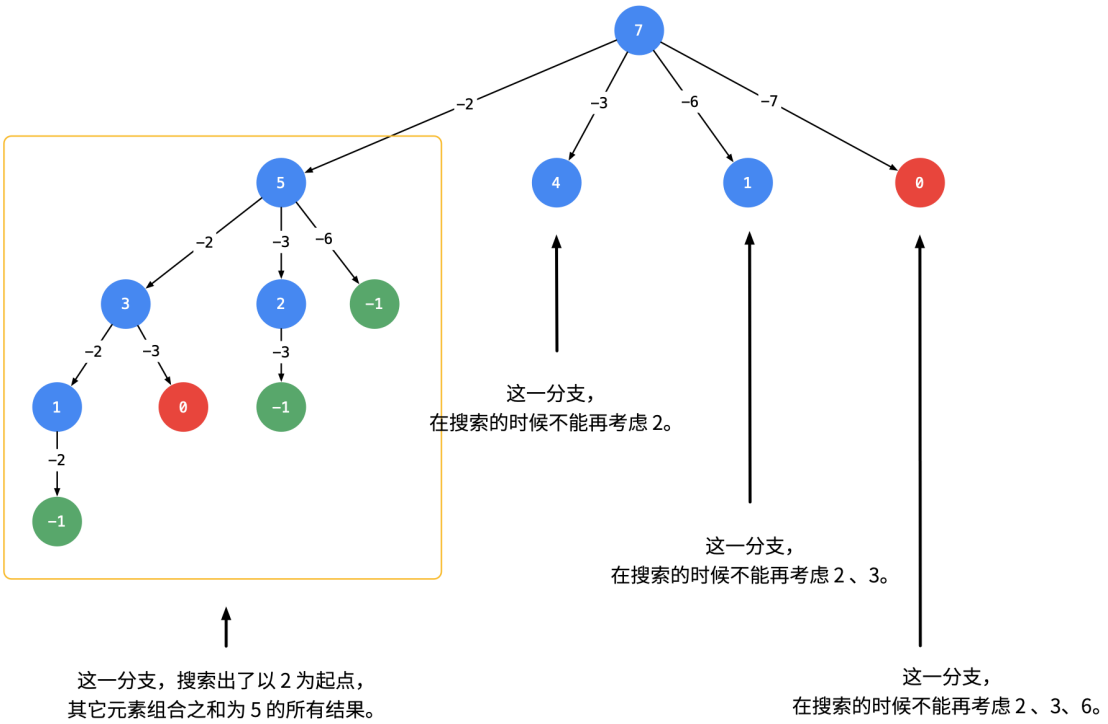
以 target = 7 为 根结点 ， 创建一个分支的时 做减法 ；

每一个箭头表示：从父亲结点的数值减去边上的数值，得到孩子结点的数值。边的值就是题目中给出的 candidate 数组的每个元素的值；

减到 0 或者负数的时候停止，即：结点 00 和负数结点成为叶子结点；

所有从根结点到结点 0的路径（只能从上往下，没有回路）就是题目要找的一个结果。

去重：



- 从每一层的第 22 个结点开始，都不能再搜索产生同一层结点已经使用过的 `candidate` 里的元素。

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
```

```
        def dfs(candidates, begin, size, path, target):
            if target == 0:
                res.append(path)
                return

            for index in range(begin, size):
                residue = target - candidates[index]
                if residue < 0:
                    break

                dfs(candidates, index, size, path + [candidates[index]],
residue)

        size = len(candidates)
        if size == 0:
            return []
        candidates.sort()
        path = []
        res = []
        dfs(candidates, 0, size, path, target)
        return res
```

2. 动态规划

求解一个值，需要求解上一个值，进而需要求解上上个值...，这种方法就是动态规划。重复的值只需要进行一次，而后再使用到这个值时，只需要将他的结果拿来适当增加即可。

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
        dict = {}
        for i in range(1,target+1):
            dict[i]=[]

        for i in range(1,target+1):
            for j in candidates:
                if i==j:
                    dict[i].append([i])
                elif i>j:
                    for k in dict[i-j]:
                        x = k[:]
                        x.append(j)
                        x.sort() # 升序，便于后续去重
                        if x not in dict[i]:
                            dict[i].append(x)

        return dict[target]
```

优化：

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
        dp = {i:[] for i in range(target+1)}

        # 这里一定要将candidates降序排列
        for i in sorted(candidates,reverse=True):
            for j in range(i,target+1):
                if j==i:
                    dp[j] = [[i]]
                else:
                    dp[j].extend([x+[i] for x in dp[j-i]])
        return dp[target]
```

46、全排列

给定一个 **没有重复** 数字的序列，返回其所有可能的全排列

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

- **高分解答**

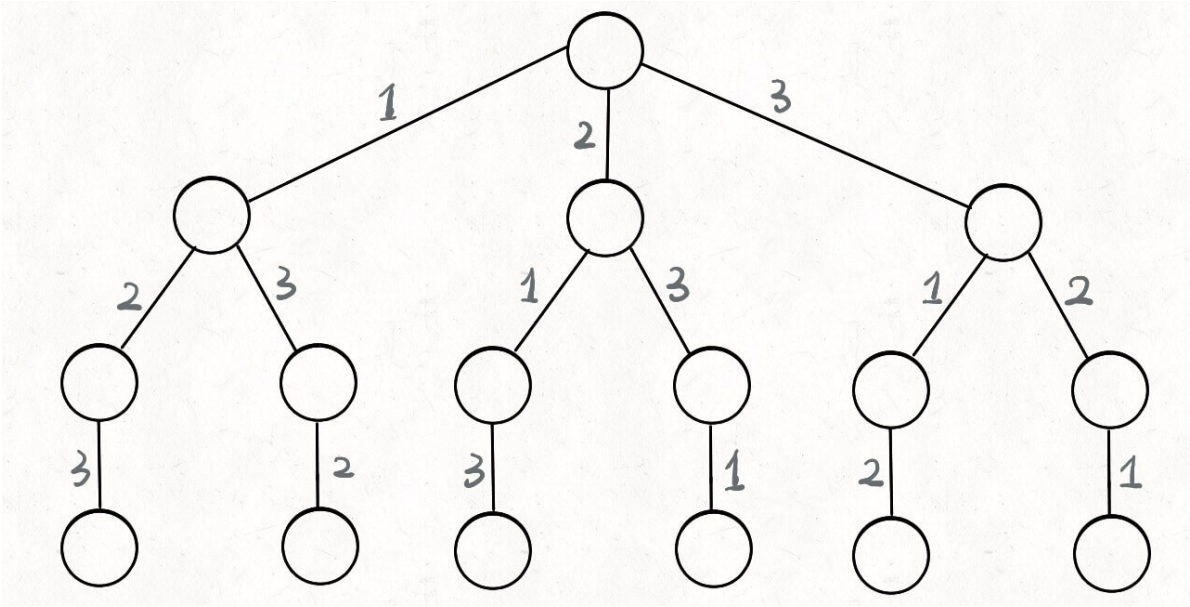
  1、回溯算法

  回溯法的**三个基本要素**：

  - 路径：已经做出的选择；
  - 选择列表：当前可以做出的选择；
  - 结束条件：结束一次回溯算法的条件，即遍历到决策树的叶节点；

  通用框架：写 backtrack 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

  ```python
  结果 = []
  def backtrack(路径, 选择列表):
      if 满足结束条件:
          结果.append(路径)
          return
      for 选择 in 选择列表:     # 核心代码段
          做出选择
          递归执行backtrack
          撤销选择
  ```




```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        # 回溯算法，复杂度较高，因为回溯算法就是暴力穷举，遍历整颗决策树是不可避免的
        res = []
        def backtrack(path=[], selects=nums):
            if not selects:
                res.append(path[:])
                return
            for i in range(len(selects)):
                path.append(selects[i])
                backtrack(path, selects[:i] + selects[i+1:])
                path.pop()
        backtrack()
        return res
```

2、库函数`itertools.permutations`

```python
#直出结果
def permute(self, nums: List[int]) -> List[List[int]]:
        return list(itertools.permutations(nums))
```

3、同31题（求解输入数组的下一个排列），假设输入第一个数为最小的数，不断调用31题程序

4、见缝插针，同22题，分别在每个数的左右插入一个数

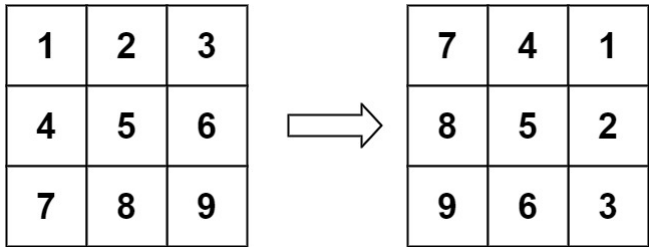
```python
class Solution:
    def permute(self, nums):
        res = [[nums[0]]]
        tem = []
        n = len(nums)
        for i in range(1,n):
            for k in res:
                for j in range(len(k)+1):
                    k.insert(j,nums[i])#在索引为j的位置插入元素
                    tem.append(k[:])
                    k.pop(j)#删除索引为j的元素
            res=tem
            tem=[]
```

```
        return res
```

48、旋转图像

给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

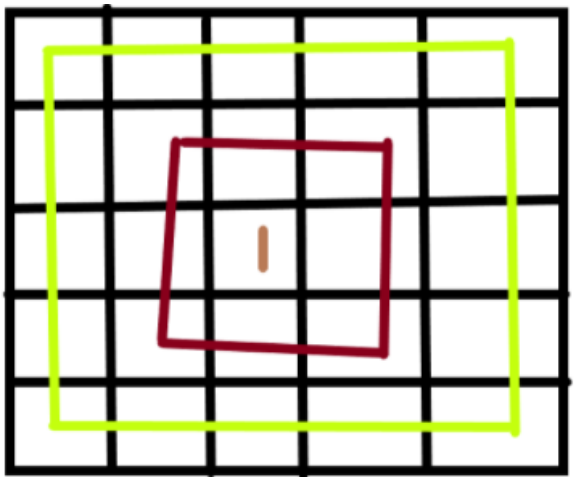


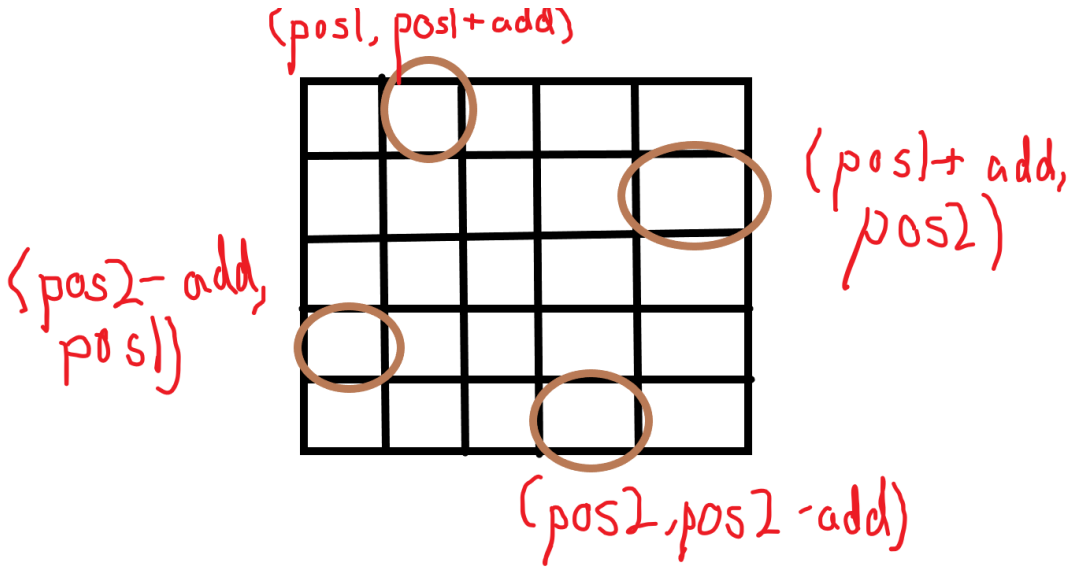
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[[7,4,1],[8,5,2],[9,6,3]]

输入：matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
输出：[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

- **高分解答**

  1、逐层平移和偏移量的增加






```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        pos1,pos2 = 0,len(matrix)-1
        while   pos1<pos2:
            add = 0
            while   add < pos2-pos1:
                #左上角为0块，右上角为1块，右下角为2块，左下角为3块
                temp = matrix[pos2-add][pos1]
                matrix[pos2-add][pos1] = matrix[pos2][pos2-add]
                #3 = 2
                matrix[pos2][pos2-add] = matrix[pos1+add][pos2]
                #2 = 1
                matrix[pos1+add][pos2] = matrix[pos1][pos1+add]
                #1 = 0
                matrix[pos1][pos1+add] = temp
                #0 = temp
                add = add+1
            pos1 = pos1+1
```

2、对角线翻转+水平翻转

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)
        # 对角线翻转
        for i in range(n - 1):
            for j in range(i + 1, n):
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
        # 水平翻转
        for i in range(n):
            for j in range(n >> 1):
                matrix[i][j], matrix[i][n - j - 1] = matrix[i][n - j - 1],
matrix[i][j]
```

49、字母异位词分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

> 输入: ["eat", "tea", "tan", "ate", "nat", "bat"]
> 输出:
> [
> ["ate","eat","tea"],
> ["nat","tan"],
> ["bat"]]

- **高分解答**

1、主要是`dict.get()`函数的理解：`dict.get(key, default=None)`

1. key -- 字典中要查找的键。
2. default -- 如果指定键的值不存在时，返回该默认值。

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        dict = {}
        for item in strs:
            key = tuple(sorted(item))
                            #因为字典的键，必须是不可变类型，所以用tuple。
            dict[key] = dict.get(key, []) + [item]
        return list(dict.values())
```

2、质数代替

用一个字符串内不同字符代表的不同质数，相乘在一起，生成的数字一定是一个合数，这个合数质因数分解，一定是有某些字符生成的，顺序可以变化，因为是乘法，所以一个字符串，只要是固定的几个字符，固定的数量，那么这个合数一定是唯一的 从而确定唯一键

```
class Solution:
    prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103]
    def ks(s):
        #求字符串的质数乘积
        c = 1
        for cc in s: c *= Solution.prime[ord(cc)-97]
        return c

    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        res = {}
        for s in strs:
            k = Solution.ks(s)
            if k not in res: res[k] = [s]
            else: res[k].append(s)
        return list(res.values())
```

3、排序+哈希

`setdefault()`：如果键不存在于字典中，将会添加键并将值设为默认值。

```python
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = {}
        for s in strs:
            d.setdefault("".join(sorted(s)), []).append(s)
        return list(d.values())
```

55、跳跃游戏

给定一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

> 输入：nums = [2,3,1,1,4]
> 输出：true
> 解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。
>
> 输入：nums = [3,2,1,0,4]
> 输出：false
> 解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。

- **高分解答**

  1、贪心原则：尽可能到达最远位置

  如果能到达某个位置，那一定能到达它前面的所有位置。

  方法：初始化最远位置为 0，然后遍历数组，如果当前位置能到达，并且当前位置+跳数>最远位置，就更新最远位置。最后比较最远位置和数组长度。

```python
class Solution:
    def canJump(self, nums) :
        max_i = 0        #初始化当前能到达最远的位置
        for i, jump in enumerate(nums):    #i为当前位置，jump是当前位置的跳数
            if max_i>=i and i+jump>max_i:  #如果当前位置能到达，并且当前位置+跳数>最远位置
                max_i = i+jump  #更新最远能到达位置
        return max_i>=i
```

  2、倒序：通过倒推回起点

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        index = len(nums)-1
        for i in range(len(nums)-2, -1, -1):
            if i + nums[i] >= index:
                index = i
        return index == 0
```

56、合并区间

以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi] 。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间

> 输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
> 输出：[[1,6],[8,10],[15,18]]
> 解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].

> 输入：intervals = [[1,4],[4,5]]
> 输出：[[1,5]]
> 解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。

- **高分解答**

  1、排序

  - 先按首位置进行排序;
  - 接下来,如何判断两个区间是否重叠呢?比如 `a = [1,4],b = [2,3]`，当 `a[1] >= b[0]` 说明两个区间有重叠.
  - 左边位置一定是确定，就是 `a[0]`，而右边位置是 `max(a[1], b[1])` 所以,我们就能找出整个区间为: `[1,4]`

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        if not intervals: return []
        intervals.sort()
        res = [intervals[0]]
        for x, y in intervals[1:]:
            if res[-1][1] < x:
                res.append([x, y])
            else:
                res[-1][1] = max(y, res[-1][1])
        return res
```

62、不同路径

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。机器人达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?


```
输入：m = 3, n = 7
输出：28
```

```
输入：m = 3, n = 2
输出：3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下
```

- **第一想法**

  找规律：

1	**1**	**1**	**1**	**1**	**1**	**1**
1	2	3	4	5	6	7
1	3	6	10	15	21	28

  使用动态规划

  1. 定义状态：即定义数据元素的含义，这里定义`dp[i][j]`为当前位置的路径数，i表示i列，j表示j行
  2. 到达`dp[i][j]`就可能是经过`dp[i-1][j]`到达，也可能是经过`dp[i][j-1]`到达。所以状态转移方程为：`dp[i][j]=dp[i-1][j]+dp[i][j-1]`
  3. 设置最初的路径条数即`dp[i][0]=1,dp[0][j]=1`，即第一行，第一列值为1。

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        res=[[1] * n for _ in range(m)]
        for x in range(1,m):
            for y in range(1,n):
                res[x][y]=res[x-1][y]+res[x][y-1]

        return res[-1][-1]
```

- **高分解答**

  1、动态规划+空间压缩

  - 对二维矩阵进行压缩成一位数组,将最新生成的值覆盖掉旧的值,逐行求解当前位置的最新路径条数!

- 状态转移公式变成：f[i] = f[i-1]+f[i]
- f[-1]表示可能路径的总数

```
class Solution(object):
    def uniquePaths(self, m, n):
        f = [1]*m
        for j in range(1,n):
            for i in range(1,m):
                f[i] = f[i-1]+f[i]
        return f[-1]
```

2、组合数（杨辉三角形）

- *n*行m列说明最终路径里面一定会有n-1个向下的步骤和m-1个向右的步骤
- 意思就是在m+n-2个步骤的路径上，选n-1个位置来向下走，剩下的位置向右走即可
- 计算 组合数 C(m+n-2, m-1) 或者 C(m+n-2, n-1)

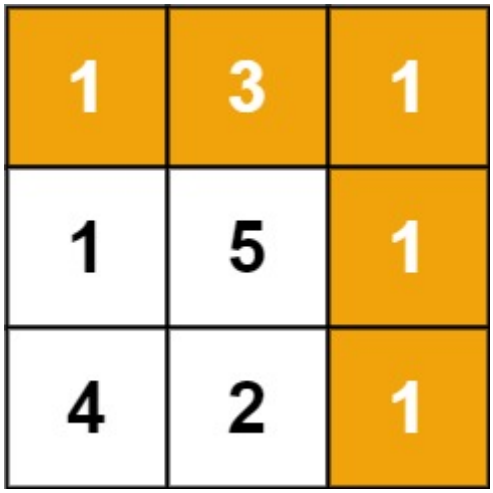
```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        return  comb(m + n - 2, n - 1)
```

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        a = m + n - 2
        b = (m if m > n else n) - 1
        ans = 1
        for i in range(b+1, a+1):
            ans *= i
        for i in range(2, a-b+1):
            ans //= i
        return ans

    #  A(m+n-2)(max(m,n))/A(min(a,b)-1)(2)
```

64、最小路径和

给定一个包含非负整数的 `m x n` 网格 `grid` ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。




```
输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
输出：7
解释：因为路径 1→3→1→1→1 的总和最小。
```

- **第一想法**

  1、动态规划

  - 构建一个res=m×n的动态规划矩阵，保存最短路径和，`res[-1][-1]`
  - [x,y]位置的最短路径和，应该和`res[x-1][y]`，`res[x][y-1]`相比较，
  - 小的加上`res[x][y]`即为当前路径最小和

1	**4**	**5**
2	7	6
6	8	7

```python
import numpy as np
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        (m,n)=np.shape(grid)
        res=[[grid[0][0]]*n for _ in range(m)]
        for i in range(1,n):
            res[0][i]=res[0][i-1]+grid[0][i]
        for i in range(1,m):
            res[i][0]=res[i-1][0]+grid[i][0]

        for x in range(1,m):
            for y in range(1,n):
                res[x][y]=min(res[x-1][y],res[x][y-1])+grid[x][y]

        return res[-1][-1]
```

- **高分解答**

  1、动态规划，对上面程序的优化

```
class Solution:
    def minPathSum(self, grid: [[int]]) -> int:
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if i == j == 0: continue
                elif i == 0:  grid[i][j] = grid[i][j - 1] + grid[i][j]
                elif j == 0:  grid[i][j] = grid[i - 1][j] + grid[i][j]
                else: grid[i][j] = min(grid[i - 1][j], grid[i][j - 1]) +
grid[i][j]
        return grid[-1][-1]
```

空间压缩

```
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        if not grid or not grid[0]:
            return 0
        dp = [float('inf')] * len(grid[0])
        dp[0] = 0
        #dp[1]是真正的起始点，前面没值，用+1替代-1，避免掉 -1超界的情况
        for row in grid:
            for index, num in enumerate(row):
                if index == 0:
                    dp[index] = dp[index] + num
                else:
                    dp[index] = min(dp[index], dp[index - 1]) + num

        return dp[-1]
```

75、颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

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

> 输入：nums = [2,0,1]
> 输出：[0,1,2]

- **第一想法**

  1、排序，一句代码

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        nums.sort()
```

2、计算数组有多少0，1，2，然后补进去

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        m = len(nums)
        num_sort = [0 for _ in range(3)]
        for num in nums:
            num_sort[num] += 1
        for index, count in enumerate(num_sort):
            n = count
            while n>0:
                nums.append(index)
                n -= 1
        del nums[0:m]
```

- **高分解答**

  1、快速排序partition

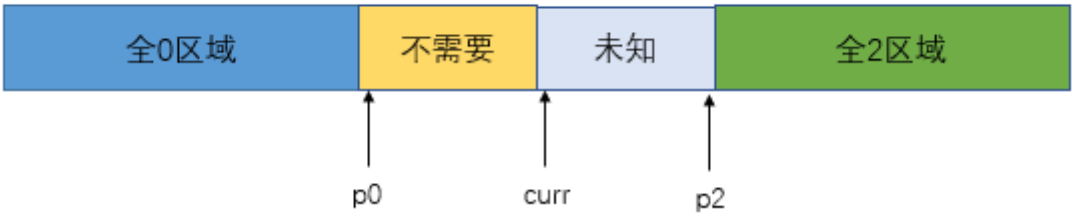
  - 选择一个标定元素（称为 pivot ，一般而言随机选择），然后通过一次扫描，把数组分成三个部分：
    - 第 1 部分严格小于 pivot 元素的值；
    - 第 2 部分恰好等于 pivot 元素的值；
    - 第 3 部分严格大于 pivot 元素的值。
  - 第 2 部分元素就是排好序以后它们应该在的位置，接下来只需要递归处理第 1 部分和第 3 部分的元素。
  - 经过一次扫描把整个数组分成 33 个部分，正好符合当前问题的场景。写对这道题的方法是：把循环不变量的定义作为注释写出来，然后再编码。

  - 设计循环不变量的原则是 **不重不漏**。
    - len 是数组的长度；
    - 变量 zero 是前两个子区间的分界点，一个是闭区间，另一个就必须是开区间；
    - 变量 i 是循环变量，一般设置为开区间，表示 i 之前的元素是遍历过的；
    - two 是另一个分界线，我设计成闭区间。
  - 如果循环不变量定义如下：
    - 所有在子区间 [0, zero) 的元素都等于 0；
    - 所有在子区间 [zero, i) 的元素都等于 1；
    - 所有在子区间 [two, len - 1] 的元素都等于 2

```python
from typing import List
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        def swap(nums, index1, index2):
            nums[index1], nums[index2] = nums[index2], nums[index1]

        size = len(nums)
```

```
        if size < 2:
            return

        zero = 0
        two = size
        i = 0

        while i < two:
            if nums[i] == 0:
                swap(nums, i, zero)
                i += 1
                zero += 1
            elif nums[i] == 1:
                i += 1
            else:
                two -= 1
                swap(nums, i, two)
```

2、双指针算法




```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        p0, curr, p2 = 0, 0, len(nums) - 1
        while curr <= p2:
            if nums[curr] == 0: # 为0，做出决策
                nums[p0], nums[curr] = nums[curr], nums[p0]
                p0 += 1
                curr += 1
            elif nums[curr] == 2:   # 为2，做出决策
                nums[p2], nums[curr] = nums[curr], nums[p2]
                p2 -= 1
            else:                   # 为1，做出决策
                curr += 1
```

78、子集

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。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。你可以按 **任意顺序** 返回解集。

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

- **高分解答**

  1、库函数

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []
        for i in range(len(nums)+1):
            for tmp in itertools.combinations(nums, i):
                res.append(tmp)
        return res
```

  2、迭代

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = [[]]
        for i in nums:
            res = res + [[i] + num for num in res]
        return res
```

  3、递归（回溯）算法

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []
        n = len(nums)

        def helper(i, tmp):
            res.append(tmp)
            for j in range(i, n):
                helper(j + 1,tmp + [nums[j]] )
        helper(0, [])
        return res
```

  4、动态规划

  - 首先定义初始状态和状态转移方程： dp[i], 其中i代表nums数组里数字的个数
    - dp[0] = [[]]，数组里没有数时，只有一种子集，为空
    - dp[1] = [[], [n]]，当数组里只有一个数n的时候，在dp[0]的基础上，加了一种单个的。
    - dp[i] = dp[i - 1]的所有组合 + dp[i - 1]里所有组合都加上第i个数。
    - 也就是说，每次数组新加一个数，首先要把之前的所有组合放进结果，然后把之前所有组合里，都加一个新的数，放进结果里。

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        N = len(nums)
        dp = [[]]
        cur_dp = []
        for i in range(N):
            cur_dp = []
            cur_dp.extend(dp)
            for comb in dp:
                cur_dp.append(comb + [nums[i]])
            dp = cur_dp
        return cur_dp
```

79、单词搜索

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

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

> board =
> [
> ['A','B','C','E'],
> ['S','F','C','S'],
> ['A','D','E','E']
>]
>
> 给定 word = "ABCCED", 返回 true
> 给定 word = "SEE", 返回 true
> 给定 word = "ABCB", 返回 false

- **高分解答**

  1、DFS、回溯

  - 其中判断单词是否存在于二维数组中的依据是：
    - 单词存在于二维数组必须是按照字母顺序，且构建单词字母必须是相邻单元格（相邻单元格表示当前单元格四个方位相邻的其他单元格）
    - 同个单元格的字母不能重复使用。
  - 那么，我们可以发现要去判断单词是否存在于二维数组中，首先我们需要先找到单词的起始字母，然后再向四周扩散搜索。具体的思路如下：
    - 默认从坐标 (0, 0) 开始搜索，先找到单词首字母，然后对其先进行标记（防止同个单元格被重复使用）；
    - 然后向四个方位进行扩散搜索（注意限定边界，以及注意是否已被标记），对单元格的字母与单词中的字母继续比对：
      - 若匹配，则进行标记，继续扩散搜索；
      - 若不匹配，则尝试其他方位；
      - 若完全不匹配（四个方位都不匹配），则进行回退，同时释放当前单元格的标记。
    - 重复上面的步骤：
      - 当单词完全匹配，可直接返回 True；

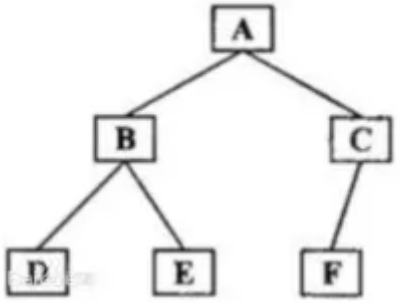
- 若所有单元格均搜索无果，则返回 False。

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        m = len(board)
        n = len(board[0])
        visited = [[False] * n for _ in range(m)]
        rows = [-1, 0, 1, 0]
        cols = [0, 1, 0, -1]

        def dfs(x, y, idx):
            """搜索单词
            Args:
                x: 行索引
                y: 列索引
                idx: 单词对应的字母索引
            """
            if board[x][y] != word[idx]:
                return False

            if idx == len(word) - 1:
                return True

            # 先标记
            visited[x][y] = True

            # 找到符合的字母时开始向四个方向扩散搜索
            for i in range(4):
                nx = x + rows[i]
                ny = y + cols[i]
                if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny] and
dfs(nx, ny, idx+1):
                    return True
            # 扩散未搜索对应的字母，释放标记
            # 继续往其他方位搜索
            visited[x][y] = False
            return False

        for x in range(m):
            for y in range(n):
                if dfs(x, y, 0):
                    return True
        return False
```

94、二叉树的中序遍历

给定一个二叉树的根节点 `root` ，返回它的 **中序** 遍历。

中序遍历首先遍历左子树，然后访问根结点，最后遍历右子树。若二叉树为空则结束返回



中序遍历结果：DBEAFC

- **高分解答**

  1、颜色标记法

  - 使用颜色标记节点的状态，新节点为白色，已访问的节点为灰色。
  - 如果遇到的节点为白色，则将其标记为灰色，然后将其右子节点、自身、左子节点依次入栈。
  - 如果遇到的节点为灰色，则将节点的值输出。

  前序、中序、后序遍历代码一样

  ```
      中
  左    右
  ```

  前序遍历：中，左，右
  中序遍历：左，中，右
  后序遍历：左，右，中

  对应本题中序遍历：

  出栈顺序为：`左，中，右`

  入栈顺序为：`右，中，左`

```python
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((WHITE, node.right))
                stack.append((GRAY, node))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res
```

  2、递归

```python
# 递归1：二叉树遍历最易理解和实现版本
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        # 前序递归
```

```
        return [root.val] + self.preorderTraversal(root.left) +
self.preorderTraversal(root.right)
        # # 中序递归
        # return self.inorderTraversal(root.left) + [root.val] +
self.inorderTraversal(root.right)
        # # 后序递归
        # return self.postorderTraversal(root.left) +
self.postorderTraversal(root.right) + [root.val]

# 递归2：通用模板，可以适应不同的题目，添加参数、增加返回条件、修改进入递归条件、自定义返回
值
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        def dfs(cur):
            if not cur:
                return
            # 前序递归
            res.append(cur.val)
            dfs(cur.left)
            dfs(cur.right)
            # # 中序递归
            # dfs(cur.left)
            # res.append(cur.val)
            # dfs(cur.right)
            # # 后序递归
            # dfs(cur.left)
            # dfs(cur.right)
            # res.append(cur.val)
        res = []
        dfs(root)
        return res
```

3、迭代

```
# 迭代1：前序遍历最常用模板（后序同样可以用）
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        res = []
        stack = [root]
        # # 前序迭代模板：最常用的二叉树DFS迭代遍历模板
        while stack:
            cur = stack.pop()
            res.append(cur.val)
            if cur.right:
                stack.append(cur.right)
            if cur.left:
                stack.append(cur.left)
        return res

        # # 后序迭代，相同模板：将前序迭代进栈顺序稍作修改，最后得到的结果反转
        # while stack:
        #     cur = stack.pop()
        #     if cur.left:
        #         stack.append(cur.left)
        #     if cur.right:
```

```
        #         stack.append(cur.right)
        #     res.append(cur.val)
        # return res[::-1]

# 迭代1：层序遍历最常用模板
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        cur, res = [root], []
        while cur:
            lay, layval = [], []
            for node in cur:
                layval.append(node.val)
                if node.left: lay.append(node.left)
                if node.right: lay.append(node.right)
            cur = lay
            res.append(layval)
        return res

# 迭代2：前、中、后序遍历通用模板（只需一个栈的空间）
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        res = []
        stack = []
        cur = root
        # 中序，模板：先用指针找到每颗子树的最左下角，然后进行进出栈操作
        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            res.append(cur.val)
            cur = cur.right
        return res

        # # 前序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.left
        #     cur = stack.pop()
        #     cur = cur.right
        # return res

        # # 后序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.right
        #     cur = stack.pop()
        #     cur = cur.left
        # return res[::-1]

# 迭代3：标记法迭代（需要双倍的空间来存储访问状态）：
# 前、中、后、层序通用模板，只需改变进栈顺序或即可实现前后中序遍历，
```

```python
# 而层序遍历则使用队列先进先出。0表示当前未访问，1表示已访问。
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        res = []
        stack = [(0, root)]
        while stack:
            flag, cur = stack.pop()
            if not cur: continue
            if flag == 0:
                # 前序，标记法
                stack.append((0, cur.right))
                stack.append((0, cur.left))
                stack.append((1, cur))

                # # 后序，标记法
                # stack.append((1, cur))
                # stack.append((0, cur.right))
                # stack.append((0, cur.left))

                # # 中序，标记法
                # stack.append((0, cur.right))
                # stack.append((1, cur))
                # stack.append((0, cur.left))
            else:
                res.append(cur.val)
        return res

        # # 层序，标记法
        # res = []
        # queue = [(0, root)]
        # while queue:
        #     flag, cur = queue.pop(0)  # 注意是队列，先进先出
        #     if not cur: continue
        #     if flag == 0:
                # 层序遍历这三个的顺序无所谓，因为是队列，只弹出队首元素
        #         queue.append((1, cur))
        #         queue.append((0, cur.left))
        #         queue.append((0, cur.right))
        #     else:
        #         res.append(cur.val)
        # return res
```

4、莫里斯遍历

5、n叉树遍历

```python
# N叉树简洁递归
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root: return []
        res = [root.val]
        for node in root.children:
            res.extend(self.preorder(node))
        return res

# N叉树通用递归模板
class Solution:
```

```python
    def preorder(self, root: 'Node') -> List[int]:
        res = []
        def helper(root):
            if not root:
                return
            res.append(root.val)
            for child in root.children:
                helper(child)
        helper(root)
        return res

# N叉树迭代方法
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root:
            return []
        s = [root]
        # s.append(root)
        res = []
        while s:
            node = s.pop()
            res.append(node.val)
            # for child in node.children[::-1]:
            #     s.append(child)
            s.extend(node.children[::-1])
        return res
```

96、不同的二叉搜索树

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种?

若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树

> 输入: 3
> 输出: 5
> 解释:
> 给定 n = 3, 一共有 5 种不同结构的二叉搜索树:
>
> ```
> 1     3   3    2    1
> \    /   /    /\    \
>  3   2   1    1  3    2
> /   /    \         \
> 2   1     2         3
> ```

- **高分解答**

  1、找规律、公式

  从 1,2,...n 数列构建搜索树，实际上只是一个不断细分的过程
  例如，我要用 [1,2,3,4,5,6] 构建
  首先，提起 "2" 作为树根，[1]为左子树，[3,4,5,6] 为右子树

现在就变成了一个更小的问题：如何用 [3,4,5,6] 构建搜索树?
比如，我们可以提起 "5" 作为树根，[3,4] 是左子树，[6] 是右子树

- 我们可以很容易得知几个简单情况 `f(0) = 1`, `f(1) = 1`, `f(2) = 2`
- 我们来看 `[1,2,3]`
  - 如果提起 1 作为树根，左边有f(0)种情况，右边 f(2) 种情况，左右搭配一共有 f(0)*f(2) 种情况
  - 如果提起 2 作为树根，左边有f(1)种情况，右边 f(1) 种情况，左右搭配一共有 f(1)*f(1) 种情况
  - 如果提起 3 作为树根，左边有f(2)种情况，右边 f(0) 种情况，左右搭配一共有 f(2)*f(0) 种情况
  - `f(4) = f(0)*f(3) + f(1)*f(2) + f(2)*f(1) + f4(3)*f(0)`

```
class Solution:
    def numTrees(self, n: int) -> int:
        store = [1,1] #f(0),f(1)
        if n <= 1:
            return store[n]
        for m in range(2,n+1):
            s = m-1
            count = 0
            for i in range(m):
                count += store[i]*store[s-i]
            store.append(count)
        return store[n]
```

2、递归

```
from functools import lru_cache
class Solution:
    @lru_cache()        #用到了装饰器functools.lru_cache做缓存处理
    def numTrees(self, n: int) -> int:
        if n <= 0: return 1
        if n <= 2: return n

        return sum([self.numTrees(i-1) * self.numTrees(n-i) for i in
range(1, n + 1)])
```

98、验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

```
输入:
  5
 / \
1   4
   / \
  3   6
```

输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
根节点的值为 5 ， 但是其右子节点值为 4 。

- **高分解答**

  1、因为二叉搜索树中序遍历是递增的,所以我们可以中序遍历判断前一数是否小于后一个数.

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        res = []
        def helper(root):
            if not root:
                return
            helper(root.left)
            res.append(root.val)
            helper(root.right)
        helper(root)
        return res == sorted(res) and len(set(res)) == len(res)      #判断是否符合中序遍历，以及没有重复值
```

  2、迭代，中序遍历迭代方式

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        stack = []
        p = root
        pre = None
        while p or stack:
            while p:
                stack.append(p)
                p = p.left
            p = stack.pop()
            if pre and p.val <= pre.val:
                return False
            pre = p
            p = p.right
        return True
```

  3、递归中序遍历

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        self.pre = None
        def isBST(root):
            if not root:
                return True
```

```
            if not isBST(root.left):
                return False
            if self.pre and self.pre.val >= root.val:
                return False
            self.pre = root
            #print(root.val)
            return  isBST(root.right)
        return isBST(root)
```

4、利用最大值最小值

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        def isBST(root, min_val, max_val):
            if root == None:
                return True
            # print(root.val)
            if root.val >= max_val or root.val <= min_val:
                return False
            return isBST(root.left, min_val, root.val) and isBST(root.right,
root.val, max_val)
        return isBST(root, float("-inf"), float("inf"))
```

102、二叉树的层序遍历

给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。

> 二叉树：[3,9,20,null,null,15,7],
>
> ```
>     3
> / \
> 9  20
>  /  \
> 15   7
> ```
>
> 返回其层序遍历结果：
>
> [[3],
> [9,20],
> [15,7]]

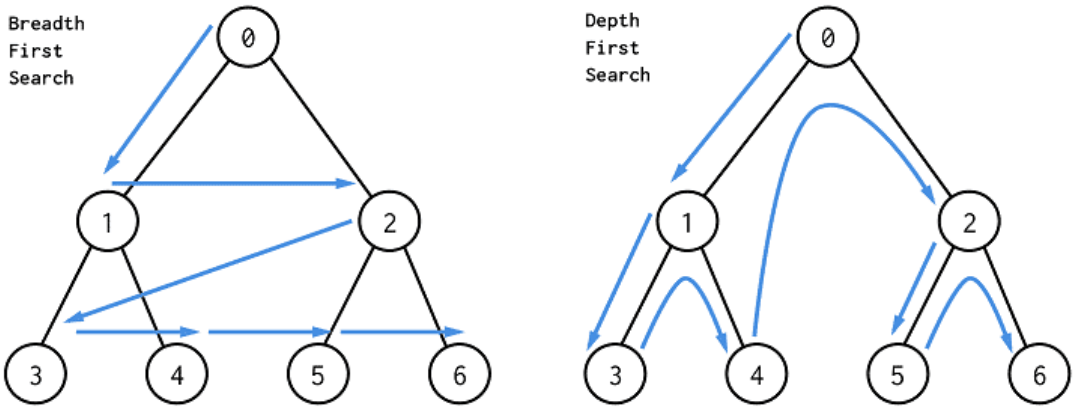
- **第一想法**

  同94题中解答

```
# 迭代1：层序遍历最常用模板
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        cur, res = [root], []
        while cur:
```

```
            lay, layval = [], []
            for node in cur:
                layval.append(node.val)
                if node.left: lay.append(node.left)
                if node.right: lay.append(node.right)
            cur = lay
            res.append(layval)
        return res
```

- **高分解答**



1、BFS：使用队列，把每个还没有搜索到的点依次放入队列，然后再弹出队列的头部元素当做当前遍历点。BFS总共有两个模板：

- 如果不需要确定当前遍历到了哪一层，BFS模板如下。

```
while queue 不空：
    cur = queue.pop()
    for 节点 in cur的所有相邻节点：
        if 该节点有效且未访问过：
            queue.push(该节点)
```

- 如果要确定当前遍历到了哪一层，BFS模板如下。
  这里增加了level表示当前遍历到二叉树中的哪一层了，也可以理解为在一个图中，现在已经走了多少步了。size表示在当前遍历层有多少个元素，也就是队列中的元素数，我们把这些元素一次性遍历完，即把当前层的所有元素都向外走了一步。

```
level = 0
while queue 不空：
    size = queue.size()
    while (size --) {
        cur = queue.pop()
        for 节点 in cur的所有相邻节点：
            if 该节点有效且未被访问过：
                queue.push(该节点)
    }
    level ++;
```

使用队列保存每层的所有节点，每次把队列里的原先所有节点进行出队列操作，再把每个元素的非空左右子节点进入队列。因此即可得到每层的遍历。

```
class Solution(object):
    def levelOrder(self, root):
```

```
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        queue = collections.deque()
        queue.append(root)
        res = []
        while queue:
            size = len(queue)
            level = []
            for _ in range(size):
                cur = queue.popleft()
                if not cur:
                    continue
                level.append(cur.val)
                queue.append(cur.left)
                queue.append(cur.right)
            if level:
                res.append(level)
        return res
```

2、DFS

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

```
class Solution(object):
    def levelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        res = []
        self.level(root, 0, res)
        return res

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

105、从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

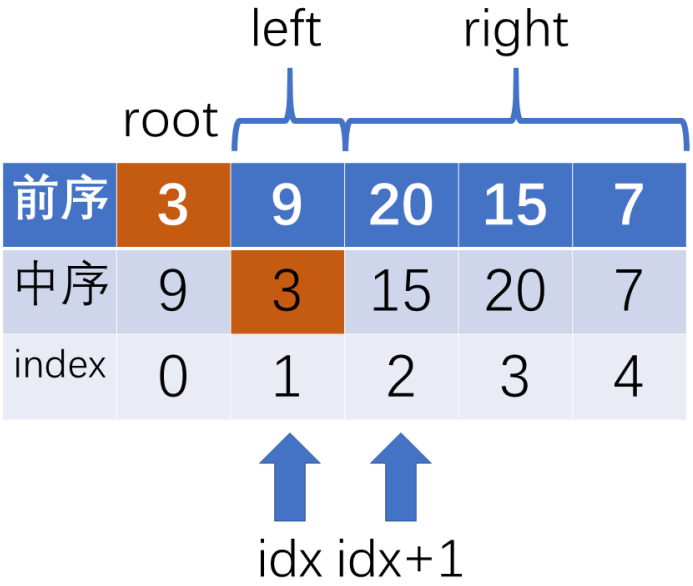
你可以假设树中没有重复的元素。

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

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

- **高分解答**

  1、先序找根，划分左右，再递归构造左右子树

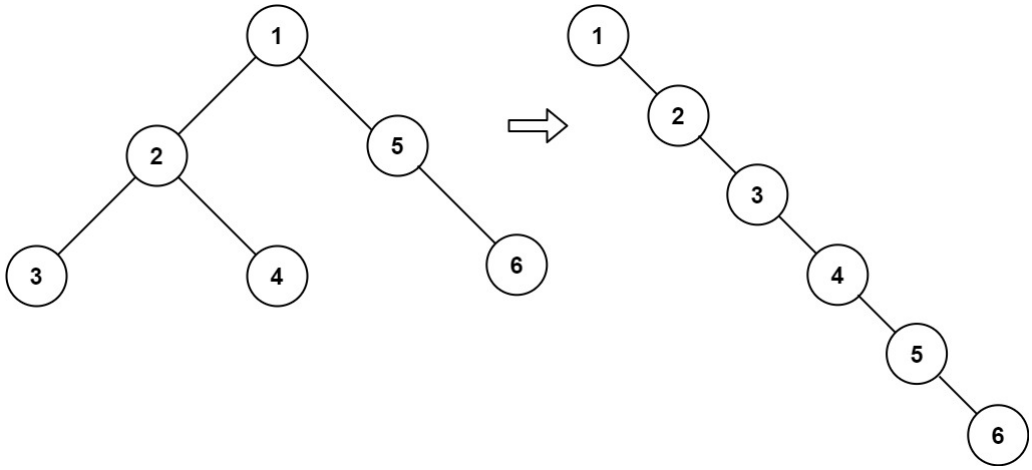
  - 通过先序遍历我们可以找到root，根据root我们可以再中序找到当前root对应的左右子树，再递归对当前root的左右子树进行构造
  - 知道inorder中，当前root的左侧的所有点就是其左子树，root的右侧的所有点就是当前root的右子树，就把这左右两堆数字想成当前root的左右2个节点就好，然后扔到函数里进行下一层的递归。
  - `inorder.index(preorder[0])` 这一步获取根的索引值，题目说树中的各个节点的值都不相同，也确保了这步得到的结果是唯一准确的。而且这个idx还能当长度用相当于 左+根 的长度，因为 左+根 和 根+左 是等长的。
  - 


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

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

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

114、二叉树展开为链表

给你二叉树的根结点 root ， 请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 TreeNode ， 其中 right 子指针指向链表中下一个结点，而左子指针始终为 null 。
- 展开后的单链表应该与二叉树 先序遍历 顺序相同。




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

- **第一想法**

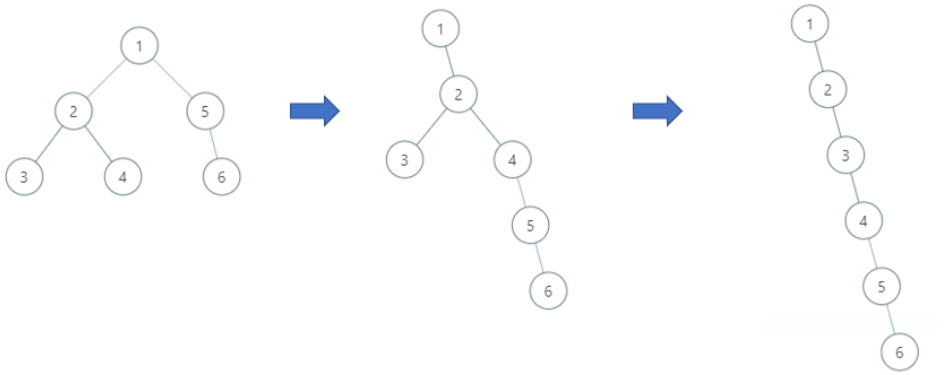
  1、前序遍历+列表循环

  因为变成右子树的是前序遍历后的列表，可以先前序遍历得到树的列表，然后依次给右子树

```python
class Solution(object):
    def flatten(self, root):
        if not root:
            return
        queue = []
        # 前序遍历整棵二叉树，并将遍历的结果放到数组中
        def dfs(root):
            if not root:
                return
            queue.append(root)
            dfs(root.left)
            dfs(root.right)
        dfs(root)
        head = queue.pop(0)
        head.left = None
        # 遍历链表，将链表中的TreeNode节点前后串联起来
        while queue:
            tmp = queue.pop(0)
            tmp.left = None
            head.right = tmp
            head = tmp
```

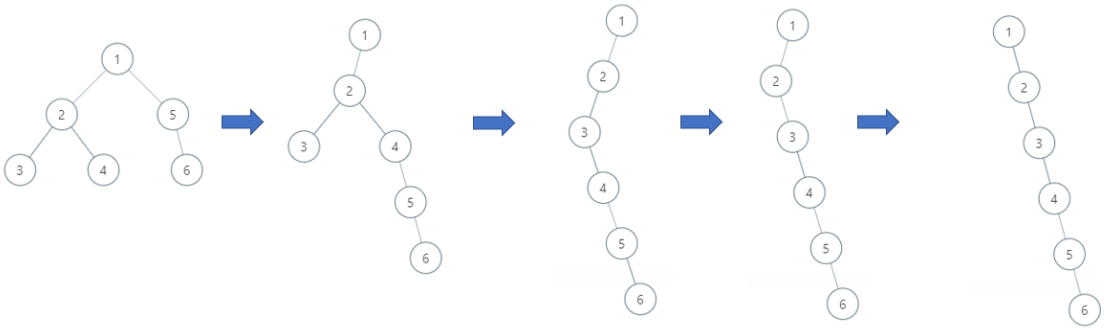
- **高分解答**

  1、迭代：原地操作



```
class Solution:
    def flatten(self, root):
        while root:
            if root.left:    #左子树存在的话才进行操作
                sub_left = root.left
                while sub_left.right:    #左子树的右子树找到最深
                    sub_left = sub_left.right
                sub_left.right = root.right #将root的右子树挂到左子树的右子树的最
深
                root.right = root.left       #将root的左子树挂到右子树
                root.left = None             #将root左子树清空
            root = root.right                #继续下一个节点的操作
```

2、递归，后序遍历



```
class Solution(object):
    def flatten(self, root):
        def dfs(root):
            if not root:
                return
            dfs(root.left)
            dfs(root.right)
            # 将右子树挂到 左子树的最右边
            # 再将整个左子树挂到根节点的右边
            if root.left:
                pre = root.left
                while pre.right:
                    pre = pre.right
                pre.right = root.right
                root.right = root.left
                root.left = None
        dfs(root)
```

3、前序遍历是:打印根节点-左节点-右节点 这样的顺序，如果是:右节点-左节点-打印根节点 这样完全相反的顺序遍历

前序遍历完是1,2,3,4,5,6，按照这种新的方式遍历其结果是:6,5,4,3,2,1。动画演示

既然得到了反向的顺序，那么就可以把前后节点串联起来，当遍历到根节点1的时候，整个串联就完成了，二叉树就变成了链表。

```
class Solution(object):
    def flatten(self, root):
        self.pre = None
        def dfs(root):
            if not root:
                return None
            # 右节点-左节点-根节点  这种顺序正好跟前序遍历相反
            # 用pre节点作为媒介，将遍历到的节点前后串联起来
            dfs(root.right)
            dfs(root.left)
            root.left = None
            root.right = self.pre
            self.pre = root
```