139、单词拆分

给定一个非空字符串 *s* 和一个包含非空单词的列表 `wordDict`，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

> 输入: s = "leetcode", wordDict = ["leet", "code"]
> 输出: true
> 解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

> 输入: s = "applepenapple", wordDict = ["apple", "pen"]
> 输出: true
> 解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
> 注意你可以重复使用字典中的单词。

> 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
> 输出: false

- **第一想法**

  1、构建一个空词组st，遍历s，查询在字典中是否存在，存在即重新构建st

  在特殊例子中会出现错误，未考虑多种的表示方式

  例： s = "catsand", wordDict = ["cats", "and", "cat"]

  输出：False　　正确输出：True

```python
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        st=''
        for i in s:
            st+=i
            if st in wordDict:
                st=''
        if len(st)>0:
            return False
        return True
```

- **高分解答**

  1、动态规划

""	l	e	e	t	c	o	d	e
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE

  1. 初始化 dp=[False，，,False]，长度为 n+1。n为字符串长度。dp[i] 表示 s 的前 i位是否可以用 wordDict 中的单词表示。

2. 初始化 dp[0]=True，空字符可以被表示。
3. 遍历字符串的所有子串，遍历开始索引 i，遍历区间 [0,n)：
    - 遍历结束索引 j，遍历区间 [i+1,n+1)：
        - 若 dp[i]=True 且 s[i,⋯,j) 在 wordlist 中：dp[j]=True。解释：dp[i]=True说明 s 的前 i位可以用 wordDict 表示，则 s[i,⋯,j) 出现在 wordDict 中，说明 s 的前 j 位可以表示。
4. 返回 dp[n]

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        n=len(s)
        dp=[False]*(n+1)
        dp[0]=True
        for i in range(n):
            for j in range(i+1,n+1):
                if(dp[i] and (s[i:j] in wordDict)):
                    dp[j]=True
        return dp[-1]
```

2、记忆化回溯

1. 使用记忆化函数，保存出现过的 backtrack(s)，避免重复计算。
2. 定义回溯函数 backtrack(s)
    - 若 s长度为 0，则返回 True，表示已经使用 wordDict 中的单词分割完。
    - 初试化当前字符串是否可以被分割 res=False
    - 遍历结束索引 i，遍历区间 [1,n+1)：
        - 若 s[0,⋯,i−1] 在 wordDict 中：res=backtrack(s[i,⋯,n−1]) or res。解释：保存遍历结束索引中，可以使字符串切割完成的情况。
    - 返回 resres
3. 返回 backtrack(s)

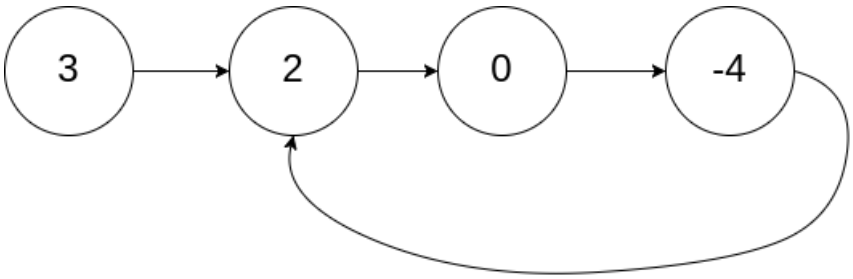
```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        import functools
        @functools.lru_cache(None)
        def back_track(s):
            if(not s):
                return True
            res=False
            for i in range(1,len(s)+1):
                if(s[:i] in wordDict):
                    res=back_track(s[i:]) or res
            return res
        return back_track(s)
```

142、环形链表II

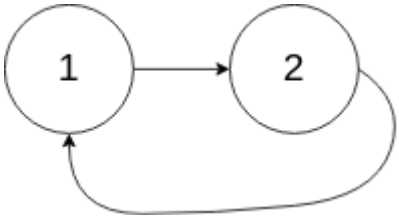
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

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 仅仅是用于标识环的情况，并不会作为参数传递到函数中。

说明：不允许修改给定的链表。




```
输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。
```




```
输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。
```


```
输入：head = [1], pos = -1
输出：返回 null
解释：链表中没有环。
```

- **第一想法**

  1、使用数组，循环遍历列表，

  - 有环就会存在数组中
  - 无环就输出null

```python
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        res=[]
        while head:
            if head in res:
                return head
            res.append(head)
            head=head.next
        return head
```

- **高分解答**

1、对上面数组使用哈希字典，

```python
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        m = {}
        while head:
            if m.get(head):
                return head      #如果改点之前被访问过，直接返回该点位置
            m[head] = 1
            head = head.next
        return head
```

2、双指针法

1. 双指针第一次相遇： 设两指针 fast，slow 指向链表头部 head，fast 每轮走 2 步，slow 每轮走 1 步；
    1. 第一种结果： fast 指针走过链表末端，说明链表无环，直接返回 null；
        - TIPS: 若有环，两指针一定会相遇。因为每走 11 轮，fast 与 slow 的间距 +1+1，fast 终会追上 slow；
    2. 第二种结果： 当fast == slow时， 两指针在环中 第一次相遇 。下面分析此时fast 与 slow走过的 步数关系 ：
        - 设链表共有 a+b 个节点，其中 链表头部到链表入口 有 a个节点（不计链表入口节点）， 链表环 有 b 个节点（这里需要注意，a和 b是未知数；设两指针分别走了 f，s步，则有：
            1. fast 走的步数是slow步数的 2 倍，即 f = 2s；（解析： fast 每轮走 2 步）
            2. fast 比 slow多走了 n个环的长度，即 f = s + nb；（解析： 双指针都走过 a 步，然后在环内绕圈直到重合，重合时 fast 比 slow 多走 环的长度整数倍）；
            3. 以上两式相减得：f = 2nb，s = nb，即fast和slow 指针分别走了 2n，n 个 环的周长 （注意： n 是未知数，不同链表的情况不同）。
2. 目前情况分析：
    - 如果让指针从链表头部一直向前走并统计步数k，那么所有 走到链表入口节点时的步数是：k=a+nb（先走 a步到入口节点，之后每绕 1 圈环（ b 步）都会再次到入口节点）。
    - 而目前，slow 指针走过的步数为 nb 步。因此，我们只要想办法让 slow 再走 a 步停下来，就可以到环的入口。
    - 但是我们不知道 a 的值，该怎么办？依然是使用双指针法。我们构建一个指针，此指针需要有以下性质：此指针和slow 一起向前走 a 步后，两者在入口节点重合。那么从哪里走到入口节点需要 a 步？答案是链表头部head。
3. 双指针第二次相遇：

- slow指针 位置不变 ，将fast指针重新 指向链表头部节点 ； slow和fast同时每轮向前走 1 步；
  - TIPS：此时 f = 0，s = nb；
- 当 fast 指针走到f = a步时，slow 指针走到步s = a+nb，此时 两指针重合，并同时指向链表环入口 。

4. 返回slow指针指向的节点。

```
class Solution(object):
    def detectCycle(self, head):
        fast, slow = head, head
        while True:
            if not (fast and fast.next): return
            fast, slow = fast.next.next, slow.next
            if fast == slow: break
        fast = head
        while fast != slow:
            fast, slow = fast.next, slow.next
        return fast
```

146、LRU缓存机制

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制 。
实现 LRUCache 类：

- LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
- int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
- void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

- **高分解答**

  1、哈希表+双向链表

1. 哈希表用于满足题目时间复杂度O(1)的要求，双向链表用于存储顺序
2. 哈希表键值类型：<Integer, ListNode>，哈希表的键用于存储输入的key，哈希表的值用于存储双向链表的节点
3. 双向链表的节点中除了value外还需要包含key，因为在删除最久未使用的数据时，需要通过链表来定位hashmap中应当删除的键值对
4. 一些操作：双向链表中，在后面的节点表示被最近访问
    - i. 新加入的节点放在链表末尾，addNodeToLast(node)
    - ii. 若容量达到上限，去除最久未使用的数据，removeNode(head.next)
    - iii. 若数据新被访问过，比如被get了或被put了新值，把该节点挪到链表末尾，moveNodeToLast(node)
5. 为了操作的方便，在双向链表头和尾分别定义一个head和tail节点。

```
class ListNode:
    def __init__(self, key = 0, val = 0):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.hashmap = {}
        self.head = ListNode()
        self.tail = ListNode()
        self.head.next = self.tail
        self.tail.prev = self.head

    def remove_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def add_node_to_last(self, node):
        self.tail.prev.next = node
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev = node

    def move_node_to_last(self, node):
        self.remove_node(node)
        self.add_node_to_last(node)

    def get(self, key: int) -> int:
        if key not in self.hashmap:
            return -1
        node = self.hashmap[key]
        self.move_node_to_last(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.hashmap:
            node = self.hashmap[key]
            node.val = value
            self.move_node_to_last(node)
            return
```

```
        if len(self.hashmap) == self.capacity:
            del self.hashmap[self.head.next.key]
            self.remove_node(self.head.next)
        node = ListNode(key, value)
        self.hashmap[key] = node
        self.add_node_to_last(node)
```

2、有序字典OrderedDict

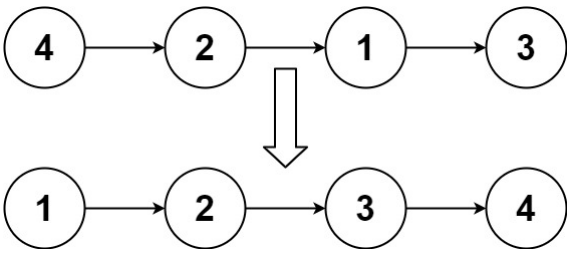
colletions包里的有序字典类OrderedDict底层基于哈希表和双链表，可在快速访问元素的同时维护元素的顺序

1. get
    - 如果key不在缓存， 返回-1
    - key在缓存，返回key的值，并将key对应的item移至有序字典尾部
2. put
    - 如果缓存区满且key未在缓存，弹出有序字典头部最久未使用的item
    - 更新字典，并将key对应的item移至有序字典尾部

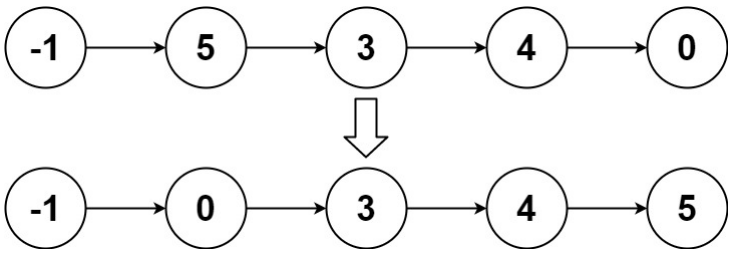
```
class LRUCache:

    def __init__(self, capacity: int):
        self.od, self.cap = collections.OrderedDict(), capacity

    def get(self, key: int) -> int:
        if key in self.od.keys():
            self.od.move_to_end(key)
            return self.od[key]
        else: return -1

    def put(self, key: int, value: int) -> None:
        if len(self.od) == self.cap and key not in self.od.keys():
self.od.popitem(last=False)
        self.od[key] = value
        self.od.move_to_end(key)
```

148、排序链表

给你链表的头结点 `head` ，请将其按 **升序** 排列并返回 **排序后的链表** 。




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




```
输入：head = [-1,5,3,4,0]
输出：[-1,0,3,4,5]
```

- **第一想法**

  1、链表转换为列表，排序后再转换成链表

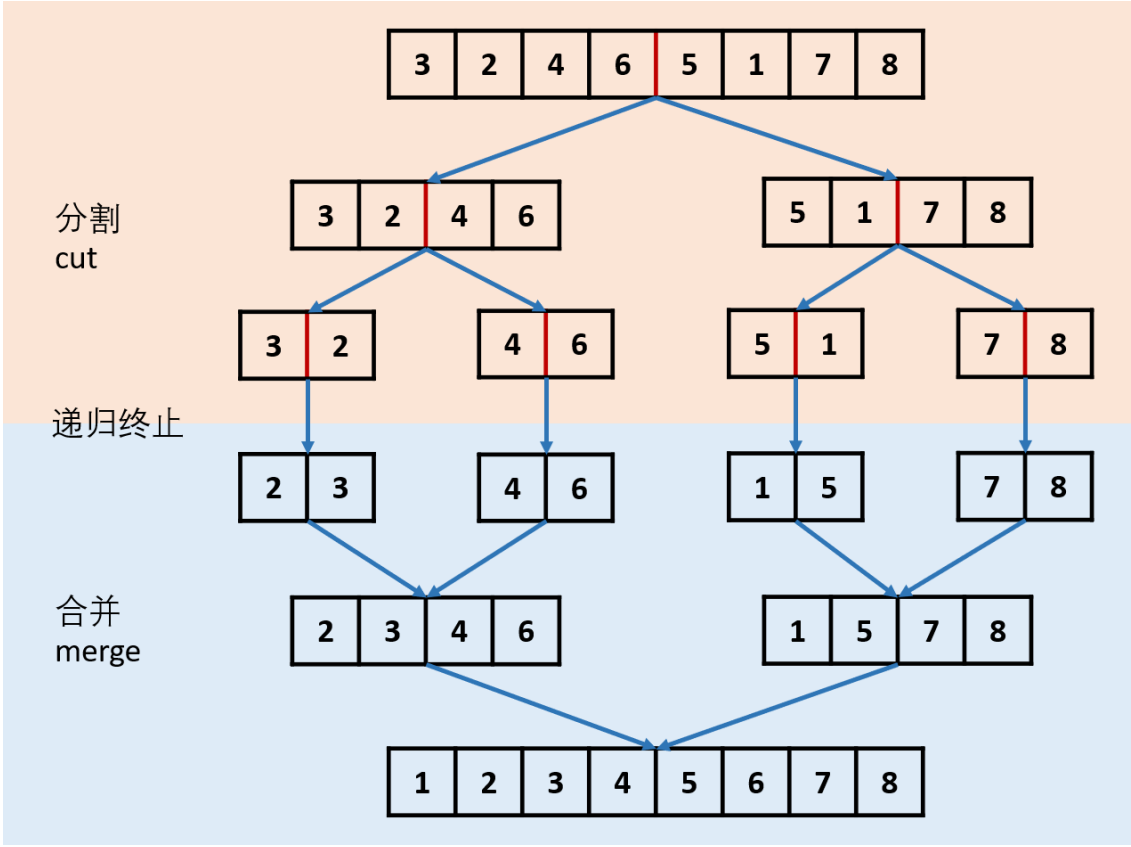
```python
class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        st=[]
        while head:
            st.append(head.val)
            head=head.next

        st=sorted(st)
        res=ListNode()
        cur=res
        while st:
            cur.next=ListNode(st.pop(0))
            cur=cur.next

        return res.next
```

- **高分解答**

  1、归并排序（快慢指针+合并有序链表）

  归并排序：每次将链表从中间断开，各自排序，然后合并有序链表。

  - find_mid(self, head)快慢指针找链表中点，即第(n+1)//2个节点，下面是细节。
  - 链表长度=2n 返回第n个; 链表长度=2n+1 返回第n+1
    - 2n: fast每次跳2步，第一次不满足while条件的是第 2+2(n-1)个指针, 此时slow=1+(n-1)=n,
    - 2n+1: fast每次跳2步，第一次不满足while条件的是 2+2n, 此时slow=1+n.
  - merged(self, left, right) 用于合并两个有序链表




```
class Solution:
    # 归并排序
    def sortList(self, head: ListNode) -> ListNode:
        if not head or not head.next: return head
        left_end = self.find_mid(head)
        mid = left_end.next
        left_end.next = None
        left, right = self.sortList(head), self.sortList(mid)
        return self.merged(left, right)
    # 快慢指针查找链表中点
    def find_mid(self, head):
        if head is None or head.next is None: return head
        slow,fast = head, head.next
        while fast is not None and fast.next is not None:
            slow=slow.next
            fast=fast.next.next
        return slow
    # 合并有序链表
    def merged(self, left, right):
        res = ListNode()
        h = res
        while left and right:
            if left.val < right.val:
                h.next, left = left, left.next
            else:
                h.next, right = right, right.next
            h = h.next
        h.next = left if left else right
        return res.next
```

2、快速排序（）

- 此题由于一些测试用例使用了worst case, 例如递增或递减, 此时会退化成O(N*2), 导致超时, 所以该思路仅供参考

```python
class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        def quicksort(node):
            if not node:
                return [node, node]
            # 设置pivot节点为头结点, 同时遍历起点设为其next
            pivot = node
            cur = pivot.next
            # 孤立pivot节点, 子区间排完序之后再单独处理pivot节点
            pivot.next = None
            # 初始化左右子区间哨兵节点
            leftdummy = ListNode(0)
            left = leftdummy
            rightdummy = ListNode(0)
            right = rightdummy
            # 遍历整个链表, 根据和pivot的关系分成左右两个子区间
            while cur:
                nex = cur.next
                cur.next = None
                if cur.val <= pivot.val:
                    left.next = cur
                    left = left.next
                else:
                    right.next = cur
                    right = right.next
                cur = nex
            # 得到左右子区间排序后的结果
            lefthead, lefttail = quicksort(leftdummy.next)
            righthead, righttail = quicksort(rightdummy.next)
            # 得到当前区间的头尾(根据左右子区间存在与否来决定)
            head = lefthead if lefthead else pivot
            tail = righttail if righttail else pivot
            # 连接pivot和左右区间
            if lefttail:
                lefttail.next = pivot
            pivot.next = righthead
            return [head, tail]

        return quicksort(head)[0]
```

152、乘积最大子数组

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

> 输入: [2,3,-2,4]
> 输出: 6
> 解释: 子数组 [2,3] 有最大乘积 6。

> 输入: [-2,0,-1]
> 输出: 0
> 解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

- **第一想法**

  1、 动态规划

- **高分解答**

  1、滑动窗口

  `product(i, j) = product(0, j) / product(0, i)` 从数组 i 到 j 的累乘等于 从数组开头到 j 的累乘除以从数组开头到 i 的累乘(这里先忽略 0 的情况)，要考虑三种情况

  - 累乘的乘积等于 0，就要重新开始
  - 累乘的乘积小于 0，要找到前面最大的负数，这样才能保住从 i 到 j 最大
  - 累乘的乘积大于 0，要找到前面最小的正数，同理!

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        if not nums: return
        cur_pro = 1             # 目前的累乘
        min_pos = 1             # 前面最小的正数
        max_neg = float("-inf")  # 前面最大的负数
        res = float("-inf")      # 结果

        for num in nums:
            cur_pro *= num
            # 大于0
            if cur_pro > 0:
                res = max(res, cur_pro // min_pos)
                min_pos = min(min_pos, cur_pro)
            # 小于0
            elif cur_pro < 0:
                if max_neg != float("-inf"):
                    res = max(res, cur_pro // max_neg)
                else:
                    res = max(res, num)
                max_neg = max(max_neg, cur_pro)
            # 等于0
            else:
                cur_pro = 1
                min_pos = 1
                max_neg = float("-inf")
                res = max(res, num)
        return res
```

  2、动态规划

  我们只要记录前i的最小值, 和最大值, 那么 `dp[i] = max(nums[i] * pre_max, nums[i] * pre_min, nums[i])`, 这里0 不需要单独考虑, 因为当相乘不管最大值和最小值,都会置0

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        if not nums: return
        res = nums[0]
        pre_max = nums[0]
        pre_min = nums[0]
        for num in nums[1:]:
            cur_max = max(pre_max * num, pre_min * num, num)
            cur_min = min(pre_max * num, pre_min * num, num)
            res = max(res, cur_max)
            pre_max = cur_max
            pre_min = cur_min
        return res
```

3、根据符号个数

- 当负数个数为偶数时候，全部相乘一定最大
- 当负数个数为奇数时候，它的左右两边的负数个数一定为偶数，只需求两边最大值
- 当有 `0` 情况，重置就可以了

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        reverse_nums = nums[::-1]
        for i in range(1, len(nums)):
            nums[i] *= nums[i - 1] or 1      #当乘积为0时，置为1
            reverse_nums[i] *= reverse_nums[i - 1] or 1
        return max(nums + reverse_nums)
```

200、岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

> 输入：grid = [
> ["1","1","1","1","0"],
> ["1","1","0","1","0"],
> ["1","1","0","0","0"],
> ["0","0","0","0","0"]
>]
> 输出：1

> 输入：grid = [
> ["1","1","0","0","0"],
> ["1","1","0","0","0"],
> ["0","0","1","0","0"],
> ["0","0","0","1","1"]
>]
> 输出：3

- **高分解答**

1、深度优先搜索（DFS）

- 每次深搜可以遍历图中的一个连通块，所以，深搜的次数就是连通块的个数
- 每访问一个为`"1"`的位置，将其替换为`"0"`，代替visited数组标记节点是否被访问过

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        # 深度优先搜索：
        def dfs(i, j):
            grid[i][j] = '0'

            for x, y in [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]:
                if x>=0 and x<n and y>=0 and y<m and grid[x][y]=='1':
                    dfs(x, y)

        n, m = len(grid), len(grid[0])
        res = 0
        for i in range(n):
            for j in range(m):
                # 深搜的次数就是岛屿的个数
                if grid[i][j] == '1':
                    res += 1
                    dfs(i, j)
        return res
```

2、广度优先搜索（BFS）

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        # 广度优先搜索：
        def bfs(i, j):
            queue = [(i, j)]
            grid[i][j] = "0"
            while queue:
                x, y = queue.pop(0)
                for a, b in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
                    if a>=0 and a<n and b>=0 and b<m and grid[a][b]=="1":
                        queue.append((a, b))
                        grid[a][b] = "0"

        n, m = len(grid), len(grid[0])
        res = 0
        for i in range(n):
            for j in range(m):
                if grid[i][j] == "1":
                    res += 1
                    bfs(i, j)
        return res
```

3、并查集

- 初始化每个位置为一个集合，用`i*col+j`唯一标识每个元素*
- *初始化`size=col*row`
- 合并连通的元素为一个集合，每合并一次，size-1

- 最终size = 连通块的个数 + 水的元素个数
  - 故在遍历每个元素的时候，统计水的个数
  - 岛屿数量 = size - 水的个数

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        # 并查集：使用横坐标*列数+纵坐标作为元素在并查集中的唯一标识
        # 岛屿数量 = （总元素个数 - 水的个数）中的连通块的个数
        class UnionFind:
            def __init__(self, n):
                # 总数
                self.size = n
                self.p = [i for i in range(n)]

            def find(self, x):
                # 查找根节点，即当前元素所属的集合
                if self.p[x] != x:
                    self.p[x] = self.find(self.p[x])
                return self.p[x]

            def union(self, a, b):

                ar, br = self.find(a), self.find(b)
                # 两个元素位于同一个集合，跳过
                if ar == br:
                    return
                # 不在同一个集合，合并
                else:
                    self.p[ar] = br
                    self.size -= 1

        n, m = len(grid), len(grid[0])
        ocean = 0   # 统计水的个数

        uf = UnionFind(n*m)

        for i in range(n):
            for j in range(m):
                # 统计水的个数
                if grid[i][j] == "0":
                    ocean += 1
                else:
                    # 只需向右和向下查看
                    if i+1 < n and grid[i+1][j]=="1":
                        uf.union(i*m+j, (i+1)*m+j)
                    if j+1 < m and grid[i][j+1]=="1":
                        uf.union(i*m+j, i*m+(j+1))
        return uf.size - ocean
```

207、课程表

你这个学期必须选修 numCourses 门课程，记为 0 到 numCourses - 1 。

在选修某些课程之前需要一些先修课程。 先修课程按数组 prerequisites 给出，其中 prerequisites[i] = [ai, bi] ，表示如果要学习课程 ai 则 必须 先学习课程 bi 。

- 例如，先修课程对 [0, 1] 表示：想要学习课程 0 ，你需要先完成课程 1 。

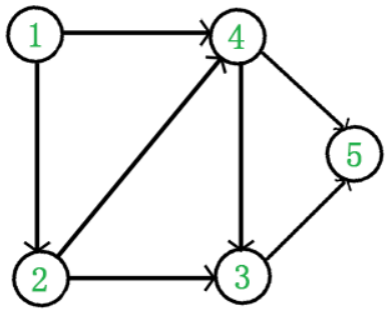
请你判断是否可能完成所有课程的学习？如果可以，返回 true ；否则，返回 false 。

> 输入：numCourses = 2, prerequisites = [[1,0]]
> 输出：true
> 解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0 。这是可能的。

> 输入：numCourses = 2, prerequisites = [[1,0],[0,1]]
> 输出：false
> 解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0 ；并且学习课程 0 之前，你还应先完成课程 1 。这是不可能的。

- **高分解答**

  1、有向无环图+拓扑排序

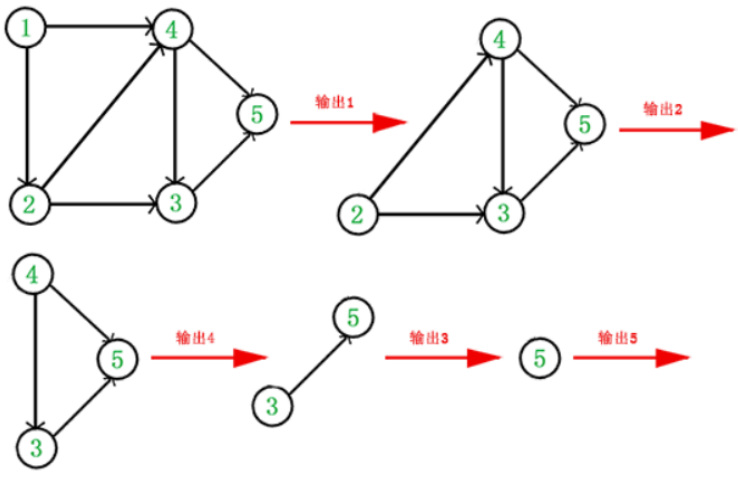


它是一个 DAG 图，那么如何写出它的拓扑排序呢？这里说一种比较常用的方法：

1：从 DAG 图中选择一个 没有前驱（即入度为0）的顶点并输出。

2：从图中删除该顶点和所有以它为起点的有向边。

3：重复 1 和 2 直到当前的 DAG 图为空或当前图中不存在无前驱的顶点为止。后一种情况说明有向图中必然存在环。



于是，得到拓扑排序后的结果是 { 1, 2, 4, 3, 5 }。

```
from collections import deque

class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) ->
bool:
        indegrees = [0 for _ in range(numCourses)]
        adjacency = [[] for _ in range(numCourses)]
        queue = deque()
        # 1、节点的入度：指向该点的其他点的数量
        # 2、相连边
        for cur, pre in prerequisites:
            indegrees[cur] += 1
            adjacency[pre].append(cur)
        # 从入度为0的节点开始遍历：
        num = 0
        for i in range(len(indegrees)):
            if not indegrees[i]:
                queue.append(i)
                num += 1
        # BFS TopSort.
        # 减小该点所指向的点的入度，若为0则继续遍历：
        while queue:
            pre = queue.popleft()
            for cur in adjacency[pre]:
                indegrees[cur] -= 1
                if not indegrees[cur]:
                    queue.append(cur)
                    num += 1
        return num >= numCourses
```

2、DFS：原理是通过 DFS 判断图中是否有环

1. 借助一个标志列表 flags，用于判断每个节点 i （课程）的状态：
   - 未被 DFS 访问：i == 0；
   - 已被其他节点启动的 DFS 访问：i == -1；
   - 已被当前节点启动的 DFS 访问：i == 1。
2. 对 numCourses 个节点依次执行 DFS，判断每个节点起步 DFS 是否存在环，若存在环直接返回 False。DFS 流程；
   1. 终止条件：
      - 当 flag[i] == -1，说明当前访问节点已被其他节点启动的 DFS 访问，无需再重复搜索，直接返回 True。
      - 当 flag[i] == 1，说明在本轮 DFS 搜索中节点 i 被第 22 次访问，即 课程安排图有环，直接返回 FalseFalse。
   2. 将当前访问节点 i 对应 flag[i] 置 1，即标记其被本轮 DFS 访问过；
   3. 递归访问当前节点 i 的所有邻接节点 j，当发现环直接返回 False；
   4. 当前节点所有邻接节点已被遍历，并没有发现环，则将当前节点 flag 置为 -1并返回 True。
3. 若整个图 DFS 结束并未发现环，返回 TrueTrue。

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) ->
bool:
        def dfs(i, adjacency, flags):
            if flags[i] == -1: return True
```

```
            if flags[i] == 1: return False
            flags[i] = 1
            for j in adjacency[i]:
                if not dfs(j, adjacency, flags): return False
            flags[i] = -1
            return True

        adjacency = [[] for _ in range(numCourses)]
        flags = [0 for _ in range(numCourses)]
        for cur, pre in prerequisites:
            adjacency[pre].append(cur)
        for i in range(numCourses):
            if not dfs(i, adjacency, flags): return False
        return True
```

208、实现Trie（前缀树）

Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

- Trie() 初始化前缀树对象。
- void insert(String word) 向前缀树中插入字符串 word 。
- boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
- boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。

> 输入
> ["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
> [[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
> 输出
> [null, null, true, false, true, null, true]
>
> 解释
> Trie trie = new Trie();
> trie.insert("apple");
> trie.search("apple");   // 返回 True
> trie.search("app");   // 返回 False
> trie.startsWith("app"); // 返回 True
> trie.insert("app");
> trie.search("app");   // 返回 True

- **高分解答**

  1、字典树

```
class Trie:
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.lookup = {}
```

```
    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        tree = self.lookup
        for a in word:
            if a not in tree:
                tree[a] = {}
            tree = tree[a]
        # 单词结束标志
        tree["#"] = "#"             # self.lookup={'a': {'p': {'p': {'l':
{'e': {'#': '#'}}}}}}
                                    # 之所以要tree = tree[a]就是要顺着要查询的单
词的每一个字母，一层一层跳进子字典里

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        tree = self.lookup
        for a in word:
            if a not in tree:
                return False
            tree = tree[a]
        if "#" in tree:
            return True
        return False

    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given
prefix.
        """
        tree = self.lookup
        for a in prefix:
            if a not in tree:
                return False
            tree = tree[a]
        return True
```

215、数组中的第K个最大元素

在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

> ```
> 输入: [3,2,1,5,6,4] 和 k = 2
> 输出: 5
> ```
>
> ```
> 输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
> 输出: 4
> ```

- **第一想法**

  1、`sorted()`函数自动排序

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        nums=sorted(nums,reverse=True)
        return nums[k-1]
```

  2、循环k次，每次查找最大值，并删除

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        for i in range(k):
            x=max(nums)
            nums.remove(x)

        return x
```

- **高分解答**

  1、堆排序

  - 库函数:

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        return heapq.nlargest(k, nums)[-1]
```

  - 手写堆

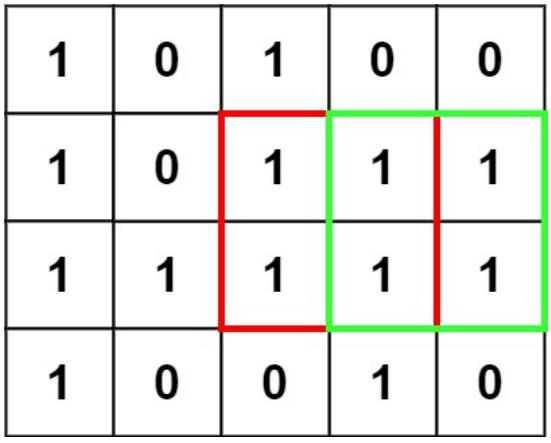
```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def adjust_heap(idx, max_len):
            left = 2 * idx + 1
            right = 2 * idx + 2
            max_loc = idx
            if left < max_len and nums[max_loc] < nums[left]:
                max_loc = left
            if right < max_len and nums[max_loc] < nums[right]:
                max_loc = right
            if max_loc != idx:
                nums[idx], nums[max_loc] = nums[max_loc], nums[idx]
                adjust_heap(max_loc, max_len)

        # 建堆
        n = len(nums)
        for i in range(n // 2 - 1, -1, -1):
            adjust_heap(i, n)
        #print(nums)
        res = None
        for i in range(1, k + 1):
            #print(nums)
            res = nums[0]
            nums[0], nums[-i] = nums[-i], nums[0]
            adjust_heap(0, n - i)
        return res
```

2、快速排序

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def partition(left, right):
            pivot = nums[left]
            l = left + 1
            r = right
            while l <= r:
                if nums[l] < pivot and nums[r] > pivot:
                    nums[l], nums[r] = nums[r], nums[l]
                if nums[l] >= pivot:
                    l += 1
                if nums[r] <= pivot:
                    r -= 1
            nums[r], nums[left] = nums[left], nums[r]
            return r
        left = 0
        right = len(nums) - 1
        while 1:
            idx = partition(left, right)
            if idx == k - 1:
                return nums[idx]
            if idx < k - 1:
                left = idx + 1
            if idx > k - 1:
                right = idx - 1
```

221、最大正方形

在一个由 `'0'` 和 `'1'` 组成的二维矩阵内，找到只包含 `'1'` 的最大正方形，并返回其面积。



输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
输出：4

- **高分解答**

  1、 动态规划

  - `dp[i][j]`表示以第i行，第j列处为右下角的最大正方形的边长。
  - 仅当该位置为1时，才有可能存在正方形。且递推公式为：

- `dp[i][j]=min(dp[i-1][j-1],dp[i-1][j],dp[i][j-1])+1`
- 含义为若当前位置为1，则此处可以构成的最大正方形的边长，是其正上方，左侧，和左上界三者共同约束的，且为三者中的最小值加1。
  - 特判，若matrix为空，返回0
  - 初试化matrix的行m，列n，初始化dp=[[0,⋯,0],⋯,[0,⋯,0]]，维度为(m+1)*(n+1)，这样便于处理。初试化最大边长res=0。
  - 遍历dp数组，遍历行i，遍历区间[1,m+1):
    - 遍历列jj，遍历区间[1,n+1)[1,n+1):
    - 若`matrix[i-1][j-1]=="1"`，此时可能存在正方形:
    - `dp[i][j]=min(dp[i-1][j-1],dp[i-1][j],dp[i][j-1])+1`
    - 并更新最大边长`res=max(res,dp[i][j])`
- **输入:**

  ```
  1 0 1 0 0
  1 0 1 1 1
  1 1 1 1 1
  1 0 0 1 0
  ```

  输出: 4
- dp

0	0	0	0	0	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	1	2	2
0	1	0	0	1	0


```python
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        if(not matrix):
            return 0
        m=len(matrix)
        n=len(matrix[0])
        res=0
        dp=[[0]*(n+1) for _ in range(m+1)]
        for i in range(1,m+1):
            for j in range(1,n+1):
                if(matrix[i-1][j-1]=="1"):
                    dp[i][j]=min(dp[i-1][j-1],dp[i-1][j],dp[i][j-1])+1
                    res=max(dp[i][j],res)
        return res*res
```

2、二进制思路

- 首先，把矩阵的每一行看作一个二进制数，对于矩阵:
  1 0 1 0 0
  1 0 1 1 1
  两数相与得到
  1 0 1 0 0
  在这个数字中，连续的 1 就可以看作是正方形的宽，而高度就是 2，因为是 2 个数相与的。

要得到正方形，宽和高必须相等，那么就是取宽高中较小的那个作为正方形的边，所以能构成的最大正方形边长为 min(宽,高)=min(1,2)=1。

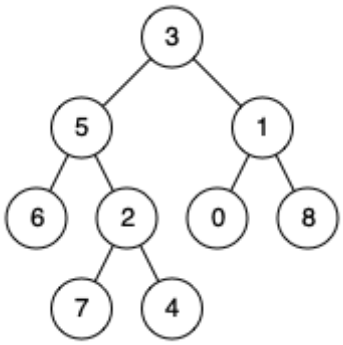
- 再来看一个矩阵：
  1 0 1 1 1
  1 1 1 1 1
  相与得到
  1 0 1 1 1
  连续的 1 有 3 个，所以宽是 3，高仍然是 2，构成的最大正方形边长为 min(宽,高)=min(3,2)=2。
- 现在我们的任务有 3 个：
  1. 把矩阵每一行看作二进制数的表示形式，进而转成数字，得到一个一维数组；
  2. 用 2 个指针 i,j 分别表示开始行和最后一行，这2行之间的所有行（包括这 2 行）全部相与，看能得到多少个连续最多的 1，亦即求宽度，而高度则等于 j-i+1。枚举所有的(i,j) 组合，并保存结果中的最大值。
  3. 如何求一个数中连续最多的 1？小技巧：每次将这个数和它左移一位后的数相与，直到它变成 0，记录操作次数，这个操作次数就是连续最多的 1。
     举个例子：
     - 对于数字 10101010，将这个数和它左移一位后的数相与，操作1次，这个数就变成 0 了，连续最多的 1只有 1 个；
     - 对于数字 10101110，操作 1 次得到 00001100，还需再操作2次才能得到 0，所以连续最多的 1是 3；
     - 相当于每次都错一位相与，所以这个数最多有几个连续的 1，就能错一位相与几次才能得到 0。

```
class Solution:
    def getWidth(self,num):  #步骤3：求一个数中连续最多的1
        w=0
        while num>0:
            num&=num<<1
            w+=1
        return w
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        nums=[int(''.join(n),base=2) for n in matrix]  #步骤1：每一行当作二进制数
        res,n=0,len(nums)
        for i in range(n):   #步骤2：枚举所有的组合，temp存储相与的结果
            temp=nums[i]
            for j in range(i,n):
                temp&=nums[j]
                w=self.getWidth(temp)
                h=j-i+1
                res=max(res,min(w,h))
        return res*res
        #优化
        #当某一行与后面行相与的时候，1 的个数只可能变少，即宽度只会越变越小，当宽度（连续1
        #的个数）小于高度的时候，会以宽度作为正方形的边长，此时高度再增加已经没有意义了，因
        #为不可能再有最大值产生了，所以可以跳出循环，直接开始遍历下一个 start 行：
        for j in range(i,n):
                temp&=nums[j]
                if self.getWidth(temp)<j-i+1:
                    break
                res=max(res,j-i+1)
```

236、二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

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




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

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

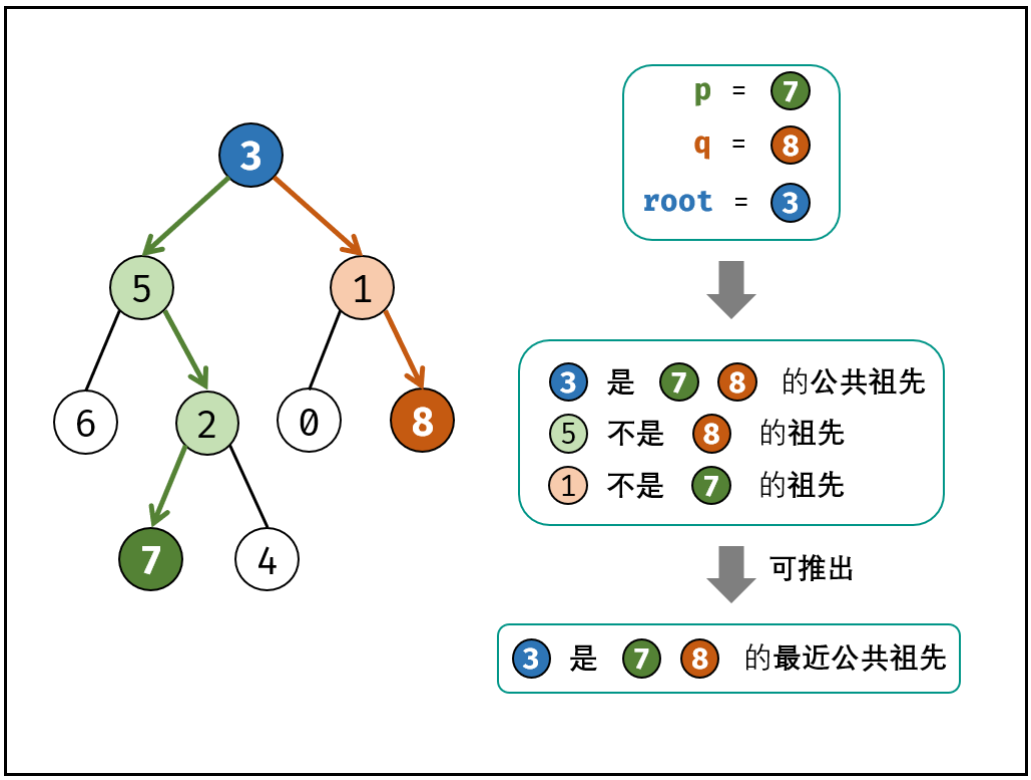
- **高分解答**

  1、后序遍历（DFS）动画演示

  - 最近公共祖先的定义： 设节点 root 为节点 p, q的某公共祖先，若其左子节点 root.left和右子节点 root.right都不是 p,q的公共祖先，则称 root是 “最近的公共祖先” 。

    根据以上定义，若 root 是 p, q 的 最近公共祖先 ，则只可能为以下情况之一：

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



- 考虑通过递归对二叉树进行后序遍历，当遇到节点 p 或 q时返回。从底至顶回溯，当节点 p, q 在节点 root的异侧时，节点 root即为最近公共祖先，则向上返回 root 。

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

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        # 边界条件，如果匹配到left或right就直接返回停止递归
        if not root or root == p or root == q: return root
        # 这两行代码可以无脑先写好！
        # 因为是DFS算法，这个模板可以无脑套用，写上之后可能你思路就清晰很多
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if not left: return right
        if not right: return left
        return root
```

情况 `1.`，`2.`，`3.`，`4.` 的展开写法如下。

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        if not root or root == p or root == q: return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if not left and not right: return # 1.
        if not left: return right # 3.
        if not right: return left # 4.
        return root # 2. if left and right:
```

2、使用字典存储双亲结点

由于每个节点只有唯一一个父节点，我们可以使用到字典的value-key的形式（节点-父节点）字典中预置根节点的父节点为None。

字典建立完成后，二叉树就可以看成一个所有节点都将最终指向根节点的链表了。

于是在二叉树中寻找两个节点的最小公共节点就相当于，在一个链表中寻找他们相遇的节点

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
        dic = {root:None}
        def dfs(node):
            if node:
                if node.left:
                    dic[node.left] = node
                if node.right:
                    dic[node.right] = node
                dfs(node.left)
                dfs(node.right)
        dfs(root)
        #记公共祖先节点是A，整棵树的根节点为root， P、Q距离A的步数分别是：a、b步，A
距
        #离root距离是c步， 那么P沿着父节点方向往上走a+c步就到达root节点，这时候让P
再
        #沿着Q到A的路径走b步，就停在A处，此时P的路程是a+c+b；同样的方法，Q的路程是
        #b+c+a。 发现两者同时走到A，即判定条件 l1==l2时要跳出循环。另外，如果a==b
的
        #话，那么P，Q不用完全走完就会相遇。
        l1, l2 = p, q
        while(l1!=l2):
            l1 = dic.get(l1, q)      #走到头从新赋值l1=q
            l2 = dic.get(l2, p)
        return l1
```

238、除自身以外数组的乘积

给你一个长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output ，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

> 输入: [1,2,3,4]
> 输出: [24,12,8,6]
>
> 题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。
>
> 说明: 请不要使用除法，且在 O(n) 时间复杂度内完成此题。

- **高分解答**

  1、保存左积和右积

  1. 初始化数组长度nn。初始化res=[0,0,...,0]为1*n的数组。初试化乘积k=1
  2. 从左向右遍历，遍历区间[0,n):
     - res每个位置保存它左侧所有元素的乘积。即res[i]=k,k*=nums[i]
  3. 重置乘积k=1，用来保存元素右边的乘积和
  4. 从右向左遍历，遍历区间(n,0]:
     - res[i]*=k，表示将当前位置的左积乘以右积。
     - 更新右积k*=nums[i]
  5. 返回res

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n=len(nums)
        res=[0]*n
        k=1
        for i in range(n):
            res[i]=k
            k=k*nums[i]
        k=1
        for i in range(n-1,-1,-1):
            res[i]*=k
            k*=nums[i]
        return res
```

  2、双指针

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        res, l, r = [1] * len(nums), 1, 1
        for i, j in zip(range(len(nums)), reversed(range(len(nums)))):
            #l,r,分别保存左积和右积
            res[i], l = res[i] * l, l * nums[i]
            res[j], r = res[j] * r, r * nums[j]
        return res
```

240、搜索二维矩阵II

编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

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

```
输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 5
输出：true
```

```
输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 20
输出：false
```

- **第一想法**

  1、暴力搜索，循环每行，查询是否存在该数字

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        for i in range(len(matrix)):
            if target in matrix[i]:
                return True
        return False
```

  2、递归

  - 从右上角开始递归，大于target向左移动，小于target向下移动，设定边界条件

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m,n=len(matrix),len(matrix[0])
        def dfs(i,j):
            if i==m or j<0 :
                return 0

            if matrix[i][j]==target:return 1
            elif matrix[i][j]>target:return dfs(i,j-1)
            else:return dfs(i+1,j)

        return dfs(0,n-1)
```

- **高分解答**

  1、折线搜索

  - 整体思路是尽可能多的排除掉无关 行/列 ，可以从 第一排最后一个/第一列最后一个 开始搜索，这里选择从 第一排最后一个 开始
  - 由于行列规则等价，搜索策略 先按行排除/按列排除 也是等价的，这里选择 按行排除
  - 搜索规则：小于 target 则向左搜索，大于 则向下搜索，可以保证 global search
  - 若超出 矩阵大小 则意味着没有匹配 target，输出 False

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        row = len(matrix)
        col = len(matrix[0])
        i = 0
        j = col - 1
        while matrix[i][j] != target:
            if matrix[i][j] < target:
                i += 1
            else:
                j -= 1
            if i >= row or j < 0:
                return False
        return True
```

253、会议室II（会员专属）

279、完全平方数

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

给你一个整数 n ， 返回和为 n 的完全平方数的 最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

> 输入：n = 12
> 输出：3
> 解释：12 = 4 + 4 + 4

> 输入：n = 13
> 输出：2
> 解释：13 = 4 + 9

- **第一想法**

  1、深度优先遍历：每条路都走一遍，输出最短的路——超出时间限制

```python
def numSquares_recur(self, n: int) -> int:
        def recur(n):
            if n == 0:
                return 0
            res = float('inf')
            for i in range(1, int(n**0.5) + 1):
                res = min(res, recur(n - i*i) + 1)
            return res
        return recur(n)
```

- **高分解答**

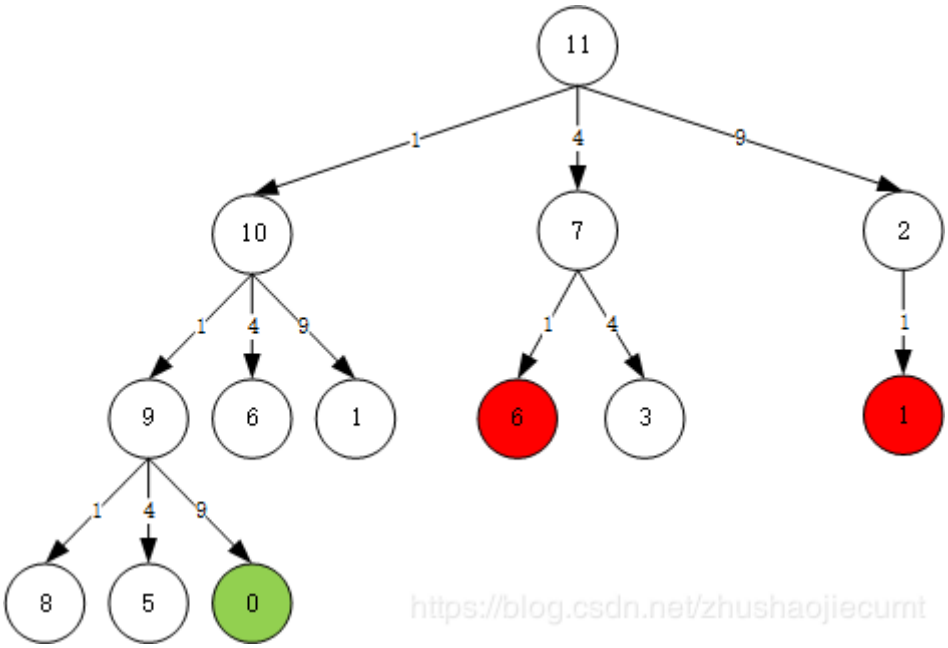
  1、动态规划

  dp[i]表示ii最少可以由几个平方数构成。

  1. 初试化dp=[0,1,2,⋯,n]，长度为n+1，最多次数就是全由1构成。
  2. 遍历dp，对于ii，遍历区间[2,n+1)[2,n+1)：
     - 遍历所有平方数小于i的数j，遍历区间`[1,int(\sqrt{i})+1)`
     - `dp[i]=min(dp[i],dp[i-j*j]+1)`。始终保存所有可能情况中的最小值。
  3. 返回dp[n]

```python
class Solution:
    def numSquares(self, n: int) -> int:
        dp=[i for i in range(n+1)]
        for i in range(2,n+1):
            for j in range(1,int(i**(0.5))+1):
                dp[i]=min(dp[i],dp[i-j*j]+1)
        return dp[-1]
```

  2、广度优先搜索



因为是广度优先遍历，顺序遍历每一行，所以当节点差出现0时，此时一定是最短的路径。
如绿色所示，若此时节点为0，表示根节点可以由路径上的平方数{1,1,9}{1,1,9}构成，返回此时的路径长度3，后续不在执行。
如红色所示，若节点值在之前已经出现，则不需要再计算，一定不会是最短路径，最短路径还未出现。

借助队列实现广度优先遍历（层次遍历）

1. 初始化队列queue=[n]，访问元组visited={}，初试化路径长度step=0
2. 特判，若n==0，返回0。
3. 循环条件，队列不为空：
    - step+=1，因为循环一次，意味着一层中的节点已经遍历完，所以路径长度需要加一。
    - 定义当前层中的节点数l=len(queue)，遍历当前层的所有节点：
        - 令tmp为队首元素。
        - 遍历所有可能数ii的平方数，遍历区间`[1,int(\sqrt{tmp})+1)`
            - 定义x=tmp-i^{2}
            - 若x==0，返回当前的路径长度。
              若x not in visited，表示当前节点未出现过：将该节点入队并在访问数组中加入。
4. 返回step

```
class Solution:
    def numSquares(self, n: int) -> int:
        from collections import deque
        if n == 0: return 0
        queue = deque([n])
        step = 0
        visited = set()
        while(queue):
            step+=1
            l=len(queue)
            for _ in range(l):
                tmp=queue.pop()
                for i in range(1,int(tmp**0.5)+1):
                    x=tmp-i**2
                    if(x==0):
                        return step
                    if(x not in visited):
                        queue.appendleft(x)
                        visited.add(x)
        return step
```

287、寻找重复数

给定一个包含 n + 1 个整数的数组 nums ， 其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。

假设 nums 只有 一个重复的整数 ， 找出 这个重复的数 。

> 输入：nums = [1,3,4,2,2]
> 输出：2

> 输入：nums = [1,1,2]
> 输出：1

- **第一想法**

1、循环一次，存储数组，搜索是否曾经出现（时间6%，空间100%）

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        b=[]
        for i in nums:
            if i not in b:
                b.append(i)
            else:
                return i
```

```
#时间5%，空间100%
#使用自身数组搜索
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        for i in range(len(nums)):
            if nums[i] in nums[:i]:
                return nums[i]
```

将数组换成字典，可以节省很多时间（时间95%，空间20%）

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        b={}
        for i in nums:
            if i not in b:
                b[i]=i
            else:
                return i
```

- **高分解答**

  1、二分法查找

  按题目表达，设数组长度为n，则数组中元素∈[1,n−1]，且只有一个重复元素。一个直观的想法，设一个数字k∈[1,n−1]，统计数组中小于等于k的数字的个数count：

  1. 若count<=k，说明重复数字一定在(k,n−1]的范围内。
  2. 若count>k，说明重复数字一定在[0,k]的范围内。

  利用这个性质，我们使用二分查找逐渐缩小重复数字所在的范围。

  1. 初试化左右 数字 边界left=1,right=n-1
  2. 循环条件left<right:
     - mid=(left+right)//2
     - 按照性质，统计数组中小于等于mid的元素个数count
     - 若 count<=mid，说明重复数字一定在(mid,right]的范围内。令left=mid+1
     - 若count>mid，说明重复数字一定在[left,mid]的范围内。令right=mid。
  3. 返回leftleft

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        left = 1
        right = len(nums) - 1
        while(left<right):
            mid=(left+right)//2
```

```
            count=0
            for num in nums:
                if(num<=mid):
                    count+=1
            if(count<=mid):
                left=mid+1
            else:
                right=mid
        return left
```

2、 快慢指针

1.找到环　2. 找到环的入口

找环：

1. 定义快慢指针slow=0,fast=0
2. 进入循环：
   - slow每次走一步，即slow=nums[slow]
   - fast每次走两步，即fast=nums[nums[fast]]
   - 当slow==fast时，退出循环。
     当快慢指针相遇时，一定在环内。此时假设slow走了k步，则fast走了2k步。设环的周长为c，则 `k\%c==0` 。

找环的入口：

1. 定义新的指针find=0
2. 进入循环：
   - find每次走一步，即find=nums[find]
   - slow每次走一步，即slow=nums[slow]
   - 当两指针相遇时，即find==slow，返回find
3. 为何相遇时，找到的就是入口：
   假设起点到环的入口(重复元素)，需要m步。此时slow走了n+m步，其中n是环的周长c的整数倍，所以相当于slow走了m步到达入口，再走了n步。所以相遇时一定是环的入口。

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        slow=0
        fast=0
        while(1):
            slow=nums[slow]
            fast=nums[nums[fast]]
            if(slow==fast):
                break
        find=0
        while(1):
            find=nums[find]
            slow=nums[slow]
            if(find==slow):
                return find
```

300、最长递增子序列

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

> 输入：nums = [10,9,2,5,3,7,101,18]
> 输出：4
> 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

> 输入：nums = [0,1,0,3,2,3]
> 输出：
> 4

- **高分解答**

  1、动态规划

  - 状态定义：
    - dp[i]的值代表以nums[i]结尾的最长子序列长度
  - 转移方程： 设 j∈[0,i)，考虑每轮计算新 dp[i] 时，遍历 [0,i) 列表区间，做以下判断：
    1. 当 `nums[i] > nums[j]` 时： nums[i] 可以接在 nums[j] 之后（此题要求严格递增），此情况下最长上升子序列长度为 dp[j] + 1 ；
    2. 当 `nums[i] <= nums[j]` 时： nums[i] 无法接在 nums[j]之后，此情况上升子序列不成立，跳过。
       - 上述所有 1. 情况 下计算出的 dp[j] + 1的最大值，为直到 i的最长上升子序列长度（即 dp[i]）。实现方式为遍历 j 时，每轮执行 `dp[i] = max(dp[i], dp[j] + 1)`
       - 转移方程： `dp[i] = max(dp[i], dp[j] + 1) for j in [0, i)`。
  - 初始状态：
    - dp[i] 所有元素置 1，含义是每个元素都至少可以单独成为子序列，此时长度都为 1。
  - 返回值：
    - 返回 dp 列表最大值，即可得到全局最长上升子序列长度。

```python
# Dynamic programming.
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums: return 0
        dp = [1] * len(nums)
        for i in range(len(nums)):
            for j in range(i):
                if nums[j] < nums[i]: # 如果要求非严格递增，此行 '<' 改为 '<=' 即可
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp)
```

  2、动态规划+二分查找动画演示

  - 状态定义：
    - tails[k]的值代表 长度为 k+1子序列 的尾部元素值。

- 转移方程： 设 res为 tails 当前长度，代表直到当前的最长上升子序列长度。设 j∈[0,res)，考虑每轮遍历 nums[k]时，通过二分法遍历 [0,res) 列表区间，找出 nums[k] 的大小分界点，会出现两种情况：
  - 区间中存在 tails[i] > nums[k]： 将第一个满足 tails[i] > nums[k] 执行 tails[i] = nums[k]； 因为更小的 nums[k]后更可能接一个比它大的数字（前面分析过）。
  - 区间中不存在 tails[i] > nums[k]： 意味着 nums[k] 可以接在前面所有长度的子序列之后，因此肯定是接到最长的后面（长度为 res ），新子序列长度为 res + 1。
- 初始状态：
  - 令 tails 列表所有值 =0。
- 返回值：
  - 返回 res ，即最长上升子子序列长度。

```
# Dynamic programming + Dichotomy.
class Solution:
    def lengthOfLIS(self, nums: [int]) -> int:
        tails, res = [0] * len(nums), 0
        for num in nums:
            i, j = 0, res
            while i < j:
                m = (i + j) // 2
                if tails[m] < num: i = m + 1 # 如果要求非严格递增，将此行 '<' 改为 '<=' 即可。
                else: j = m
            tails[i] = num
            if j == res: res += 1
        return res
```

309、最佳买卖股票时机含冷冻期

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

- **高分解答**

  1、动态规划

  - 首先量取prices的长度为n，对n进行判断，接着以[0,0]为一个单位对n进行循环得到dp
    - `dp[][]`这两个框前面代表天数，后面代表是否持有股票
    - `dp[i][0]`就是i-1天不持股或者i-1天持股然后i天卖出
    - `dp[i][1]`则不同了，按照我们正常的理解应该是i-1天持股或者i-1天不持股然后第i天持股

        - 但是我们这里有一个冷冻期，是需要隔一天的，这里`dp[i-2][0]`是利润，而`dp[i-2][0]`有可能是i-3天不持股或者i-3天持股i-2天卖出,无论哪种选择，当到`dp[i][1]`的时候如果是`dp[i-1][0]`是没有冷冻期的，只有是`dp[i-2][0]`中间才有i-1这一天作为冷冻期
    - 最后返回`dp[-1][0]`因为不持股利润是要大于等于持股的(价格等于0相等)

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        if n < 2: return 0
        dp = [[0,0] for i in range(n)]   #dp的初始化

        dp[0][0] = 0     #第0天不持股自然就为0了
        dp[0][1] = -prices[0]   #第0天持股，那么价格就是-prices[0]了
        #第1天不持股，要么第0天就不持股，要么就是第0天持股，然后第1天卖出
        dp[1][0] = max(dp[0][0],dp[0][1]+prices[1])
        #第一天持股，要么就是第0天就持股了，要么就是第0天不持股第1天持股
        dp[1][1] = max(dp[0][1],dp[0][0]-prices[1])

        for i in range(2,n):
            dp[i][0] = max(dp[i-1][0],dp[i-1][1]+prices[i])
            dp[i][1] = max(dp[i-1][1],dp[i-2][0]-prices[i])
        return dp[-1][0]
```

2、动态规划II

- 定义dp[i]表示第i天结束时的状态

```
dp[i][0]: 目前持有一只股票的最大收益
dp[i][1]: 目前不持有股票，且处于冷冻期的最大收益
dp[i][2]: 目前不持有股票，且不处于冷冻期的最大收益
```

- `dp[i][0]`（第i天结束时，持有一只股票）的转移来源：

```
a. 第i-1天持有股票，第i天什么都不做；
b. 第i-1天不持有股票且不处于冷冻期，则在第i天买入；

转移方程为：dp[i][0] = max(dp[i-1][0], dp[i-1][2] - prices[i])
```

- `dp[i][1]`（第i天结束时，不持有股票，且处于冷冻期）的转移来源：

```
a. 第i-1天持有股票，在第i天卖掉；

转移方程为：dp[i][1] = dp[i-1][0] + prices[i]
```

- `dp[i][2]`（第i天结束时，不持有股票，且不处于冷冻期）的转移来源：

```
a. 第i-1天不持有股票，且不处于冷冻期，第i天什么都不做；
b. 第i-1天不持有股票，且处于冷冻期，第i天什么都不做；

转移方程为：dp[i][2] = max(dp[i-1][2], dp[i-1][1])
```

- 初始状态

```
dp[0] = [-prices[0], 0, 0]
```

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices: return 0

        dp = [[0] * 3 for _ in range(len(prices))]
        dp[0][0] = -prices[0]
        for i in range(1, len(prices)):
            # 根据转移方程编写代码
            dp[i][0] = max(dp[i-1][0], dp[i-1][2] - prices[i])
            dp[i][1] = dp[i - 1][0] + prices[i]
            dp[i][2] = max(dp[i - 1][2], dp[i - 1][1])
        return max(dp[-1])
```

322、零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

> ```
> 输入：coins = [1, 2, 5], amount = 11
> 输出：3
> 解释：11 = 5 + 5 + 1
> ```
>
> ```
> 输入：coins = [2], amount = 3
> 输出：-1
> ```
>
> ```
> 输入：coins = [1], amount = 0
> 输出：0
> ```

- **第一想法**

  1、动态规划，同279完全平方数

  1. dp[j]代表含义：填满容量为j的背包最少需要多少硬币
  2. 初始化dp数组：因为硬币的数量一定不会超过amount，而amount <= 10^4 ，因此初始化数组值为10001；dp[0] = 0
  3. 转移方程：`dp[j] = min(dp[j], dp[j - coin] + 1)`
     当前填满容量j最少需要的硬币 = min(之前填满容量j最少需要的硬币, 填满容量 j - coin 需要的硬币 + 1个当前硬币)
  4. 返回dp[amount]，如果dp[amount]的值为10001没有变过，说明找不到硬币组合，返回-1

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [0] + [10001] * amount
        for coin in coins:
            for j in range(coin, amount + 1):
                dp[j] = min(dp[j], dp[j - coin] + 1)
        return dp[-1] if dp[-1] != 10001 else -1
```

- **高分解答**

1、背包问题全解

1. 01硬币找零问题（01背包

给定不同面额的硬币 `coins` 和总金额 `m`。每个硬币最多**选择一次**。计算可以凑成总金额所需的**最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 -1。

```python
def func_2(coins, m):
    f = [float('inf')] * (m + 1)
    f[0] = 0
    for c in coins:  # 枚举硬币总数
        for j in range(m, c-1, -1):  # 从大到小枚举金额，确保j-c >= 0.
            f[j] = min(f[j], f[j - c] + 1)
    return f[m] if f[m] != float('inf') else -1  # 如果为inf说明状态不可达，返回-1即可。
#输入：coins = [1, 2, 5], amount = 11
#f=[0,1,1,2,_,1,2,2,3,_,_,_]
```

2. 完全硬币找零问题（完全背包）

给定不同面额的硬币 coins 和总金额 m。每个硬币可以选择无数次。计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

```python
class Solution:
    def coinChange(self, coins, m):
        f = [float('inf')]*(m+1)
        f[0] = 0
        for c in coins:  # 枚举硬币种数
            for j in range(c, m+1):  # 从小到大枚举金额，确保j-c >= 0.
                    f[j] = min(f[j], f[j - c] + 1)
        return f[m] if f[m] != float('inf') else -1  # 如果为inf说明状态不可达，返回-1即可。
```

3. 多重硬币找零问题（多重背包）

给定不同面额的硬币 coins 和总金额 m。每个硬币选择的次数有限制为 s。计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

- 这里和完全硬币问题的的初始状态表态表示很相似。
- 考第i种硬币，我们可以不拿，或者拿1...k个，直到拿到个数的限制。
- `f[i][j] = min(f[i-1]f[j], f[i-1][j-c]+1, f[i-1][j-2*c]+2, ..., f[i-1][j-k*c]+k)`

```python
def func_3(coins, m, s):
    f = [float('inf')] * (m + 1)
    f[0] = 0
    for i in range(len(coins)):
        for j in range(m, coins[i]-1, -1):
            for k in range(1, s[i]+1):  # 枚举每个硬币的个数 [1, s[i]]
                if j >= k*coins[i]:  # 确保不超过金额 j
                    f[j] = min(f[j], f[j - k*coins[i]] + k)
    print(f)
    return -1 if f[m] > m else f[m]
```

2、递归（dfs）

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        def dp(n):
            if n == 0:
                return 0
            elif n < 0:
                return -1
            else:
                res = float("inf")
                for coin in coins:
                    subproblem = dp(n - coin)
                    if subproblem == -1:
                        continue
                    res = min(res, subproblem + 1)
                return res if res != float("inf") else -1

        return dp(amount)
```

337、打家劫舍Ⅲ

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

> 输入: [3,2,3,null,3,null,1]
>
> 3
> / \
> 2 3
> \ \
> 3 1
>
> 输出: 7
> 解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

```
输入: [3,4,5,1,3,null,1]

     3
 / \
4   5
/ \   \
1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

- **高分解答**

  1、动态规划

  - 利用深度优先搜索的后序遍历，自底向上地，保存以各节点为起始点时小偷可以盗取的最大收益（这里利用哈希表），最终返回以root为起始点时的最大收益。
  - 设函数f(node)表示打劫当前节点node时的最大收益；函数g(node)表示不打劫当前节点时的最大收益，则会有以下两种情况：
    - 打劫当前节点node，则node的左右子节点都不能被打劫，此时以node为根节点的树的最大收益为 `f(node)=g(node.left)+g(node.right)+node.val`
    - 不打劫当前节点node，则node的左右子节点可以被打劫或不打劫，此时以node为根节点的树的最大收益为 `g(node)=max(f(node.left),g(node.left))+max(f(node.right),g(node.right))`
  - 最终，完成两张表f和g的存储后，返回max(f(root),g(root))即可。
    代码

```python
class Solution:
    def rob(self, root: TreeNode) -> int:
        def dfs(root):
            if not root:
                return
            dfs(root.left)
            dfs(root.right)
            f[root] = g[root.left] + g[root.right] +root.val
            g[root] = max(f[root.left],g[root.left]) +
max(f[root.right],g[root.right])

        f = {None:0}
        g = {None:0}
        dfs(root)
        return max(f[root],g[root])
```

  2、分治（递归）

  - 对于当前节点node，有两种可能：
    - 打劫节点node，则以node为根节点的树的最大收益为不打劫左子节点时左子树的最大收益+不打劫右子节点时右子树的最大收益+node.val
    - 不打劫节点node，则node的左右子节点可以选择打劫或不打劫，以node为根节点的树的最大收益为左子树的最大收益max(打劫左子节点，不打劫左子节点)+右子树的最大收益max(打劫右子节点，不打劫右子节点)

- 可编写深度优先搜索的后序遍历函数实现上述思路。注意，此dfs函数的返回值有两个：打劫当前节点的最大收益，不打劫当前节点的最大收益。

```
class Solution:
    def rob(self, root: TreeNode) -> int:
        def _rob(root):
            if not root: return 0, 0  # 偷，不偷

            left = _rob(root.left)
            right = _rob(root.right)
            # 偷当前节点，则左右子树都不能偷
            v1 = root.val + left[1] + right[1]
            # 不偷当前节点，则取左右子树中最大的值
            v2 = max(left) + max(right)
            return v1, v2

        return max(_rob(root))
```

338、比特位计数

给定一个非负整数 **num**。对于 **0 ≤ i ≤ num** 范围中的每个数字 **i** ，计算其二进制数中的 1 的数目并将它们作为数组返回。

> 输入: 2
> 输出: [0,1,1]

> 输入: 5
> 输出: [0,1,1,2,1,2]

- **第一想法**

  1、循环num+1次，转成二进制统计1的个数

```
class Solution:
    def countBits(self, num: int) -> List[int]:
        res=[]
        for i in range(num+1):
            res.append(bin(i).count('1'))

        return res
```

  2、数字减1进行‘与’操作

  1. 如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减1，那么原来处在整数最右边的1就会变为0，原来在1后面的所有的0都会变成1(如果最右边的1后面还有0的话)。其余所有位将不会受到影响。
  2. 把原来的整数和减去1之后的结果做与运算，从原来整数最右边一个1那一位开始所有位都会变成0。

```
class Solution:
    def countBits(self, num: int) -> List[int]:
```

```
        def nums(n):
            count=0
            while n!=0:
                count+=1
                n=n&(n-1)
            return count

        res=[]
        for i in range(num+1):
            res.append(nums(i))

        return res
```

- **高分解答**

  1、递归

  - 如果 i 是偶数，那么它的二进制 1 的位数与 i / 2 的二进制 1 的位数相等；因为偶数的二进制末尾是 0，右移一位等于 i / 2，而二进制中 1 的个数没有变化。
  - 如果 i 是奇数，那么它的二进制 1 的位数 = i - 1 的二进制位数 + 1的二进制位数+1；因为奇数的二进制末尾是 1，如果把末尾的 1 去掉就等于 i - 1。又 i - 1 是偶数，所以奇数 i的二进制 1 的个数等于 i/2 中二进制 1 的位数 +1.

```
class Solution(object):
    def countBits(self, num):
        res = []
        for i in range(num + 1):
            res.append(self.count(i))
        return res

    def count(self, num):
        if num == 0:
            return 0
        if num % 2 == 1:
            return self.count(num - 1) + 1
        return self.count(num // 2)
```

2、记忆化搜索

所谓记忆化搜索，就是在每次递归函数结束的时候，把计算结果保存起来。这样的话，如果下次递归的时候遇到了同样的输入，则直接从保存的结果中直接查询并返回，不用再次递归。

```
class Solution(object):
    def countBits(self, num):
        self.memo = [0] * (num + 1)
        res = []
        for i in range(num + 1):
            res.append(self.count(i))
        return res

    def count(self, num):
        if num == 0:
            return 0
        if self.memo[num] != 0:
            return self.memo[num]
        if num % 2 == 1:
            res = self.count(num - 1) + 1
```

```
        else:
            res = self.count(num // 2)
        self.memo[num] = res
        return res
        #其实上面这个记忆化搜索的方法也可以不用 memomemo 数组，直接利用 resres 保存结果。
```

3、动态规划

```
class Solution:
    def countBits(self, num):
        res = [0] * (num + 1)
        for i in range(1, num + 1):
            res[i] = res[i >> 1] + (i & 1)
            #右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，>> 右边的数字指定了移动的位数

        return res
```

4、动态规划

- 任何一个大于等于1的数字n，它的二进制数中一的个数实际就是之前某个数n-k的一的个数+1。
- k很显然就是不大于n的2的整数次幂中的最大值。
- 在进行遍历的时候只需要额外维护一个k值，每当n>=k，k更新为k*2

```
class Solution:
    def countBits(self, num: int) -> List[int]:
        dp = [0]*(num+1)
        k = 1
        for i in range(1,num+1):
            if i >= k*2 :
                k = k*2
            dp[i] = dp[i-k]+1

        return dp
```

347、前K个高频元素

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

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

> 输入: nums = [1], k = 1
> 输出: [1]

- **第一想法**

  1、创建 元素：个数 字典，输出前个元素

```
```

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        dict_k={}
        for i in nums:
            if i not in dict_k:
                dict_k[i]=1
            else:
                dict_k[i]+=1
        n=0
        res=[]
        for i in sorted(dict_k.items(),key=lambda x:(x[1]),reverse=True):
            if n<k:
                res.append(i[0])
                n+=1
            else:
                break
        return res
```

- **高分解答**

1、直接排序

- 记录每个数字出现的次数
- 返回次数最高的k个数

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = collections.Counter(nums)
        return [item[0] for item in count.most_common(k)]
```

2、堆排序一

- 记录每个数字出现的次数
- 把数字和对应出现次数放入堆中
- 返回堆的前K大元素

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = collections.Counter(nums)
        heap = [(val, key) for key, val in count.items()]
        return [item[1] for item in heapq.nlargest(k, heap)]
```

3、 堆排序二

- 记录每个数字出现的次数
- 把数字和对应的出现次数放到堆中（小顶堆）
- 如果堆已满（大小>=k）且当前数的次数比堆顶大，用当前元素替换堆顶元素
- 返回堆中的数字部分

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = collections.Counter(nums)
        heap = []
        for key, val in count.items():
            if len(heap) >= k:
                if val > heap[0][0]:
                    heapq.heapreplace(heap, (val, key))
            else:
                heapq.heappush(heap, (val, key))
        return [item[1] for item in heap]
```

394、字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

> 输入：s = "3[a]2[bc]"
> 输出："aaabcbc"

> 输入：s = "3[a2[c]]"
> 输出："accaccacc"

> 输入：s = "2[abc]3[cd]ef"
> 输出："abcabccdcdcdef"

- **高分解答**

  1、辅助栈法动画演示

  构建辅助栈 `stack`， 遍历字符串 `s` 中每个字符 `c`；

  - 当 c 为数字时，将数字字符转化为数字 multi，用于后续倍数计算；
  - 当 c 为字母时，在 res 尾部添加 c；
  - 当 c 为 [时，将当前 multi 和 res 入栈，并分别置空置 0：
    - 记录此 [前的临时结果 res 至栈，用于发现对应] 后的拼接操作；
    - 记录此 [前的倍数 multi 至栈，用于发现对应] 后，获取 multi × [...] 字符串。
    - 进入到新 [后，res 和 multi 重新记录。
  - 当 c 为] 时，stack 出栈，拼接字符串 res = last_res + cur_multi * res，其中:
    - last_res是上个 [到当前 [的字符串，例如 "3[a2[c]]" 中的 a；
    - cur_multi是当前 [到] 内字符串的重复倍数，例如 "3[a2[c]]" 中的 2。

```python
class Solution:
    def decodeString(self, s: str) -> str:
        stack, res, multi = [], "", 0
        for c in s:
            if c == '[':
                stack.append([multi, res])
                res, multi = "", 0
            elif c == ']':
                cur_multi, last_res = stack.pop()
                res = last_res + cur_multi * res
            elif '0' <= c <= '9':
                multi = multi * 10 + int(c)  #出现两位数的数字
            else:
                res += c
        return res
```

2、递归法

- 当 s[i] == ']' 时，返回当前括号内记录的 res 字符串与] 的索引 i （更新上层递归指针位置）；
- 当 s[i] == '[' 时，开启新一层递归，记录此 [...] 内字符串 tmp 和递归后的最新索引 i，并执行 res + multi * tmp 拼接字符串。
- 遍历完毕后返回 res。

```python
class Solution:
    def decodeString(self, s: str) -> str:
        def dfs(s, i):
            res, multi = "", 0
            while i < len(s):
                if '0' <= s[i] <= '9':
                    multi = multi * 10 + int(s[i])
                elif s[i] == '[':
                    i, tmp = dfs(s, i + 1)
                    res += multi * tmp
                    multi = 0
                elif s[i] == ']':
                    return i, res
                else:
                    res += s[i]
                i += 1
            return res
        return dfs(s,0)
```

399、除法求值

给你一个变量对数组 equations 和一个实数值数组 values 作为已知条件，其中 equations[i] = [Ai, Bi] 和 values[i] 共同表示等式 Ai / Bi = values[i] 。每个 Ai 或 Bi 是一个表示单个变量的字符串。

另有一些以数组 queries 表示的问题，其中 queries[j] = [Cj, Dj] 表示第 j 个问题，请你根据已知条件找出 Cj / Dj = ? 的结果作为答案。

返回 所有问题的答案 。如果存在某个无法确定的答案，则用 -1.0 替代这个答案。如果问题中出现了给定的已知条件中没有出现的字符串，也需要用 -1.0 替代这个答案。

注意：输入总是有效的。你可以假设除法运算中不会出现除数为 0 的情况，且不存在任何矛盾的结果。

输入：equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]
输出：[6.00000,0.50000,-1.00000,1.00000,-1.00000]
解释：
条件：a / b = 2.0, b / c = 3.0
问题：a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
结果：[6.0, 0.5, -1.0, 1.0, -1.0]

输入：equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]
输出：[3.75000,0.40000,5.00000,0.20000]

输入：equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]
输出：[0.50000,2.00000,-1.00000,-1.00000]

- **高分解答**

1、Graph+BFS

1. 遍历路径，表示Graph。
2. 遍历nodes，对每个节点使用BFS，计算该点到其它可访问的点的weight
3. 遍历queries。

```
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        nodes = set()
        dct = {}
        for equation, value in zip(equations, values):
            nodes |= set(equation)
            if equation[0] in dct:
                dct[equation[0]][equation[1]] = value
            else:
                dct[equation[0]] = {equation[0]:1, equation[1]: value}
            if equation[1] in dct:
                dct[equation[1]][equation[0]] = 1/value
            else:
                dct[equation[1]] = {equation[1]:1, equation[0]: 1/value}
        for node in nodes:
            scaned = set()
            queue = [node]
            weight = [1]
            while queue:
                p = queue.pop()
                scaned.add(p)
                w = weight.pop()
                for k, v in dct[p].items():
                    if k not in scaned:
                        queue.append(k)
                        weight.append(w*v)
                        dct[node][k] = w*v
        ret = [-1] * len(queries)
        for i, query in enumerate(queries):
            if query[0] in dct and query[1] in dct[query[0]]:
```

```
                ret[i] = dct[query[0]][query[1]]
        return ret
```

2、DFS+剪枝

```
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float],
queries: List[List[str]]) -> List[float]:
        def dfs(c1, c2, v):
            if c1==c2: return v
            seen.add(c1)
            for [x1,x2],value in z:
                if x1==c1 and x2 not in seen:
                    if (res:= dfs(x2, c2, v*value)):
                        return res
                if x2==c1 and x1 not in seen:
                    if (res:= dfs(x1, c2, v/value)):
                        return res
            return

        s_t, z, ans = set(sum(equations, [])), list(zip(equations, values)),
list()
        for x, y in queries:
            res = -1.0
            if x in s_t and y in s_t:
                seen = set()
                res = r if (r:=dfs(x, y, 1.0)) else res
            ans.append(res)
        return ans
```

3、带权重的并查集

```
class UnionFind:
    def __init__(self):
        """
        记录每个节点的父节点
        记录每个节点到根节点的权重
        """
        self.father = {}
        self.value = {}

    def find(self,x):
        """
        查找根节点
        路径压缩
        更新权重
        """
        root = x
        # 节点更新权重的时候要放大的倍数
        base = 1
        while self.father[root] != None:
            root = self.father[root]
            base *= self.value[root]

        while x != root:
            original_father = self.father[x]
```

```
            ##### 离根节点越远，放大的倍数越高
            self.value[x] *= base
            base /= self.value[original_father]
            #####
            self.father[x] = root
            x = original_father

        return root

    def merge(self,x,y,val):
        """
        合并两个节点
        """
        root_x,root_y = self.find(x),self.find(y)

        if root_x != root_y:
            self.father[root_x] = root_y
            ##### 四边形法则更新根节点的权重
            self.value[root_x] = self.value[y] * val / self.value[x]

    def is_connected(self,x,y):
        """
        两节点是否相连
        """
        return x in self.value and y in self.value and self.find(x) ==
self.find(y)

    def add(self,x):
        """
        添加新节点，初始化权重为1.0
        """
        if x not in self.father:
            self.father[x] = None
            self.value[x] = 1.0


class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float],
queries: List[List[str]]) -> List[float]:
        uf = UnionFind()
        for (a,b),val in zip(equations,values):
            uf.add(a)
            uf.add(b)
            uf.merge(a,b,val)

        res = [-1.0] * len(queries)

        for i,(a,b) in enumerate(queries):
            if uf.is_connected(a,b):
                res[i] = uf.value[a] / uf.value[b]
        return res
```

406、根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

```
输入：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出：[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释：
编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。
```

- **高分解答**

  1、先排序，在插队动画演示

  首先对数对进行排序，按照数对的元素 1 降序排序，按照数对的元素 2 升序排序。原因是，按照元素 1 进行降序排序，对于每个元素，在其之前的元素的个数，就是大于等于他的元素的数量，而按照第二个元素正向排序，我们希望 k 大的尽量在后面，减少插入操作的次数。

```python
class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        res = []
        people = sorted(people, key = lambda x: (-x[0], x[1]))
        for p in people:
            if len(res) <= p[1]:
                res.append(p)
            elif len(res) > p[1]:
                res.insert(p[1], p)      #将指定对象插入列表的指定位置
        return res
```

416、分割等和子集

给定一个**只包含正整数**的**非空**数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

```
输入: [1, 5, 11, 5]
输出: true
解释: 数组可以分割成 [1, 5, 5] 和 [11].
```

```
输入: [1, 2, 3, 5]
输出: false
解释: 数组不能分割成两个元素和相等的子集.
```

- **第一想法**

  1、动态规划

  首先判断sum，为奇数返回False，偶数sum/2就是目标值，找到一个子集和等于sum/2就好了

- **高分解答**

  1、动态规划之01背包

  本题要求把数组分成两个等和的子集，相当于找到一个子集，其和为sum/2，这个sum/2就是target

  1. 特例：如果sum为奇数，那一定找不到符合要求的子集，返回False。
  2. dp[j]含义：有没有和为j的子集，有为True，没有为False。
  3. 初始化dp数组：长度为target + 1，用于存储子集的和从0到target是否可能取到的情况。比如和为0一定可以取到（也就是子集为空），那么dp[0] = True。
  4. 接下来开始遍历nums数组，对遍历到的数nums[i]有两种操作，一个是选择这个数，一个是不选择这个数。
     - 不选择这个数：dp不变
     - 选择这个数：dp中已为True的情况再加上nums[i]也为True。比如dp[0]已经为True，那么dp[0 + nums[i]]也是True
  5. 在做出选择之前，我们先逆序遍历子集的和从nums[i]到target的所有情况，判断当前数加入后，dp数组中哪些和的情况可以从False变成True。
     （为什么要逆序，是因为dp后面的和的情况是从前面的情况转移过来的，如果前面的情况因为当前nums[i]的加入变为了True，比如dp[0 + nums[i]]变成了True，那么因为一个数只能用一次，dp[0 + nums[i] + nums[i]]不可以从dp[0 + nums[i]]转移过来。如果非要正序遍历，必须要多一个数组用于存储之前的情况。而逆序遍历可以省掉这个数组）
     - dp[j] = dp[j] or dp[j - nums[i]]
     - 这行代码的意思是说，如果不选择当前数，那么和为j的情况保持不变，dp[j]仍然是dp[j]，原来是True就还是True，原来是False也还是False；
     - 如果选择当前数，那么如果j - nums[i]这种情况是True的话和为j的情况也会是True。比如和为0一定为True，只要 j - nums[i] == 0，那么dp[j]就变成了True。
     - dp[j]和dp[j-nums[i]]只要有一个为True，dp[j]就变成True，因此用or连接两者。
  6. 最后就看dp[-1]是不是True，也就是dp[target]是不是True

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        sumAll = sum(nums)
        if sumAll % 2:
            return False
        target = sumAll // 2

        dp = [False] * (target + 1)
        dp[0] = True

        for i in range(len(nums)):
            for j in range(target, nums[i] - 1, -1):
                dp[j] = dp[j] or dp[j - nums[i]]
        return dp[-1]
```

2、暴力枚举

```
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        target, res = divmod(sum(nums), 2)
        if res: return False

        sums = {0}  # 存所有子集和
        for num in nums:
            tmp = {s + num for s in sums}  # 遍历已有的子集和，加上num就可以得到更多的子集和
            if target in tmp: return True  # 如果中途达成目标了，就可以返回True了
            sums |= tmp  # 加入这一轮得到的子集和

        return False
```

3、DFS——超时间

```
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        def dfs(n, target, start):
            if target == 0:
                return True
            if target < 0:
                return False
            for i in range(start, n):
                if dfs(n, target-nums[i], i+1):
                    return True
            return False

        numSum = sum(nums)
        if numSum % 2 != 0:
            return False
        n = len(nums)
        half = sum(nums) // 2
        nums.sort(reverse=True)
        return dfs(n, half, 0)
```

437、路径总和III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

   10
  /  \
 5   -3
```

```
/ \    \
3   2   11
/ \   \
3  -2   1

返回 3。和等于 8 的路径有:

1.  5 -> 3
2.  5 -> 2 -> 1
3.  -3 -> 11
```

- **高分解答**

  1、递归，记录当前结果

  `sumlist[]`记录当前路径上的和，在如下样例中：

  - 当DFS刚走到`2`时，此时`sumlist[]`从根节点`10`到`2`的变化过程为：
    - ```
      10
      15 5
      17 7 2
      ```
  - 当DFS继续走到`1`时，此时`sumlist[]`从节点`2`到`1`的变化为：
    - 18 8 3 1
  - 因此，只需计算每一步中，sum在数组sumlist中出现的次数，然后与每一轮递归的结果相加即可

```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> int:
        def dfs(root, sumlist):
            if root is None: return 0
            sumlist = [num + root.val for num in sumlist] + [root.val]
            return sumlist.count(sum) + dfs(root.left, sumlist) +
dfs(root.right, sumlist)
        return dfs(root, [])
```

  2、栈解法，遍历二叉树

  利用栈对二叉树进行遍历，用一个额外的数组保存所有二叉树路径的节点和，判断每个保存节点和的数组里有多少个和sum相等的数即可。

```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> int:
        if not root:
            return 0

        stack = [(root,[root.val])]
        res = 0

        while stack:
            node,temp = stack.pop()
            res += temp.count(sum)
            temp += [0]
            if node.left:
                arr = [i+node.left.val for i in temp]
```
```

```
                stack.append((node.left,arr))

            if node.right:
                arr = [i+node.right.val for i in temp]
                stack.append((node.right,arr))

        return res
```

3、动态规划

- 如果是一个线性的数组，求解存在多少个子序列之和为tartget，这样一个问题是不是很容易想到使用动态规划？
  - 设d[i]为前i个数中包含第i个元素的所有子序列的和那么其状态转移方程：`dp[i] = [list[i], list[i] + dp[i - 1][0]...list[i] + dp[i - 1][n - 1]]`。
  - 最后只需要统计dp中每个状态包含多少个目标值。
- 显然，两者的唯一区别在于数据结构的不同，一个是线性表，一个是二叉树，线性表依赖的是前驱，可以通过索引比较，那么二叉树以来的是左子树和右子树。且对于线性表，底表示从端点开始，这里的二叉树是从叶子节点开始。
  - 所以二叉树如何自底向上呢：后序遍历。
  - 因此状态方程：`dp[root] = [root.val, root.val + dp[i - 1][0]...root.val + dp[i - 1][n - 1]]`
- 至此，我们已经解决了动态规划的两个基本问题：
  - 从哪里开始
  - 状态以及如何转移

```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> int:
        dp = {}

        def search(root: TreeNode): # 后续遍历
            if root:
                search(root.left)
                search(root.right)
                a = b = []
                if root.left:
                    a = dp[root.left]
                if root.right:
                    b = dp[root.right]
                temp = [root.val]
                for t in a:
                    temp.append(root.val + t)
                for t in b:
                    temp.append(root.val + t)
                dp[root] = temp
        search(root)
        k = 0 # 统计目标值
        for r, val in dp.items():
            for t in val:
                if t == sum:
                    k += 1
        return k
```

438、找到字符串中所有字母异位词

给定一个字符串 **s** 和一个非空字符串 **p**，找到 **s** 中所有是 **p** 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 **s** 和 **p** 的长度都不超过 20100。

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

> 输入:
> s: "cbaebabacd" p: "abc"
>
> 输出:
> [0, 6]
>
> 解释:
> 起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。
> 起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

- **第一想法**

  1、滑动窗口，

  从0到len(s)-len(p)，选择一个数组保存s[i:i+len(p)]，与p相比，存在字母删除一个，，最后等于0就保存

  参考解答中排序想法做对照，，，（时间5%，内存99.5%）

```python
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        n = len(p)
        p = "".join(sorted(p))
        res = []
        for i in range(len(s) - n + 1):
            if "".join(sorted(s[i:i + n])) == p:
                res.append(i)
        return res
```

- **高分解答**

  1、滑动窗口+数组

  1. 因为字符串中的字符全是小写字母，可以用长度为26的数组记录字母出现的次数
  2. 设n = len(s), m = len(p)。记录p字符串的字母频次p_cnt，和s字符串前m个字母频次s_cnt
  3. 若p_cnt和s_cnt相等，则找到第一个异位词索引 0
  4. 继续遍历s字符串索引为[m, n)的字母，在s_cnt中每次增加一个新字母，去除一个旧字母
  5. 判断p_cnt和s_cnt是否相等，相等则在返回值res中新增异位词索引 i - m + 1

```python
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        n, m, res = len(s), len(p), []
        if n < m: return res
        p_cnt = [0] * 26
        s_cnt = [0] * 26
        for i in range(m):
```

```
            p_cnt[ord(p[i]) - ord('a')] += 1
            s_cnt[ord(s[i]) - ord('a')] += 1
        if s_cnt == p_cnt:
            res.append(0)

        for i in range(m, n):
            s_cnt[ord(s[i - m]) - ord('a')] -= 1
            s_cnt[ord(s[i]) - ord('a')] += 1
            if s_cnt == p_cnt:
                res.append(i - m + 1)
        return res
```

2、滑动窗口+双指针

1. 定义滑动窗口的左右两个指针left，right
2. right一步一步向右走遍历s字符串
3. right当前遍历到的字符加入s_cnt后不满足p_cnt的字符数量要求，将滑动窗口左侧字符不断弹出，也就是left不断右移，直到符合要求为止。
4. 当滑动窗口的长度等于p的长度时，这时的s子字符串就是p的异位词。
5. 其中，left和right表示滑动窗口在字符串s中的索引，cur_left和cur_right表示字符串s中索引为left和right的字符在数组中的索引

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        n, m, res = len(s), len(p), []
        if n < m: return res
        p_cnt = [0] * 26
        s_cnt = [0] * 26

        for i in range(m):
            p_cnt[ord(p[i]) - ord('a')] += 1

        left = 0
        for right in range(n):
            cur_right = ord(s[right]) - ord('a')
            s_cnt[cur_right] += 1
            while s_cnt[cur_right] > p_cnt[cur_right]:
                cur_left = ord(s[left]) - ord('a')
                s_cnt[cur_left] -= 1
                left += 1
            if right - left + 1 == m:
                res.append(left)
        return res
```

3、双指针+字典

1. 维护一个ascii_lowercase为key的全零字典
2. 根据p生成待匹配的字典信息dict_p
3. 同理创建针对s的ascii_lowercase为key的全零字典
4. 创建双指针，right从s[0]出发每次添加至tmp字典
5. 当左右指针差距小于len(p)时，left指针不动
6. 当左右指针差距等于len(p)时开始正式的对比操作，并每次匹配后left+=1
7. 如果tmp等于dict_p,则将left指针添加入ret。
8. 如果不匹配，删除掉左指针对应的字母。
9. right+=1
10. 当right指针走至s末尾结束判断。

```python
from string import ascii_lowercase
class Solution:
    def findAnagrams(self, s, p):
        ret = []
        k_dict = {}.fromkeys(ascii_lowercase, 0)
        for i in p:
            k_dict[i] += 1
        tmp = {}.fromkeys(ascii_lowercase, 0)
        left = right = 0
        while right < len(s):
            tmp[s[right]] += 1
            if tmp == k_dict:
                ret.append(left)
            if right - left + 1 == len(p):
                tmp[s[left]] -= 1
                left += 1
            right += 1
        return ret
```

494、目标和

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

> 输入：nums: [1, 1, 1, 1, 1], S: 3
> 输出：5
> 解释：
> -1+1+1+1+1 = 3
> +1-1+1+1+1 = 3
> +1+1-1+1+1 = 3
> +1+1+1-1+1 = 3
> +1+1+1+1-1 = 3
> 一共有5种方法让最终目标和为3。

- **高分解答**

  1、DFS

  1. 设置一个哈希表（字典），键是一个元祖，元祖第一位是目前的和，第二位是目前的位数。值是这个元祖推导到最后能有多少个解。

2. 例如d[(5，4)] = 1 代表读到4位的时候，正好有一个解符合条件（那么在这个例子中符合条件的S就是5），然后倒导d([5，3]) = 2 .....(在这种情况下，第4位是0，总共就4位)
3. 初始化节点为(0,0)，代表已经读了0位，现在和为0
4. 开始深度优先搜索，当i比位数小，说明可以深入。为了避免重复运算，要看当前节点是否在d里已经出现过。
5. 每次深入的结果，就是d[(i, cur)] = dfs(cur + nums[i], i + 1, d) + dfs(cur - nums[i], i + 1, d)。意思就是当前节点推导到最后有多少个可能性呢？这个节点再读取一位，要么是加上这一位，要么是减掉这一位，所以这个节点的可能性就是对加上下一位的可能性与减掉下一位的可能性之和。
6. 当深入到最后一位时，再深入就超了位数限制了，此时可以直接判断这个节点的和（即元祖的第一位）是否等于需要的S。是了为1，否则为0。因为dfs可能遍历到重复节点，所以return一行写作

d.get((cur，i), int(cur == S))。如果是重复节点直接返回字典里对应值就完事儿

```
  class Solution:
      def findTargetSumWays(self, nums: List[int], S: int) -> int:
          d = {}
          def dfs(cur, i, d):
              if i < len(nums) and (cur, i) not in d: # 搜索周围节点
                  d[(cur, i)] = dfs(cur + nums[i], i + 1, d) + dfs(cur -
nums[i],i + 1, d)
              return d.get((cur, i), int(cur == S))
          return dfs(0, 0, d)
```

2、01背包

我们可以将原问题转化为： 找到nums一个正子集和一个负子集，使得总和等于target，统计这种可能性的总数。

我们假设P是正子集，N是负子集。让我们看看如何将其转换为子集求和问题：

```
                    sum(P) - sum(N) = target
                    （两边同时加上sum(P)+sum(N)）
  sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)
              (因为 sum(P) + sum(N) = sum(nums))
                         2 * sum(P) = target + sum(nums)
```

因此，原来的问题已转化为一个求子集的和问题： 找到`nums`的一个子集 P，使得

`sum(P)=(target+sum(nums))/2`

根据公式，若`target + sum(nums)`不是偶数，就不存在答案，即返回0个可能解。

解决01背包问题使用的是`动态规划`的思想。`dp`的第`x`项，代表组合成数字x有多少方法。

怎么更新dp数组呢?

- 遍历nums，遍历的数记作num
    - 再逆序遍历从P到num，遍历的数记作j
        - 更新dp[j] = dp[j - num] + dp[j]
- 这样遍历的含义是，对每一个在nums数组中的数num而言，dp在从num到P的这些区间里，都可以加上一个num，来到达想要达成的P。
- 举例来说，对于数组[1,2,3,4,5]，想要康康几种方法能组合成4,那么设置dp[0]到dp[4]的数组
- 假如选择了数字2,那么dp[2:5]（也就是2到4）都可以通过加上数字2有所改变，而dp[0:2]（也就是0到1）加上这个2很明显就超了，就不管它。
- 以前没有考虑过数字2,考虑了会怎么样呢？就要更新dp[2:5]，比如说当我们在更新dp[3]的时候，就相当于dp[3] = dp[3] + dp[1],即本来有多少种方法，加上去掉了2以后有多少种方法。因为以前没有考虑过2,现在知道，只要整到了1,就一定可以整到3。

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], S: int) -> int:
        if sum(nums) < S or (sum(nums) + S) % 2 == 1: return 0
        P = (sum(nums) + S) // 2
        dp = [1] + [0 for _ in range(P)]
        for num in nums:
            for j in range(P,num-1,-1):dp[j] += dp[j - num]
        return dp[P]
```

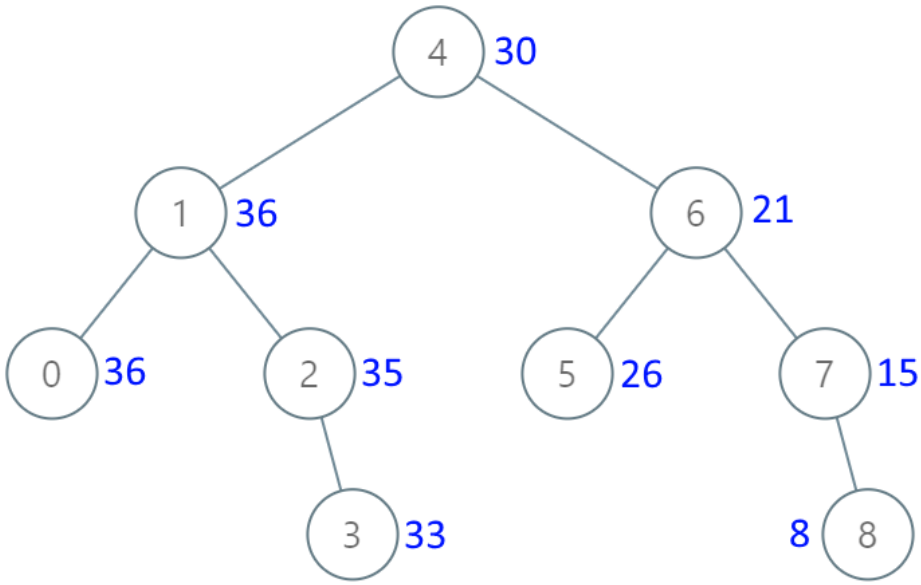
3、动态规划

- 设置一个哈希表（字典），键是一个元祖，元祖第一位是目前的和，第二位是目前的位数。值是这个元祖推导到最后能有多少个解。
- 例如d[(4,5)] = 1 代表读到4位的时候，正好有一个解符合条件（那么在这个例子中符合条件的S就是5），然后倒导d([3,5]) = 2 ……(在这种情况下，第4位是0，总共就4位)
- 因为符号要么全正，要么全负，所以元祖第二位的取值范围是 -sum(nums) ~ sum(nums)
- 状态转移公式：
  `dp[(i,j)] = dp.get((i - 1, j - nums[i]), 0) + dp.get((i - 1, j + nums[i]), 0)`
  其实本质上就是没有使用递归的`dfs`。

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], S: int) -> int:
        length, dp = len(nums), {(0,0): 1}
        for i in range(1, length+1):
            for j in range(-sum(nums), sum(nums) + 1):
                dp[(i,j)] = dp.get((i - 1, j - nums[i-1]), 0) + dp.get((i - 1, j + nums[i-1]), 0)
        return dp.get((length, S), 0)
```

538、把二叉搜索树转换为累加树

给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于或等于 node.val 的值之和。




```
输入：[4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出：[30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

```
输入：root = [0,null,1]
输出：[1,null,1]
```

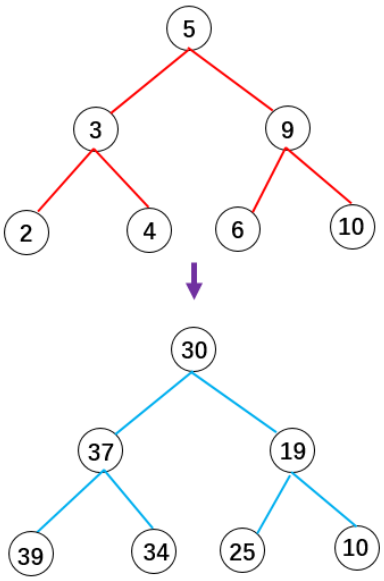
- **第一想法**

  1、反向中序遍历（右中左，累加）

```python
class Solution:
    def convertBST(self, root: TreeNode) -> TreeNode:
        def dfs(root):
            nonlocal tmp
            if not root:
                return
            dfs(root.right)
            root.val += tmp
            tmp = root.val
            dfs(root.left)

        tmp = 0
        dfs(root)
        return root
```

- **高分解答**

  1、非递归解法



知识点：二叉搜索树的中序遍历是所有节点从小到大的排序
因此中序遍历的结果为：

2 3 4 5 6 9 10

按照题目要求，这些个节点的值要变成比各自值大的所有的值的和，即

39 37 34 30 25 19 10

反向看更有思路：

10 19 25 30 34 37 39

也就是：

10 9 6 5 4 3 2 的前缀和。

19 25 30 34 37 39

因此我们的任务可以明确：反向中序遍历的过程中计算遍历数组的前缀和，然后更新访问节点的值。最后返回root即可。
问题转化为中序遍历，想怎么做就怎么做。

```python
class Solution:
    def convertBST(self, root: TreeNode) -> TreeNode:
        stack = [(1, root)]
        res = []
        dp_sum = 0
        while stack:
            color, curNode = stack.pop()
            if not curNode: continue
            if color == 1:
                stack.append((1, curNode.left))
                stack.append((0, curNode))
                stack.append((1, curNode.right))
            else:
                dp_sum += curNode.val
                curNode.val = dp_sum
        return root
```

560、和为K的子数组

给定一个整数数组和一个整数 **k**，你需要找到该数组中和为 **k** 的连续的子数组的个数。

> 输入:nums = [1,1,1], k = 2
> 输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

- **第一想法**

  1、暴力搜索（滑动窗口，与k相等：res+=1）——超时间

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

        return res
```

- **高分解答**

  1、前缀和

  - num_times存储的是某一个前缀和出现的次数，但是由于我们在动态遍历 nums 并对 num_times 这个字典的值进行更新，所以每到一个新位置我们看到的都是到当前这个新位置为止，前面的某个前缀和出现的次数。
  - 比如我们到某一个位置 i 得到前缀和为 9，也就说从 0 位置到 i 位置的所有数字的和为 9， 如果目标 k 为 3，那么我们只需要找到当前状态下，前面出现了几次 6，就知道以 nums[i] 结尾的和为 3 的连续子数组的个数有多少个。

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        # num_times 存储某“前缀和”出现的次数，这里用collections.defaultdict来定义它
        # 如果某前缀不在此字典中，那么它对应的次数为0
        num_times = collections.defaultdict(int)
        num_times[0] = 1  # 先给定一个初始值，代表前缀和为0的出现了一次
        cur_sum = 0  # 记录到当前位置的前缀和
        res = 0
        for i in range(len(nums)):
            cur_sum += nums[i]  # 计算当前前缀和
            if cur_sum - k in num_times:  # 如果前缀和减去目标值k所得到的值在字典中出现，即当前位置前缀和减去之前某一位的前缀和等于目标值
                res += num_times[cur_sum - k]
            # 下面一句实际上对应两种情况，一种是某cur_sum之前出现过（直接在原来出现的次数上+1即可），
            # 另一种是某cur_sum没出现过（理论上应该设为1，但是因为此处用defaultdict存储，如果cur_sum这个key不存在将返回默认的int，也就是0）
            # 返回0加上1和直接将其置为1是一样的效果。所以这里统一用一句话包含上述两种情况
            num_times[cur_sum] += 1
        return res
```

581、最短无序连续子数组

给你一个整数数组 nums ，你需要找出一个 连续子数组 ，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 最短 子数组，并输出它的长度。

> 输入：nums = [2,6,4,8,10,9,15]
> 输出：5
> 解释：你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。

> 输入：nums = [1,2,3,4]
> 输出：0

- **第一想法**

  1、与排序数组对比，从前和后分别对比，比较不同数字出现的位置

```python
class Solution:
    def findUnsortedSubarray(self, nums: List[int]) -> int:
        nums_s=sorted(nums)
        left,rigth=0,0
        for i in range(len(nums)):
            if nums_s[i]!=nums[i]:
                left=i
                break
        for i in range(len(nums)-1,-1,-1):
            if nums_s[i]!=nums[i]:
                rigth=i
                break

        if rigth==0:        #防止出现没有无序子数组的情况
            return 0
        else:
            return rigth-left+1
```

- **高分解答**

  1、寻找左右边界

  - 从左到右找最大值，找出end边界，最后一个比最大值小的即为end
  - 从右到左找最小值，找出start边界，最后一个比最小值大的即为start

```python
class Solution:
    def findUnsortedSubarray(self, nums: List[int]) -> int:
        start=-1
        end=-2
        Amax=nums[0]
        Amin=nums[len(nums)-1]
        for i in range(len(nums)):
            if nums[i]<Amax:
                end=i
            else:
                Amax=nums[i]
            if nums[len(nums)-i-1]>Amin:
                start=len(nums)-i-1
            else:
                Amin=nums[len(nums)-i-1]
        return end-start+1
```

621、任务调度器

给你一个用字符数组 tasks 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间，CPU 可以完成一个任务，或者处于待命状态。

然而，两个 相同种类 的任务之间必须有长度为整数 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的 最短时间 。

> 输入：tasks = ["A","A","A","B","B","B"], n = 2
> 输出：8
> 解释：A -> B -> (待命) -> A -> B -> (待命) -> A -> B
> 在本示例中，两个相同类型任务之间必须间隔长度为 n = 2 的冷却时间，而执行一个任务只需要一个单位时间，所以中间出现了（待命）状态。

> 输入：tasks = ["A","A","A","B","B","B"], n = 0
> 输出：6
> 解释：在这种情况下，任何大小为 6 的排列都可以满足要求，因为 n = 0
> ["A","A","A","B","B","B"]
> ["A","B","A","B","A","B"]
> ["B","B","B","A","A","A"]
> ...
> 诸如此类

> 输入：tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2
> 输出：16
> 解释：一种可能的解决方案是：
> A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> (待命) -> (待命) -> A -> (待命) -> (待命) -> A

- **高分解答**

  1、装袋

  例：AAAA BBBB CCC DD EE F　　n = 2

  1. 首先安排数量最多的任务 A 如下：A--A--A--A-

     记任务 A 的数量为 max_count。`total = (max_count - 1) * (n + 1) + 1`。将连续的 - 视为一个口袋，可以看到共有 4 个口袋，大小依次为 2, 2, 2, 1。

  2. 接下来安排数量同为 max_count 的任务，按行将任务依次放入这些口袋中。即，将任务 B 放入口袋 0, 1, 2, 3。AB-AB-AB-AB

     total 加上最后一个口袋中的任务数，即 `total += 1`。

  3. 剩余的每类任务的数量至多为 max_count - 1 。按行将剩余任务依次（循环）放入前 max_count - 1 个口袋中。即，将任务 C 放入口袋 0, 1, 2; 将任务 D 放入口袋 0, 1；将任务 E 放入口袋 2, 0；将任务 F 放入口袋 1。易知不会有两个同类任务在同一口袋中，而不同口袋的同类任务间的距离大于等于 n + 1。

  4. 若前 max_count - 1 个口袋未全部装满，则总位置数为 total；若皆满溢，则总位置数为总任务数。

     易知 total 和总任务数是总位置数的两个下限,故该算法是最优的。

```
    def leastInterval(self, tasks, n):
        count = [0] * 26
        for ch in tasks:
            count[ord(ch) - ord('A')] += 1

        count_max = max(count)
        total = (count_max - 1) * (n + 1)

        for _ in count:
            if _ == count_max:
                total += 1

        return max(total, len(tasks))
```

647、回文子串

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

> 输入："abc"
> 输出：3
> 解释：三个回文子串："a", "b", "c"

> 输入："aaa"
> 输出：6
> 解释：6个回文子串："a", "a", "a", "aa", "aa", "aaa"

- **第一想法**

  1、滑动窗口，判断是否为回文子串，是的话res+=1

- **高分解答**

  1、动态规划

  - dp[i][j] 表示 s[i...j]s[i...j] 是否为回文子串
  - 初始化
    `if j-i==0: dp[i][j]=1`
    `if j-i==1: dp[i][j] = s[i]==s[j]`
  - 迭代更新
    `if j-i>=2: dp[i][j]=dp[i+1][j-1] and s[i]==s[j]`
    重点：根据递推公式, 右侧的横纵坐标之差更小, 所以遍历的时候可以按照 k=j-ik=j−i 从小到大遍历
  - 返回结果
    回文子串数目 ans = dp 中等于1 的元素之和

```
def countSubstrings(s):
    ans = 0
    n = len(s)
```

```
    dp = [[False]*n for _ in range(n)]
    ans = 0
    for k in range(n):
        # i+k<n => i<n-k
        for i in range(n-k):
            j=i+k
            if k==0:
                dp[i][j] = True
            elif k==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]: ans+=1
    return ans
```

2、中心扩展法

外层循环为遍历所有的字符，内层循环为以所遍历的字符向两边扩展，如果满足要求就在总数上加1，不满足要求则停止扩展

```
class Solution:
    def countSubstrings(self, s: str) -> int:
        len_s=sum_s=len(s)
        for i in range(len_s):
            left,right=i-1,i+1
            while left>-1 and right<len_s and s[left]==s[right]:
                sum_s+=1
                left-=1
                right+=1
            left,right=i,i+1
            while left>-1 and right<len_s and s[left]==s[right]:
                sum_s+=1
                left-=1
                right+=1
        return sum_s
```

739、每日温度

请根据每日 `气温` 列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

> 给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，
> 你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

- **第一想法**

  1、循环遍历，，，找到第一个比当天高的温度，保存下来——超时间

```python
class Solution:
    def dailyTemperatures(self, T: List[int]) -> List[int]:
        res=[0]*len(T)
        for i in range(len(T)):
            for j in range(i,len(T)):
                if T[i]<T[j]:
                    res[i]=j-i
                    break

        return res
```

- **高分解答**

  1、单调栈

  碰到比栈顶元素小的入栈，如果比栈顶大则一直出栈直到栈顶比之大或者空。
  特殊情况是结束为止栈非空

```python
class Solution:
    def dailyTemperatures(self, T: List[int]) -> List[int]:
        stack = []
        out = [0 for _ in range(len(T))]
        for i in range(len(T)):
            while stack and T[stack[-1]]<T[i]:
                j=stack.pop()
                out[j] = i-j
            stack.append(i)
        return out
```

  2、 反向遍历

```python
class Solution:
    def dailyTemperatures(self, T: List[int]) -> List[int]:
        d = {}
        ans = [0] * len(T)
        for i in range(len(T) - 1, -1, -1): # 从后向前遍历
            d[T[i]] = i          # 记录当前问题的最小下标
            tmp = [d[t] - i for t in range(T[i] + 1, 101) if t in d]
            ans[i] = (min(tmp) if tmp else 0)
        return ans
```

4、寻找两个正序数组的中位数

给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数** 。

> 输入：nums1 = [1,3], nums2 = [2]
> 输出：2.00000
> 解释：合并数组 = [1,2,3] ，中位数 2

> 输入：nums1 = [1,2], nums2 = [3,4]
> 输出：2.50000
> 解释：合并数组 = [1,2,3,4] ，中位数 (2 + 3) / 2 = 2.5

- **第一想法**

  1、合并数组，输出中位数

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) ->
float:
        num=[]
        while nums1 and nums2:
            if nums1[0]<nums2[0]:
                num.append(nums1.pop(0))
            else:
                num.append(nums2.pop(0))

        if nums1:num+=nums1
        else:num+=nums2

        n=len(num)//2
        if len(num)%2:
            return num[n]
        else:
            return (num[n]+num[n-1])/2
```

  好吧，可以直接合并然后排序

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) ->
float:
        nums3 = sorted(nums1 + nums2)
        l = len(nums3)
        if l % 2 == 0: #even length
            return (nums3[l//2-1] + nums3[l//2]) / 2
        else:  #odd length
            return nums3[l//2]
```

- **高分解答**

  1、寻找第K个数

  - 对于两个有序数组，我们要找第k小的数
  - 由于时间复制度要求是log，所以自然的想法就是对两个数组每次切一半。
    好，假设我们取两个数组k/2位置上的数(这里暂时不考虑上溢)
  - 如果nums1[k/2]>=nums2[k/2]，这意味着：
  - nums2数组的左半边都不需要考虑了，因为肯定会比第k小的数要来得小。

    所以我们可以切掉nums2数组的一半，如此递归，每次都能切走一半

```
def helper(nums1,nums2,k):
    if(len(nums1) <len(nums2) ):
        nums1, nums2 = nums2 , nums1 #保持nums1比较长
    if(len(nums2)==0):
        return nums1[k-1] # 短数组空，直接返回
    if(k==1):
        return min(nums1[0],nums2[0])  #找最小数，比较数组首位
    t = min(k//2,len(nums2)) # 保证不上溢
    if( nums1[t-1]>=nums2[t-1] ):
        return helper(nums1 , nums2[t:],k-t)
    else:
        return helper(nums1[t:],nums2,k-t)
```

为了处理奇偶时候中位数不同的计算方法，
这里可以采用一个小技巧：

- 令 k1 = (len(nums1) + len(nums2) + 1) // 2
  令 k2 = (len(nums1) + len(nums2) + 2) // 2
- 对于偶数情况，k1对应中间左边，k2对应中间右边
  对于奇数情况，k1，k2都对应中间
- 所以我们得到了获得中位数的统一方法：(helper(k1)+helper(k2))/2
- 缺点是：用了两倍的计算量；优点是：代码统一、清晰。

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) ->
float:
        k1 = ( len(nums1) + len(nums2) + 1 ) // 2
        k2 = ( len(nums1) + len(nums2) + 2 ) // 2
        def helper(nums1,nums2,k): #本质上是找第k小的数
            if(len(nums1) <len(nums2) ):
                nums1, nums2 = nums2 , nums1 #保持nums1比较长
            if(len(nums2)==0):
                return nums1[k-1] # 短数组空，直接返回
            if(k==1):
                return min(nums1[0],nums2[0])  #找最小数，比较数组首位
            t = min(k//2,len(nums2)) # 保证不上溢
            if( nums1[t-1]>=nums2[t-1] ):
                return helper(nums1 , nums2[t:],k-t)
            else:
                return helper(nums1[t:],nums2,k-t)
        return ( helper(nums1,nums2,k1) + helper(nums1,nums2,k2) ) /2
```

2、 二分查找

```
class Solution:
    #这题很自然地想到归并排序，再取中间数，但是是nlogn的复杂度，题目要求logn
    #所以要用二分法来巧妙地进一步降低时间复杂度
    #思想就是利用总体中位数的性质和左右中位数之间的关系来把所有的数先分成两堆，然后再在两堆
的边界返回答案
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) ->
float:
        m = len(nums1)
        n = len(nums2)
        # 让nums2成为更长的那一个数组
        if m>n:
            nums1,nums2,m,n = nums2,nums1,n,m
```

```
        # 如果两个都为空的异常处理
        if n == 0:
            raise ValueError

        # nums1中index在imid左边的都被分到左堆，nums2中jmid左边的都被分到左堆
        imin,imax = 0,m

        # 二分答案
        while(imin<=imax):
            imid = imin + (imax-imin)//2
            # 左堆最大的只有可能是nums1[imid-1],nums2[jmid-1]
            # 右堆最小只有可能是nums1[imid],nums2[jmid]
            # 让左右堆大致相等需要满足的条件是imid+jmid = m-imid+n-jmid 即 jmid = (m+n-2imid)//2
            # 为什么是大致呢？因为有总数为奇数的情况，这里用向下取整数操作，所以如果是奇数，右堆会多1
            jmid = (m+n-2*imid)//2

            # 前面的判断条件只是为了保证不会index out of range
            if(imid>0 and nums1[imid-1] > nums2[jmid]):
                # imid太大了，这是里精确查找，不是左闭右开，而是双闭区间，所以直接移动一位
                imax = imid-1
            elif(imid<m and nums2[jmid-1] > nums1[imid]):
                imin = imid+1
            # 满足条件
            else:
                # 边界情况处理，都是为了不out of index
                # 依次得到左堆最大和右堆最小
                if(imid == m): minright = nums2[jmid]
                elif(jmid == n): minright = nums1[imid]
                else:
                    minright = min(nums1[imid],nums2[jmid])

                if(imid == 0): maxleft = nums2[jmid-1]
                elif(jmid == 0): maxleft = nums1[imid-1]
                else:
                    maxleft = max(nums1[imid-1],nums2[jmid-1])

                # 前面也提过，因为取中间的时候用的是向下取整，所以如果总数是奇数的话，
                # 应该是右边个数多一些，边界的minright就是中位数
                if((m+n)%2) == 1:
                    return minright

                # 否则我们在两个值中间做个平均
                return (maxleft + minright)/2
```

10、正则表达式匹配

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

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

所谓匹配，是要涵盖 **整个** 字符串 `s`的，而不是部分字符串。

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

> 输入：s = "aa" p = "a*"
> 输出：true
> 解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

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

> 输入：s = "aab" p = "c*a*b"
> 输出：true
> 解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因此可以匹配字符串 "aab"。

- **高分解答**

  1、回溯

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

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

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

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

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

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

2、动态规划

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

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

1. `s[i] == p[j] or p[j] == '.'`：比如 abb 和 abb，或者 abb 和 ab.，很容易得到 `dp[i][j] = dp[i-1][j-1] = True`。因为 ab 和 ab 是匹配的，如果后面分别加一个 b，或者 s 加一个 b 而 p 加一个 .，仍然是匹配的。
2. p[j] == '*'：当 p[j] 为星号时，由于星号与前面的字符相关，因此我们比较星号前面的字符 p[j-1] 和 s[i] 的关系。根据星号前面的字符与 s[i] 是否相等，又可分为以下两种情况：
   - p[j-1] != s[i]：如果星号前一个字符匹配不上，星号匹配了 0 次，应忽略这两个字符，看 p[j-2] 和 s[i] 是否匹配。 这时 `dp[i][j] = dp[i][j-2]`。
   - p[j-1] == s[i] or p[j-1] == '.':星号前面的字符可以与 s[i] 匹配，这种情况下，星号可能匹配了前面的字符的 0 个，也可能匹配了前面字符的多个，当匹配 0 个时，如 ab 和 abb，*或者 ab 和 ab.*，这时我们需要去掉 p 中的 b* 或 .* 后进行比较，即 `dp[i][j] = dp[i][j-2]`；当匹配多个时，如 abbb 和 ab*，或者 abbb 和 `a.*`，我们需要将 s[i] 前面的与 p 重新比较，即 `dp[i][j] = dp[i-1][j]`
3. 其他情况：以上两种情况把能匹配的都考虑全面了，所以其他情况为不匹配，即 dp[i][j] = False

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

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

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

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

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

```
                        dp[r][c] = dp[r][c - 2]
                else:
                    dp[r][c] = False
        return dp[m - 1][n - 1]
```

23、合并k个升序链表

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

> 输入：lists = [[1,4,5],[1,3,4],[2,6]]
> 输出：[1,1,2,3,4,4,5,6]
> 解释：链表数组如下：
> [
> 1->4->5,
> 1->3->4,
> 2->6
>]
> 将它们合并到一个有序链表中得到。
> 1->1->2->3->4->4->5->6

- **高分解答**

  1、 暴力法

  直接将所有的元素取出，然后排个序，再重组就达到了目的

```
class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if not lists or len(lists) == 0:
            return None
        import heapq
        all_vals = []
        for l in lists:
            while l:
                all_vals.append(l.val)
                l = l.next
        all_vals.sort()
        dummy = ListNode(None)
        cur = dummy
        for i in all_vals:
            temp_node = ListNode(i)
            cur.next = temp_node
            cur = temp_node

        return dummy.next
```

  2、堆

  维护一个小顶堆，首先将所有链表的第一个节点加入小顶堆，每次从小顶堆中弹出一个节点node加入到最终结果的链表里，并将node的next加入到小顶堆中。

```
import heapq

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if not lists:
            return None
        dummy = ListNode()
        heap = []
        for i in range(len(lists)):
            if lists[i]:
                heapq.heappush(heap, (lists[i].val, i))
                lists[i] = lists[i].next
        cur = dummy
        while heap:
            val, idx = heapq.heappop(heap)
            cur.next = ListNode(val)
            cur = cur.next
            if lists[idx]:
                heapq.heappush(heap, (lists[idx].val, idx))
                lists[idx] = lists[idx].next
        return dummy.next
```

3、分治

归并排序：先两两合并mergeTwoLists，然后对已经经过一次两两合并的链表们再进行一次两两合并，一直合并到生成最终的升序链表为止

1. 特判（即递归终止条件）：如果待合并的链表个数为0返回空，个数为1返回lists[0]无需合并
2. 当待合并的链表个数大于等于2时，调用mergeTwoLists进行两两合并
3. 这里mergeTwoLists的两个参数采用递归mergeKLists的方式，也就是对已经合并好的lists前一半的链表和后一半的链表进行合并，如果没合并好，先分别合并了再最终合并

```
class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        n = len(lists)
        if n == 0: return None
        if n == 1: return lists[0]

        mid = n // 2
        return self.mergeTwoLists(self.mergeKLists(lists[:mid]),
self.mergeKLists(lists[mid:]))

    # 迭代写法
    def mergeTwoLists(self, node1, node2):
        dummy = cur = ListNode()
        while node1 and node2:
            if node1.val <= node2.val:
                cur.next = node1
                node1 = node1.next
            else:
                cur.next = node2
                node2 = node2.next
            cur = cur.next

        cur.next = node1 if node1 else node2
        return dummy.next

    # 递归写法
```

```
    def mergeTwoLists(self, node1, node2):
        if not node1 or not node2:
            return node1 if node1 else node2
        if node1.val <= node2.val:
            node1.next = self.mergeTwoLists(node1.next, node2)
            return node1
        else:
            node2.next = self.mergeTwoLists(node1, node2.next)
            return node2
```

32、最长有效括号

给你一个只包含 `'('` 和 `')'` 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

> 输入：s = "(()"
> 输出：2
> 解释：最长有效括号子串是 "()"

> 输入：s = ")()())"
> 输出：4
> 解释：最长有效括号子串是 "()()"

> 输入：s = ""
> 输出：0

- **第一想法**

  1、从最大窗开始搜索，判断是否为有效括号——超时间

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        def valid(A:str):
            bal=0
            for c in A:
                if c=='(':bal+=1
                else:bal-=1
                if bal<0:return False
            return bal==0

        for k in range(len(s),1,-1):
            for i in range(len(s)-k+1):
                num=str(s[i:i+k])
                if valid(num):
                    return k

        return 0
```

- **高分解答**

  1、 动态规划

  动态规划，dp[i]表示以索引i为结尾的最长有效括号字串的长度。

- 如果s[i] == '('，则以i结尾的字符串不可能是有效括号，dp[i] = 0。
- 如果s[i] == ')'，有以下情况：
  - s[i-1] == '('，则 `dp[i] = dp[i-2] + 2`
  - s[i-1] == ')'，则 `dp[i] = dp[i-1] + 2 + dp[i - dp[i-1] - 2]`

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        if not s:
            return 0
        dp = [0] * len(s)
        for i in range(1, len(s)):
            if s[i] == ')':
                if i >= 2 and s[i-1] == '(':
                    dp[i] = dp[i-2] + 2
                else:
                    if i - dp[i-1] > 0 and s[i - dp[i-1] - 1] == '(':
                        dp[i] = dp[i-1] + 2 + dp[i - dp[i-1] - 2]
        return max(dp)
 #   ( ) ( ( ) ( ) )
 #  0 2 0 0 2 0 4 8
```

2、栈处理

从前往后遍历字符串

- 入栈条件为

1. 栈为空
2. 当前字符是'('
3. 栈顶符号位')',

   因为三种条件都没办法消去成对的括号。

- 计算结果：符合消去成对括号时，拿当前下标减去栈顶下标即可

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        if not s: return 0

        stack = []
        ans = 0
        for i in range(len(s)):
            # 入栈条件
            if not stack or s[i] == '(' or s[stack[-1]] == ')':
                stack.append(i)     # 下标入栈
            else:
                # 计算结果
                stack.pop()
                ans = max(ans, i - (stack[-1] if stack else -1))
        return ans
```

3、比较左右括号数量

左右括弧应该相等才是正确的，但是都会担心各种各样的情况，本思路巧妙的地方在于左遍历时只要“)”比“(”数量大就把左右计数都赋予0，右边反之。

```
class Solution:
```

```
    def longestValidParentheses(self, s: str) -> int:
        return max(self.left_to_right(s), self.right_to_left(s))

    def left_to_right(self, s):
        maxnum = 0
        left, right = 0, 0
        for i in range(len(s)):
            if s[i] == "(": left += 1
            else:
                right += 1
                if right > left: left, right = 0, 0
                if right == left: maxnum = max(maxnum, right*2)
        return maxnum

    def right_to_left(self, s):
        maxnum = 0
        left, right = 0, 0
        for i in range(len(s)-1, -1, -1):
            if s[i] == ")": right += 1
            else:
                left += 1
                if right < left: left, right = 0, 0
                if right == left: maxnum = max(maxnum, left*2)
        return maxnum
```

42、接雨水

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

> 输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]
> 输出：6
> 解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

> 输入：height = [4,2,0,3,2,5]
> 输出：9

- **高分解答**

  1、暴力搜索——超时间

  对于每个下标 i，我们计算它上方可以接多少雨水，然后将所有下标上的雨水加起来即可。

因此，我们遍历每个下标，寻找它左边和右边最高的柱子，判断是否可以接到水，将可接水的结果累加即可。

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = 0
        for i in range(len(height)):
            max_left, max_right = 0,0
            # 寻找 max_left
            for j in range(0,i):
                max_left = max(max_left,height[j])
            # 寻找 max_right
            for j in range(i,len(height)):
                max_right = max(max_right,height[j])
            if min(max_left,max_right) > height[i]:
                ans += min(max_left,max_right) - height[i]

        return ans
```

2、动态规划

我们发现在寻找每个下标的左边和右边最高的柱子时，会对柱子进行反复搜索导致复杂度降低，假如我们使用两个数组 maxleft 和 maxright，maxleft[i] 表示下标 i 左边最高柱子的高度,maxright[i] 表示下标 i 右边最高柱子的高度，很明显，我们只需要一趟遍历就可以得到结果。这样由于避免了重复计算，时间复杂度会降低到 O(N)。

```
class Solution:
    def trap(self, height: List[int]) -> int:
        # 边界条件
        if not height: return 0
        n = len(height)
        maxleft = [0] * n
        maxright = [0] * n
        ans = 0
        # 初始化
        maxleft[0] = height[0]
        maxright[n-1] = height[n-1]
        # 设置备忘录，分别存储左边和右边最高的柱子高度
        for i in range(1,n):
            maxleft[i] = max(height[i],maxleft[i-1])
        for j in range(n-2,-1,-1):
            maxright[j] = max(height[j],maxright[j+1])
        # 一趟遍历，比较每个位置可以存储多少水
        for i in range(n):
            if min(maxleft[i],maxright[i]) > height[i]:
                ans += min(maxleft[i],maxright[i]) - height[i]
        return ans
```

3、双指针

- 双指针法就是将上边的一个下标 `i`，变为两个下标 `left`，`right`，分别位于输入数组的两端。向中间移动时，边移动边计算。
- 除此之外，我们使用 `maxleft` 作为 `0...left` 的最大值，`maxright` 作为 `right...结尾` 的最大值，动画演示

```
class Solution:
```

```python
    def trap(self, height: List[int]) -> int:
        # 边界条件
        if not height: return 0
        n = len(height)

        left,right = 0, n - 1  # 分别位于输入数组的两端
        maxleft,maxright = height[0],height[n - 1]
        ans = 0

        while left <= right:
            maxleft = max(height[left],maxleft)
            maxright = max(height[right],maxright)
            if maxleft < maxright:
                ans += maxleft - height[left]
                left += 1
            else:
                ans += maxright - height[right]
                right -= 1

        return ans
```

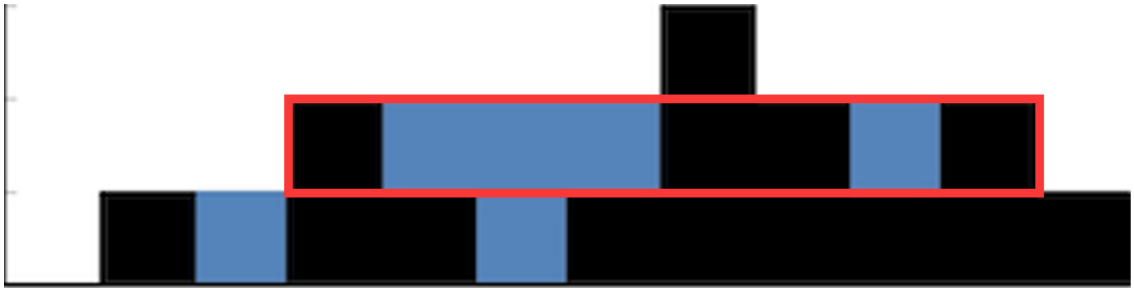
4、双指针——result = 接满雨水后的总面积数 - 柱子面积数

- 利用双指针计算每一[最低高度增加的面积]，累加可得接满雨水后的总面积数。
- 需要寻找每个高度段的范围。
  - temp_h：移动双指针前的双指针处的最小高度
  - min_l_f：移动双指针后的双指针处的最小高度
- 若移动后双指针处的最小高度发生改变，即min_l_f > temp_h，则将增加的高度(min_l_f-temp_h)与对应该高度的范围(right - left + 1)相乘并添加到总面积中
  - `result += (min_l_f-temp_h) * (right - left + 1)`
- 双指针并不断向中间移动，并将改变高度后的面积添加到总面积中。

例：

temp_h = 1
范围为[1,11]
总面积增加 11 = 1 * (11 - 1 + 1)

temp_h = 2
范围为[3,10]
总面积增加 8 = (2 - 1) * (10 - 3 + 1)



temp_h = 3
范围为[6,6]
总面积增加 1 = (3 - 2) * (6 - 6 + 1)

sum(height) = 14
result = 11 + 8 + 1 - 14 = 6

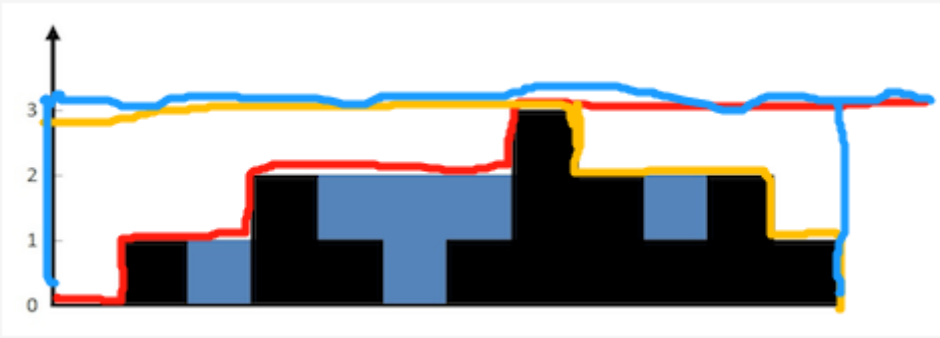
```
class Solution:
    def trap(self, height: List[int]) -> int:
        result = 0
        left, right = 0, len(height) - 1
        temp_h = 0
        while left <= right:
            min_l_f = min(height[left], height[right])
            if min_l_f > temp_h:
                result += (min_l_f-temp_h) * (right - left + 1)
                temp_h = min_l_f
            while left<=right and height[left] <= temp_h:
                left += 1
            while left<=right and height[right] <= temp_h:
                right -= 1
        result = result-sum(height)
        return result
```

5、单调栈

产生凹陷的地方才能存储雨水，那么高度一定是先减后增，所以思路就是维护一个高度递减的stack，

```
class Solution:
    def trap(self, height: List[int]) -> int:
        length = len(height)
        if length < 3: return 0
        res, idx = 0, 0
        stack = []
        while idx < length:
            while len(stack) > 0 and height[idx] > height[stack[-1]]:
                top = stack.pop()
                if len(stack) == 0:
                    break
                h = min(height[stack[-1]], height[idx]) - height[top]
                dist = idx - stack[-1] - 1
                res += (dist * h)
            stack.append(idx)
            idx += 1
        return res
```

6、几何法



- 只需要一次循环，我在这次循环中，用左右两个指针，左指针记录左边遇到的最大值，右指针记录右边遇到的最大值，
- 每轮循环将两个最大值加起来，并且减去当前柱子的高度。
- 当循环结束时，可以发现，我们多加了一个大矩形的面积，
- 所以最后返回的时候把这个矩形面积（lmax * len(height)）减掉就是我们要的结果。
- 在这个图中，红色代表左指针的变化过程，黄色代表右指针的变化过程，蓝色代表大矩形的面积。

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        lmax, rmax, res = 0, 0, 0
        for i in range(len(height)):
            lmax = max(lmax, height[i])
            rmax = max(rmax, height[-1-i])
            res += lmax + rmax - height[i]
        return res - lmax * len(height)
```

72、编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

> ```
> 输入：word1 = "horse", word2 = "ros"
> 输出：3
> 解释：
> horse -> rorse (将 'h' 替换为 'r')
> rorse -> rose (删除 'r')
> rose -> ros (删除 'e')
> ```
>
> ```
> 输入：word1 = "intention", word2 = "execution"
> 输出：5
> 解释：
> intention -> inention (删除 't')
> inention -> enention (将 'i' 替换为 'e')
> enention -> exention (将 'n' 替换为 'x')
> exention -> exection (将 'n' 替换为 'c')
> exection -> execution (插入 'u')
> ```

- **高分解答**

  1、动态规划

  `dp[i][j]` 代表 `word1` 到 `i` 位置转换成 `word2` 到 `j` 位置需要最少步数

  - 当 word1[i] == word2[j]，`dp[i][j] = dp[i-1][j-1]`；
  - 当 word1[i] != word2[j]，`dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1`
    - `dp[i-1][j-1]` 表示替换操作，
    - `dp[i-1][j]` 表示删除操作，
    - `dp[i][j-1]` 表示插入操作。
  - 注意，针对第一行，第一列要单独考虑，我们引入 '' 下图所示：

	“ ”	r	o	s
" "	0	1	2	3
h	1	1	2	3
o	2	2	1	2
r	3	2	2	2
s	4	3	3	2
e	5	4	4	3

```
#自底向上
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        n1 = len(word1)
        n2 = len(word2)
        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        # 第一行
        for j in range(1, n2 + 1):
            dp[0][j] = dp[0][j-1] + 1
        # 第一列
        for i in range(1, n1 + 1):
            dp[i][0] = dp[i-1][0] + 1
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1] ) +
1

        return dp[-1][-1]

#自顶向下
import functools
class Solution:
    @functools.lru_cache(None)
    def minDistance(self, word1: str, word2: str) -> int:
        if not word1 or not word2:
            return len(word1) + len(word2)
```

```
        if word1[0] == word2[0]:
            return self.minDistance(word1[1:], word2[1:])
        else:
            inserted = 1 + self.minDistance(word1, word2[1:])
            deleted = 1 + self.minDistance(word1[1:], word2)
            replace = 1 + self.minDistance(word1[1:], word2[1:])
            return min(inserted, deleted, replace)
```

76、最小覆盖子串

给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

> 输入：s = "ADOBECODEBANC", t = "ABC"
> 输出："BANC"

> 输入：s = "a", t = "a"
> 输出："a"

- **高分解答**

  1、滑动窗口+哈希表

  总体思路

  1. 先右指针right遍历字符串s，在s中找到有t中所有字母的子串（也就是滑动窗口）
  2. 然后开始缩小滑窗，左指针left右移，去掉可以去掉的字母，记录这个子串（滑窗）的最小长度
  3. 右指针right继续遍历，不断更新滑窗最小长度

  具体思路

  1. 用哈希表need记录字符串t中字母出现的次数；在后续操作中，如果出现某个字母的value为负数，则表示这个字母多余
  2. need_cnt用于记录当前剩余所需字母的数量，当所需字母数量为0时，说明当前滑动窗口包含了所需的所有字母，后续应当开始收紧窗口，去除多余字母；
  3. 滑动窗口的左右指针都从0开始，res用于记录符合要求的滑窗的左右指针
  4. right右移，遍历字符串s
     - 如果当前遍历到的字母（即right所指字母）在need中的value大于0，说明这是个需要的字母，need_cnt - 1
     - 无论当前字母是否是需要的字母，need相应的value都要 - 1，前面说过，如果出现某个字母的value为负数，则表示这个字母多余
     - 当need_cnt为0时，代表当前滑动窗口包含了所需的所有字符，后面开始收紧窗口，去除多于字符。
     - 进入while循环：当left所指的字符在need中value为0时，表示该字符不能再减少了，否则当前子字符串就不满足要求了，break；其他情况，表示left所指字符是多余的，left左移，去除该字符，相应need的value + 1（注意是先判断再右移的）
     - 当right - left < res[1] - res[0]时，当前字符串更小，更新res
     - right右移，更新滑动窗口，只有当滑动窗口又包含进新的所需字符时，left才会右移
  5. 若res没有改变过，说明没有符合要求的子串，返回""；否则，返回 s[res[0]: res[1] + 1]

```python
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        n, m = len(s), len(t)
        if n < m: return ""

        need = collections.defaultdict(int)
        for c in t:
            need[c] += 1
        need_cnt = m
        res = [0, n]

        left = 0
        for right, c in enumerate(s):
            if need[c] > 0:
                need_cnt -= 1
            need[c] -= 1
            if need_cnt == 0:
                while True:
                    if need[s[left]] == 0:
                        break
                    need[s[left]] += 1
                    left += 1
                if right - left < res[1] - res[0]:
                    res = [left, right]
        return s[res[0]: res[1] + 1] if res != [0, n] else ""
```

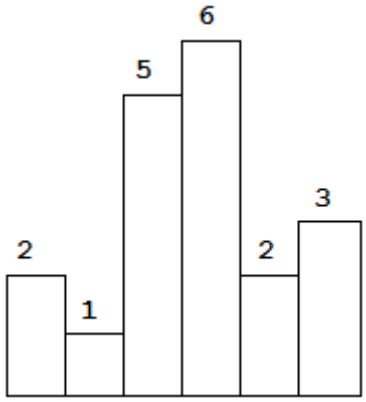
使用数组

```python
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        n, m = len(s), len(t)
        if n < m: return ""

        need = [0] * 58
        for c in t:
            need[ord(c) - ord('A')] += 1
        need_cnt = m
        res = [0, n]

        left = 0
        for right, c in enumerate(s):
            if need[ord(c) - ord('A')] > 0:
                need_cnt -= 1
            need[ord(c) - ord('A')] -= 1
            if need_cnt == 0:
                while True:
                    if need[ord(s[left]) - ord('A')] == 0:
                        break
                    need[ord(s[left]) - ord('A')] += 1
                    left += 1
                if right - left < res[1] - res[0]:
                    res = [left, right]
        return s[res[0]: res[1] + 1] if res != [0, n] else ""
```

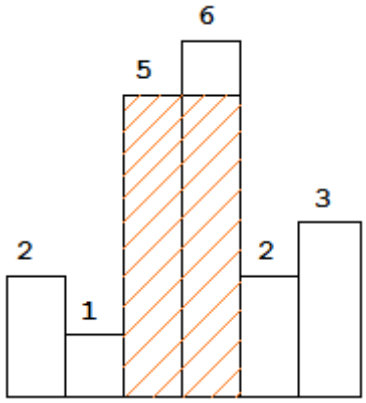
84、柱状图中最大的矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

```
输入: [2,1,5,6,2,3]
输出: 10
```

- **第一想法**

  1、滑动窗口——超时间

  滑动窗口，不断比较能组成的最大矩形

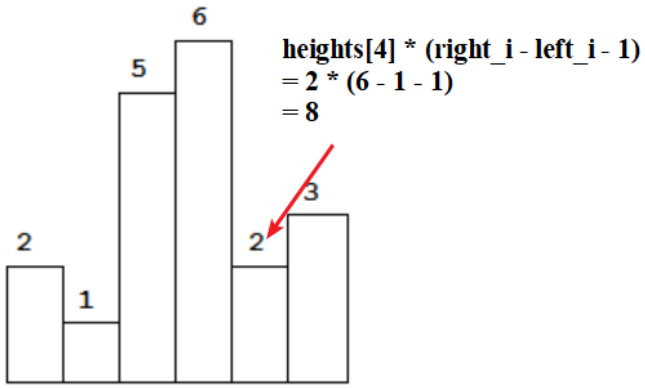
```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        res=0
        for i in range(1,len(heights)+1):
            for j in range(len(heights)-i+1):
                res=max(res,min(heights[j:j+i])*i)

        return res
```

- **高分解答**

  1、 找两边第一个小于他的值

  要想找到第 `i` 位置最大面积

是以i 为中心，向左找第一个小于 heights[i] 的位置 left_i；向右找第一个小于于 heights[i] 的位置 right_i，即最大面积为 heights[i] * (right_i - left_i -1)，如下图所示：



思路一：

当我们找 i 左边第一个小于 heights[i] 如果 heights[i-1] >= heights[i] 其实就是和 heights[i-1] 左边第一个小于 heights[i] 一样。依次类推，右边同理。

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        if not heights:
            return 0
        n = len(heights)
        left_i = [0] * n
        right_i = [0] * n
        left_i[0] = -1
        right_i[-1] = n
        for i in range(1, n):
            tmp = i - 1
            while tmp >= 0 and heights[tmp] >= heights[i]:
                tmp = left_i[tmp]
            left_i[i] = tmp
        for i in range(n - 2, -1, -1):
            tmp = i + 1
            while tmp < n and heights[tmp] >= heights[i]:
                tmp = right_i[tmp]
            right_i[i] = tmp
        # print(left_i)
        # print(right_i)
        res = 0
        for i in range(n):
            res = max(res, (right_i[i] - left_i[i] - 1) * heights[i])
        return res
```

思路二：栈

维护一个单调递增的栈，就可以找到 `left_i` 和 `right_i`。

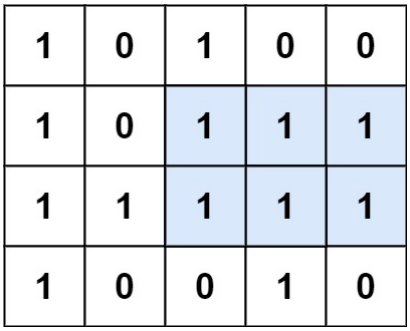
- `[0]+heights+[0]`：左边加0是为了防止栈空，避免增加判断条件;右面加0是为了防止最后栈内残留递增序列，无法完全弹出。
- 入栈的是下标，不是高度，因为我们可以很容易从下标获取高度，同时需要根据下标计算矩形宽度。
- 更新面积一定要边弹出边更新，即 `res = max(res, (i - stack[-1] - 1) * heights[tmp])` 在 `while` 之内

- 宽度=i-stack[-1]-1: 因为i 是 j 的右侧第一个对应高度小于heights[j]的下标，stack[-1]为左侧第一个对应高度小于heights[tmp]下标.

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = []
        heights = [0] + heights + [0]
        res = 0
        for i in range(len(heights)):
            #print(stack)
            while stack and heights[stack[-1]] > heights[i]:
                tmp = stack.pop()
                res = max(res, (i - stack[-1] - 1) * heights[tmp])
            stack.append(i)
        return res
```

85、最大矩形

给定一个仅包含 `0` 和 `1` 、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 `1` 的最大矩形，并返回其面积。

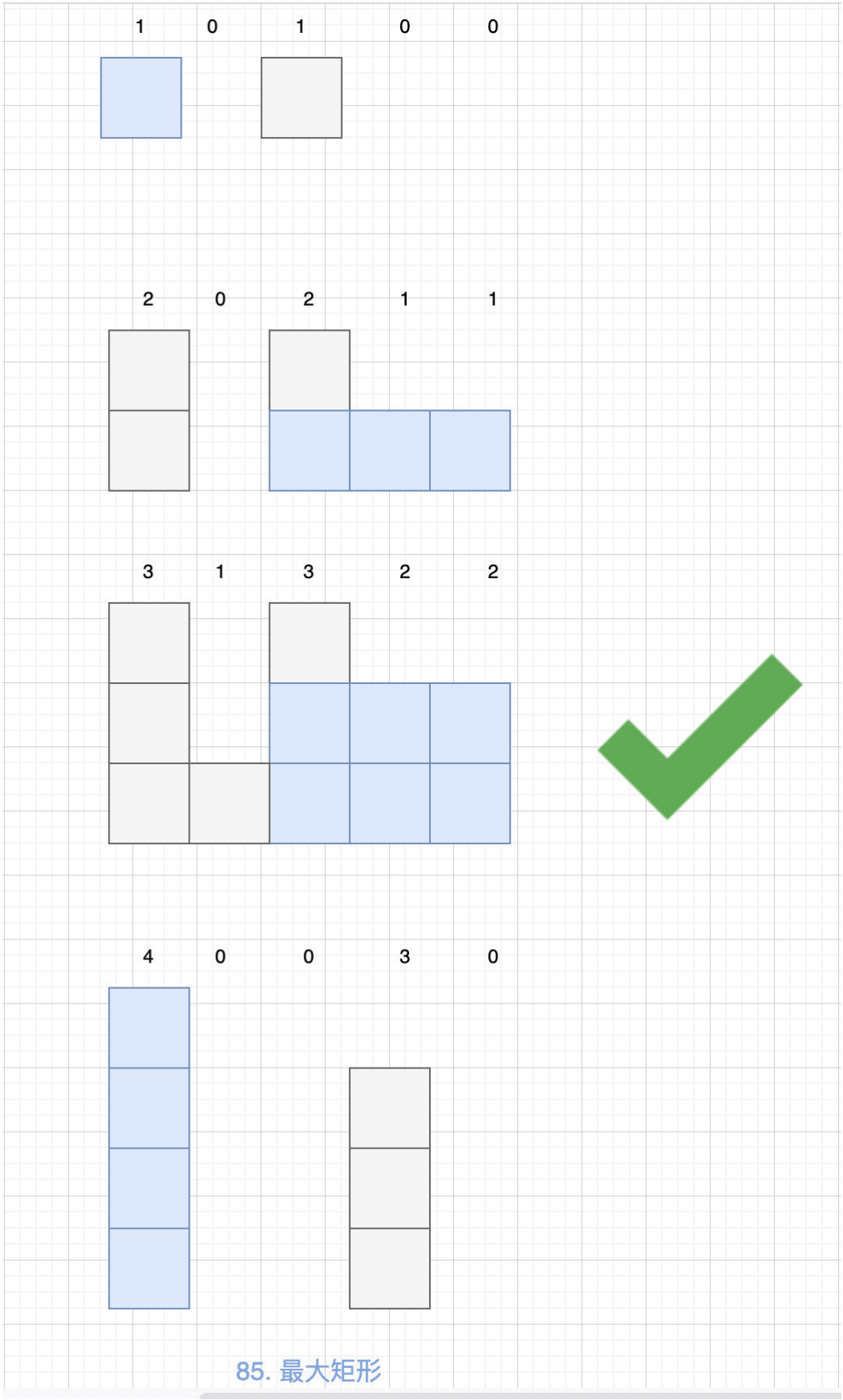


1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

> 输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],
> ["1","1","1","1","1"],["1","0","0","1","0"]]
> 输出：6

- **高分解答**

  1、逐行扫描，调用84题函数



85. 最大矩形

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n, heights, st, ans = len(heights), [0] + heights + [0], [], 0
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                ans = max(ans, heights[st.pop(-1)] * (i - st[-1] - 1))
            st.append(i)
```

```
        return ans
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        m = len(matrix)
        if m == 0: return 0
        n = len(matrix[0])
        heights = [0] * n
        ans = 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == "0":
                    heights[j] = 0
                else:
                    heights[j] += 1
            ans = max(ans, self.largestRectangleArea(heights))
        return ans
```

2、位运算

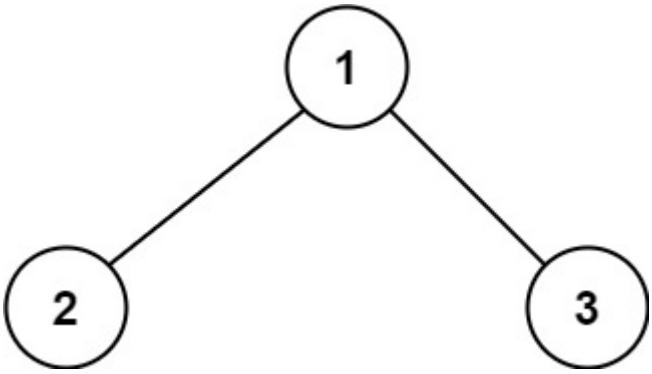
```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        if not matrix or not matrix[0]:
            return 0
        #将每一行看作一个二进制数，然后转化为一个整数
        nums = [int(''.join(row), base=2) for row in matrix]
        ans, N = 0, len(nums)
        #遍历所有行
        for i in range(N):
            j, num = i, nums[i]
            #将第i行，连续的，和接下来的所有行，做与运算
            while j < N:
                #经过与运算后，num转化为二进制中的1，表示从i到j行，可以组成一个矩形的那几列
                num = num & nums[j]
                if not num:
                    break
                l, curnum = 0, num
                #这个循环最精彩
                #每次循环将curnum和其左移一位的数做与运算
                #最终的循环次数l表示，最宽的有效宽度，
                while curnum:
                    l += 1
                    curnum = curnum & (curnum << 1)
                ans = max(ans, l * (j - i + 1))
                j += 1
        return ans
```

124、二叉树中的最大路径和

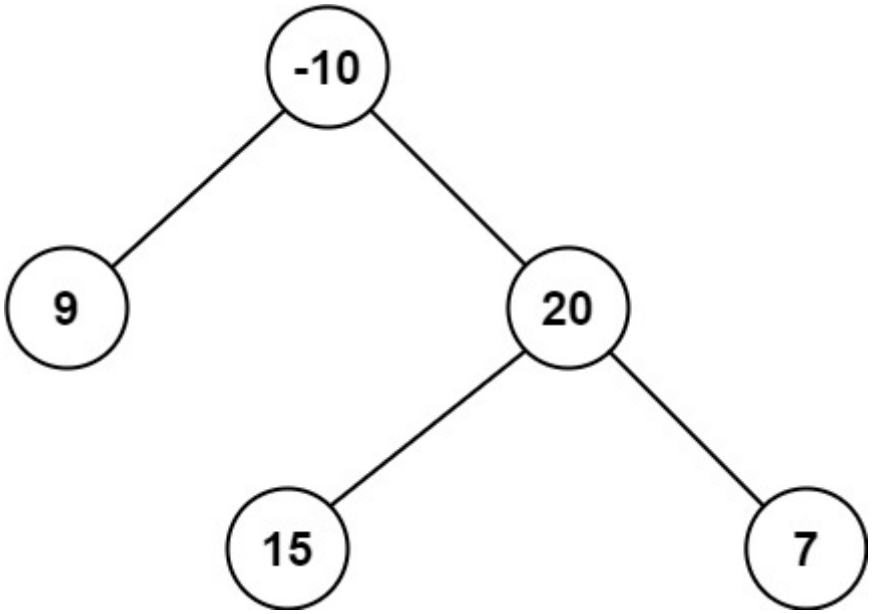
路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次 。该路径 至少包含一个 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 root ， 返回其 最大路径和 。




```
输入：root = [1,2,3]
输出：6
解释：最优路径是 2 -> 1 -> 3 ，路径和为 2 + 1 + 3 = 6
```




```
输入：root = [-10,9,20,null,null,15,7]
输出：42
解释：最优路径是 15 -> 20 -> 7 ，路径和为 15 + 20 + 7 = 42
```

- **高分解答**

  1、递归

  1. 首先，本来我想用原本的函数maxPathSum()直接递归（能简则简），但是发现行不通，因为始终需要一个外部变量参与比较，更新最大值，然后就另外写了dfs()的函数用来递归。
  2. 其次，递归函数的参数，本题不用改，因为只需要一个根节点遍历完二叉树就行，且最大值必须要遍历完才知道是不是最大值。所以是dfs(root)就行。
  3. 然后，鉴于第一点的分析，这题我需要一个外部（全局）变量self.maxsum来记录当前的最大路径值，并在递归的过程中不断更新最大值。
  4. 最后，本题最重要的就是返回值的设计，鉴于最开始的分析，其实返回值有和没有并不影响递归遍历节点，但是本题必须要借用返回值来在每一层递归中比较路径的大小，因为如果不将这个路径的大小随着函数返回的话，外部变量self.maxsum将会无法每一轮比较和更新。
  5. 在本题中，这个路径可以是任意一个路径，甚至可以只是一个节点，所以我每次比较的都是当前节点的左子树和右子树的大小，那个大选哪个，然后还要加上这个根节点的值val，最后与0相比较，因为如果这个树枝比0还小，那不要也罢，直接返回0，要不然最大值还减小了。
  6. 之前的self.maxsum还没有用，现在我们知道dfs(root)一定返回的是当前节点以下，所包含路径的最大值(有可能为0)，所以我们就每轮都比较并记录，self.maxsum与左子树与右子树的

最大值加上当前节点的值val(说的有点乱)，即self.maxsum = max(self.maxsum, left + right + root.val)。

7. 最后回到主函数，返回self.maxsum即可，而此时dfs()函数返回的是根节点以下所包含路径的最大值。

```
class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        self.maxsum = float('-inf')
        def dfs(root):
            if not root: return 0
            left = dfs(root.left)
            right = dfs(root.right)
            self.maxsum = max(self.maxsum, left + right + root.val)
            return max(0, max(left, right) + root.val)
        print(dfs(root))
        return self.maxsum
```

128、最长连续序列

给定一个未排序的整数数组 `nums` ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

> 输入：nums = [100,4,200,1,3,2]
> 输出：4
> 解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

> 输入：nums = [0,3,7,2,5,8,4,6,0,1]
> 输出：9

- **第一想法**

  1、数组排序，一次遍历，查找最长连续字符

  - 字符相等：跳过
  - 字符相减==1：长度+1
  - 字符相减！=1：长度重新归

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        if len(nums)==0:return 0
        nums.sort()
        res=1
        max_len=1
        for i in range(1,len(nums)):
            if nums[i]==nums[i-1]:
                continue
            elif (nums[i]-nums[i-1])==1:
                res+=1
                max_len=max(max_len,res)
            else:
                res=1
```

```
        return max_len
```

- **高分解答**

  1、字典

  遍历数组, 用字典(哈希)记录目前与该值可以组成最长连续序列.

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        lookup = {}
        res = 0
        for num in nums:
            if num not in lookup:
                # 判断左右是否可以连起来
                left = lookup[num - 1] if num - 1 in lookup else 0
                right = lookup[num + 1] if num + 1 in lookup else 0
                # 记录长度
                lookup[num] = left + right + 1
                # 把头尾都设置为最长长度
                lookup[num - left] = left + right + 1
                lookup[num + right] = left + right + 1
                res = max(res, left + right + 1)
        return res
```

  2、并查集——没懂，详细看题解

239、滑动窗口最大值

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

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

- **第一想法**

1、最简单的方法，保存每次滑动数组的最大值——超时间

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        res=[]
        for i in range(len(nums)-k+1):
            res.append(max(nums[i:i+k]))

        return res
```

优化：使用字典——还是超时间

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        a={}
        res=[]
        for i in range(k-1):
            a[i]=nums[i]
        for i in range(k-1,len(nums)):
            a[i]=nums[i]
            res.append(max(a.values()))
            a.pop(i-k+1)
        return res
```

2、堆——参考评论区解答

1. python默认是小根堆，故加"-"，形成"大根堆"
2. 由于堆顶元素可能不在滑动窗口内，故要维护一个二元组(num, index)
3. 通过index判断堆顶元素是否在滑动窗口内
4. 首先把 k 个元素加入大根堆
5. 接着模拟滑动窗口右移，把最新的元素加入大根堆，维护堆顶元素

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # 大根堆
        n = len(nums)
        # python默认是小根堆，故加"-"，形成"大根堆"
        # 由于堆顶元素可能不在滑动窗口内，故要维护一个二元组(num, index)
        # 通过index判断堆顶元素是否在滑动窗口内
        # 首先把 k 个元素加入大根堆
        q = [(-nums[i], i) for i in range(k)]
        heapq.heapify(q)

        ans = [-q[0][0]]
        for i in range(k, n):
            # 把最新的元素加入大根堆
            heapq.heappush(q, (-nums[i], i))
            # 判断堆顶元素（下标）是否在滑动窗口内
            while q[0][1] <= i - k:
                heapq.heappop(q)
            # 把大根堆的堆顶元素加入ans
            ans.append(-q[0][0])
        return ans
```

- **高分解答**

  1、双端队列

  1. 利用双端队列记录当前滑动窗口的元素索引
  2. 队列最左侧元素记录滑动窗口中最大元素的索引
  3. 遍历数组：
     - 如果队列最左侧索引已不在滑动窗口范围内，弹出队列最左侧索引
     - 通过循环确保队列的最左侧索引所对应元素值最大
     - 新元素入队
     - 从第一个滑动窗口的末尾索引开始将最大值存储到结果res中

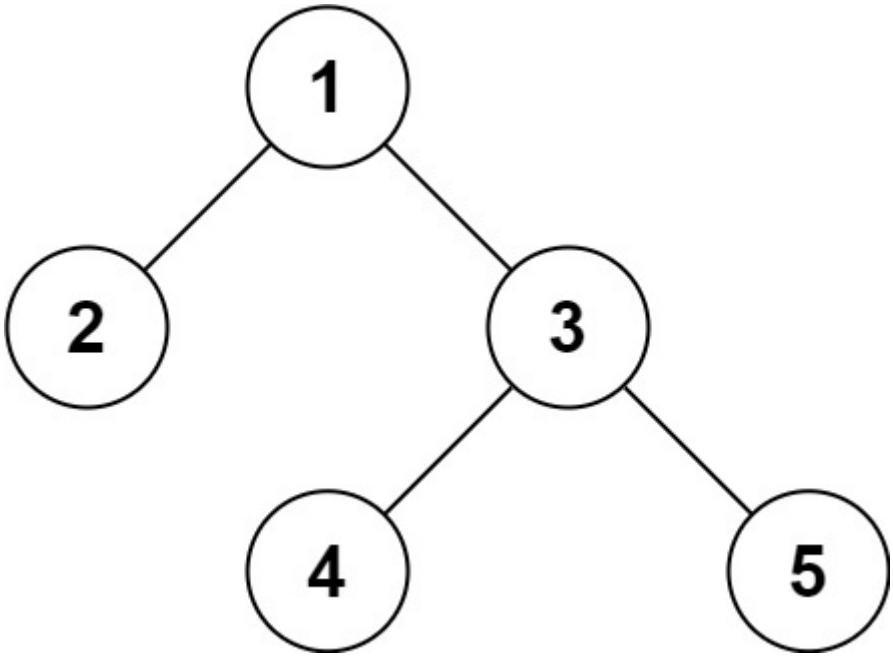
```python
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        res = []
        queue = collections.deque()
        for i, num in enumerate(nums):
            if queue and queue[0] == i - k:
                queue.popleft()
            while queue and nums[queue[-1]] < num:
                queue.pop()
            queue.append(i)
            if i >= k - 1:
                res.append(nums[queue[0]])
        return res
```

297、二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

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

提示: 输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。



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

- **高分解答**

  1、BFS

  - 序列化

    1. 用BFS遍历树，与一般遍历的不同点是不管node的左右子节点是否存在，统统加到队列中
    2. 在节点出队时，如果节点不存在，在返回值res中加入一个"null"；如果节点存在，则加入节点值的字符串形式

  - 反序列化

    1. 同样使用BFS方法，利用队列新建二叉树
    2. 首先要将data转换成列表，然后遍历，只要不为null将节点按顺序加入二叉树中；同时还要将节点入队
    3. 队列为空时遍历完毕，返回根节点

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

    def deserialize(self, data):
        if not data:
```

```
            return []
        dataList = data[1:-1].split(',')
        root = TreeNode(int(dataList[0]))
        queue = collections.deque([root])
        i = 1
        while queue:
            node = queue.popleft()
            if dataList[i] != 'None':
                node.left = TreeNode(int(dataList[i]))
                queue.append(node.left)
            i += 1
            if dataList[i] != 'None':
                node.right = TreeNode(int(dataList[i]))
                queue.append(node.right)
            i += 1
        return root
```

2、DFS

- 序列化
    1. 递归的第一步都是特例的处理，因为这是递归的中止条件：如果根节点为空，返回”null“
    2. 序列化的结果为：根节点值 + "," + 左子节点值(进入递归) + "," + 右子节点值(进入递归)
    3. 递归就是不断将“根节点”值加到结果中的过程
- 反序列化
    1. 先将字符串转换成队列（python转换成列表即可）
    2. 接下来就进入了递归
        1. i. 弹出左侧元素，即队列出队
        2. ii. 如果元素为“null”，返回null（python返回None）
        3. iii. 否则，新建一个值为弹出元素的新节点
        4. iv. 其左子节点为队列的下一个元素，进入递归；右子节点为队列的下下个元素，也进入递归
        5. v. 递归就是不断将子树的根节点连接到父节点的过程

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

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

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

301、删除无效括号

给你一个由若干括号和字母组成的字符串 `s` ，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 **任意顺序** 返回。

> ```
> 输入："()())()"
> 输出：["()()()", "(())()"]
> ```
>
> ```
> 输入："(a)())()"
> 输出：["(a)()()", "(a())()"]
> ```
>
> ```
> 输入：")("
> 输出：[""]
> ```

- **高分解答**

  1、字符串+哈希表+递归+深度优先搜索+记忆化

  1. 特殊情况判断：
     - 如果`s`长度为`1`且`s`不是括号：返回`[s]`。
     - 如果`s`长度小于`2`：返回`[""]`。
  2. 初始化一个空集合`res`。
  3. 如果s[0]等于"("与s[-1]等于")"匹配：
     - 在self.removeInvalidParentheses(s[1:-1])的结果的所有字符串的左边加上左括号，再在右边加上右括号，并加入返回集合。
     - 原因：一个有效的字符串，在左边加上左括号，再在右边加上右括号，那么，改变后的字符串依然是有效字符串。
  4. 循环分割：
     - 将左部分的结果和右部分的结果的所有组合加入返回集合。
     - 原因：一个有效的字符串和另一个有效的字符串合并，得到的字符串依然是有效字符串。
  5. 返回res中最长的几个字符串。

     注意：这里需要**记忆化**（`@lru_cache`），否则超时。

```python
class Solution:
    @lru_cache
    def removeInvalidParentheses(self, s: str) -> List[str]:
        if len(s) == 1 and s not in "()":
            return [s]
        if len(s) < 2:
            return [""]
        res = set()
        if s[0] == "(" and s[-1] == ")":
            res |= set([f"({i})" for i in
self.removeInvalidParentheses(s[1:-1])])
        for i in range(1, len(s)):
            a, b = self.removeInvalidParentheses(s[:i]),
self.removeInvalidParentheses(s[i:])
            res |= {i+j for i in a for j in b}
        p = len(max(res, key=len))
```

```
        return [i for i in res if len(i) == p]
```

2、字符串+哈希表+递归+深度优先搜索+记忆化

与方法一区别：这个解法是匹配括号，解法一是分割括号

```
class Solution:
    @lru_cache
    def removeInvalidParentheses(self, s: str) -> List[str]:
        if not s:
            return [""]
        if s[0] not in "()":
            return [s[0]+i for i in self.removeInvalidParentheses(s[1:])]
        if len(s) < 2:
            return [""]
        if s[0] == ")":
            return self.removeInvalidParentheses(s[1:])
        res = set(self.removeInvalidParentheses(s[1:]))
        for i in range(1, len(s)):
            if s[i] == ")":
                a, b = set(self.removeInvalidParentheses(s[1:i])),
set(self.removeInvalidParentheses(s[i+1:]))
                res |= {f"({i}){j}" for i in a for j in b}
        p = len(max(res, key=len))
        return [i for i in res if len(i) == p]
```

3、字符串+哈希表+动态规划

1. 特殊情况判断：
    - 如果`s`长度小于`2`：返回`[""]`。
2. 定义dp列表，默认值都为[""]。
3. 循环i, j填表（i反着循环，j正着循环，i表示开始索引，j表示结尾索引）：
    - 定义变量得到当前的字符串。
    - 剩下代码与**解法一**思路一致。
4. 返回`dp[0][-1]`。

```
class Solution:
    def removeInvalidParentheses(self, s: str) -> List[str]:
        if not s:
            return [""]
        dp = [[[""] for __ in range(len(s)+1)] for __ in range(len(s))]
        for i in range(len(s)-1, -1, -1):
            for j in range(i+1, len(s)+1):
                ss = s[i:j]
                if len(ss) == 1:
                    if ss not in "()":
                        dp[i][j] = [ss]
                    continue
                res = set()
                if ss[0] == "(" and ss[-1] == ")":
                    res |= set([f"({i})" for i in dp[i+1][j-1]])
                for k in range(i+1, j):
                    a, b = set(dp[i][k]), set(dp[k][j])
                    res |= {A+B for A in a for B in b}
                p = len(max(res, key=len))
```

```
                    dp[i][j] = [k for k in res if len(k) == p]
        return dp[0][-1]
```

312、戳气球

有 n 个气球，编号为0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

> 输入：nums = [3,1,5,8]
> 输出：167
> 解释：
> nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
> coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

> 输入：nums = [1,5]
> 输出：10

- **高分解答**

  1、动态规划

`dp[i][j]` 表示开区间 `(i,j)` 内你能拿到的最多金币

如果你此刻选择戳爆气球 k，那么你得到的金币数量就是：

`total=dp[i][k]+val[i] * val[k] * val[j]+dp[k][j]`

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

        #nums首尾添加1，方便处理边界情况
        nums.insert(0,1)
        nums.insert(len(nums),1)

        store = [[0]*(len(nums)) for i in range(len(nums))]

        def range_best(i,j):
            m = 0
            #k是(i,j)区间内最后一个被戳的气球
            for k in range(i+1,j): #k取值在(i,j)开区间中
                #以下都是开区间(i,k), (k,j)
```

```
                left = store[i][k]
                right = store[k][j]
                a = left + nums[i]*nums[k]*nums[j] + right
                if a > m:
                    m = a
            store[i][j] = m

        #对每一个区间长度进行循环
        for n in range(2,len(nums)): #区间长度 #长度从3开始，n从2开始
            #开区间长度会从3一直到len(nums)
            #因为这里取的是range，所以最后一个数字是len(nums)-1

            #对于每一个区间长度，循环区间开头的i
            for i in range(0,len(nums)-n): #i+n = len(nums)-1

                #计算这个区间的最多金币
                range_best(i,i+n)

        return store[0][len(nums)-1]
```