

技巧类

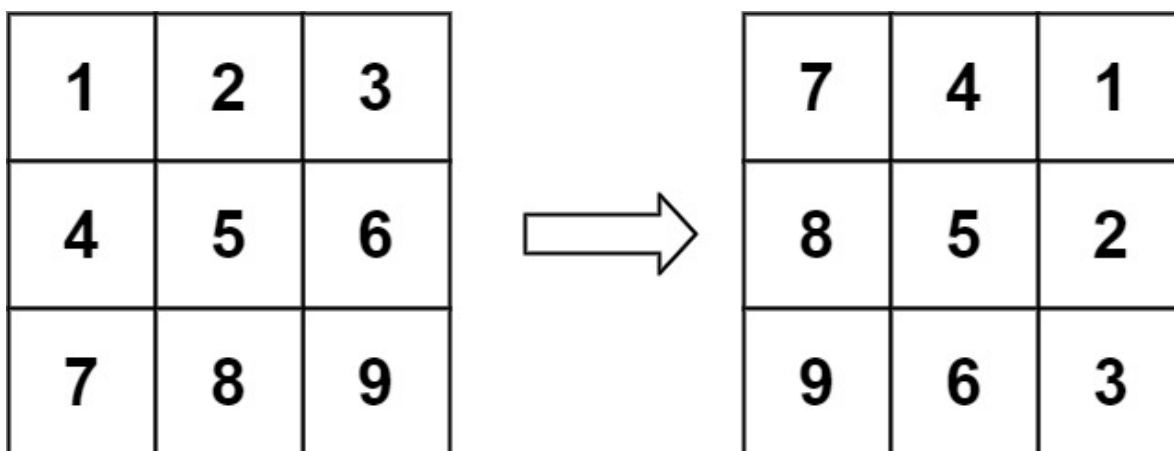
矩阵数据旋转

48. 旋转图像

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

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

示例 1:



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

输出: `[[7,4,1],[8,5,2],[9,6,3]]`

思路：对于正方形，最外圈是规律变化的，因此只需要将最外圈进行进行顺时针置换就行了。需要注意深拷贝和浅拷贝的问题。

```
class Solution:

    def rotate(self, matrix) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)
        mid = int(n/2)

        for i in range(mid):

            # 深拷贝
            left = [a[i] for a in matrix]
            right = [b[n-i-1] for b in matrix]
            upper = matrix[i].copy()
            bottom = matrix[n-i-1].copy()

            for j in range(i, n-i):
```

```
matrix[i][j] = left[n-j-1]
matrix[j][n-i-1] = upper[j]
matrix[n-i-1][j] = right[n-j-1]
matrix[j][i] = bottom[j]
```

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。

示例:

```
输入: [1,2,3,4]
输出: [24,12,8,6]
```

思路：采用左右双数组，L和R

$$L[i] = \prod_{j=0}^{i-1} nums[j]$$

$$R[i] = \prod_{j=i+1}^{n-1} nums[j]$$

$$output[i] = L[i] * R[i]$$

所以有 $L = [1,1,2,6]$, $R = [24,12,4,1]$ 。

哈希

1、两数之和 给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 和为目标值 的那两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

你可以按任意顺序返回答案。

```
输入: nums = [2,7,11,15], target = 9
输出: [0,1]
解释: 因为 nums[0] + nums[1] == 9，返回 [0, 1]。
```

解析：利用哈希，对于每一个元素 x ，利用哈希表保存 x 的值及其对应的索引，然后当 $target-y$ 在hash表中时，则说明前面有值 $x+y = target$ 。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        hashtable = {}
        for index, val in enumerate(nums):
            if target - nums[index] in hashtable:
                return [hashtable[target - nums[index]], index]
            hashtable[nums[index]] = index

        return [0, 0]
```

49. 字母异位词分组

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

示例:

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

思路: 排序+哈希

["ate","eat","tea"]排序之后，都是aet，可以将ate全都存到hash表中，key值为ate，每个key值保存的都是对应的字母异位词组

```
class Solution:
    """
    思路: 将字符串转换成列表排序后合并，然后进行映射，将结果相同的合并
    """
    def groupAnagrams(self, strs):

        res = {}
        for s in strs:
            word_label = ''.join(sorted(list(s)))
            if word_label not in res:
                res[word_label] = []

            res[word_label].append(s)

        return list(res.values())
```

169. 多数元素

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

思路:

思路一: 哈希，找到每个元素出现的次数，然后挑选出出现次数大于 $n/2$ 的

思路二: 排序，利用快排对数组进行排序，然后找到中间位置的数，即为 $n/2$ 的数据。

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。

思路: 哈希，同上一题相同思路。

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 的连续子数组的个数。

示例 1:

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

说明:

1. 数组的长度为 [1, 20,000]。
2. 数组中元素的范围是 [-1000, 1000] , 且整数 k 的范围是 [-1e7, 1e7]。

思路: 利用哈希记录当前位置的前缀值prefix, 此时prefix表示每个位置的前缀和, 以实例1为例prefix=[1,2,3]。此时要判断当前位置的连续和是否为k只需要判断prefix[i] - k 是否在prefix中就行了, 当存在前缀和的时候, 有prefix[j] + k = prefix[i],

此时[i,j]之间的数组和为k。因此只要求前缀和, 然后每次判断prefix[i]-k是否在其中, 当同一个前缀和出现多次, 表示有不同长度的数组到当前位置i的结果为k。

```
class Solution:
    """
    思路一: 暴力枚举, 不解释连招, 超时
    思路二: 利用哈希记录前缀值
    """

    def subarraySum(self, nums, k):
        # 初始化结果
        pre = {0:1}
        cur = 0
        count = 0
        for i in range(len(nums)):
            cur += nums[i]
            if cur - k in pre:
                count += pre[cur - k]

            if cur in pre:
                pre[cur] += 1
            else:
                pre[cur] = 1

        return count
```

[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。

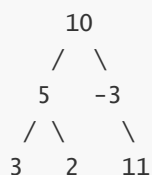
找出路径和等于给定数值的路径总数。

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。

示例:

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



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

返回 3。和等于 8 的路径有：

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

思路：前缀和+dfs。

由于路径只能从上到下，所以都是单路径，可以求每条路径上的前缀和，在判断结果是否符合。需要注意，如果两个路径都有节点的前缀和为n，那么在统计另外一个前缀和是否为n的时候，不应该考虑当前路径的情况。

```
class Solution:
    """
    前缀和+递归，与560题思路类似，注意
    """
    def pathSum(self, root: TreeNode, sum: int) -> int:

        count = 0
        prefix = {0: 1}

        def dfs(root, total, prefix):
            nonlocal count, sum

            if not root:
                return
            else:
                total += root.val
                if total - sum in prefix:
                    count += prefix[total - sum]

                if total in prefix:
                    prefix[total] += 1
                else:
                    prefix[total] = 1
                dfs(root.left, total, prefix)
                dfs(root.right, total, prefix)
                # 保持前缀都是在一条路径下面
                prefix[total] -= 1

        dfs(root, 0, prefix)
        return count
```

[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** 的字母异位词的字串，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。

说明：

- 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。

示例 1:

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。

思路: 哈希+滑窗

将p用哈希表保存, 记录下每个字母出现的次数p_count, 然后采用滑窗, 记录s在滑窗中的字符s_count, 如果p_count=s_count, 说明当前滑窗中的字符和p为异位词。然后向右移动, 左边移出字符次数-1, 右边移入字符次数+1, 然后一次对比p_count是否等于s_count。

```
class Solution:
    """
    思路: 参考49, 滑窗+哈希+排序。超时, 还需要优化
    思路二: 滑窗+哈希表, 采用哈希将结果保存
    """
    def findAnagrams(self, s: str, p: str) :
        p_count = [0] * 26
        s_count = [0] * 26

        m = len(p)
        n = len(s)

        for c in p:
            p_count[ord(c)-97] += 1

        res = []

        left, right = 0, 0
        for right in range(n):
            if right < m - 1:
                s_count[ord(s[right]) - 97] += 1
            else:
                s_count[ord(s[right]) - 97] += 1

                if s_count == p_count:
                    res.append(left)

                s_count[ord(s[left]) - 97] -= 1
                left += 1

        return res
```

[448. 找到所有数组中消失的数字](#)

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

找到所有在 $[1, n]$ 范围之间没有出现在数组中的数字。

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

示例:

```
输入:
[4,3,2,7,8,2,3,1]

输出:
[5,6]
```

思路一: 哈希, 构建一个 $1-n$ 的哈希表, 找到出现次数为0的数。时间复杂度 $O(n)$, 空间复杂度 $O(n)$

思路二: 原地修改。

贪心算法

贪心算法(又称贪婪算法)是指, 在对问题求解时, 总是做出在当前看来是最好的选择。也就是说, 不从整体最优上加以考虑, 他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解, 关键是贪心策略的选择, 选择的贪心策略必须具备无后效性, 即某个状态以前的过程不会影响以后的状态, 只与当前状态有关。

[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)。

示例 1:

```
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。
```

思路: 排序+贪心

由于个子矮的对个子高的没有影响, 因此优先将个子矮的放到前面(对于排在他前面的人数相同时)。

以下写法是考虑到插入排序的特性, 优先把高的放到一个位置, 然后将低的放到同样的位置, 高的向后移动一位。

```
class Solution:
    """
    贪心+排序：首先将身高按照降序排序，身高高的优先放到前面
    个子矮的对于个子高的没有影响，因此个子矮的一般要放到前面

    """
    def reconstructQueue(self, people):
        people.sort(key=lambda x: (-x[0], x[1]))
        output = []
        for p in people:
            output.insert(p[1], p)

        return output
```

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**。

示例 1:

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。

思路：贪心+分桶

对于不同的任务，根据任务重复数对其进行分桶，然后计算按照任务次数多少，依次往桶里面填充。

总排队时间= (桶个数 - 1) * 桶的容量 + 最后一桶的任务数

| | | | |
|---|---|---|---|
| A | B | C | F |
| A | B | C | F |
| A | B | D | |
| A | B | D | |
| A | B | D | |
| A | B | × | |

```
class Solution:
    """
    分桶的思想:
    """
    def leastInterval(self, tasks, n) -> int:
```



```
import collections
freq = collections.Counter(tasks)

# 最多的执行次数
max_exec = max(freq.values())
# 具有最多执行次数的任务数量
max_count = sum(1 for v in freq.values() if v == max_exec)
# 总排队时间 = (桶个数 - 1) * (n + 1) + 最后一桶的任务数
return max((max_exec - 1) * (n + 1) + max_count, len(tasks))
```

动态规划

算法解释：

动态规划（英语：Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。动态规划往往用于优化递归问题，例如斐波那契数列，如果运用递归的方式来求解会重复计算很多相同的子问题，利用动态规划的思想可以减少计算量。

通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，具有天然剪枝的功能，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

回文串

5.最长回文子串

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

示例 1：

```
输入: s = "babad"
输出: "bab"
解释: "aba" 同样是符合题意的答案。
```

思路：动态规划，关键思想，以当前位数`s[i]`为回文字符串尾部，每次只考虑`max_len+1`或者`max_len+2`的长度。也就是说，每一次

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        if not s:
            return ""
        length = len(s)
        # 如果字符串长度为或者s本身是字符串，直接返回
        if length == 1 or s == s[::-1]:
            return s
        max_len, start = 1, 0
        # 遍历每一个字符，假设为回文字符的尾字符
        for i in range(1, length):
            # [i-max_len, i], 一共max_len+1个元素
            even = s[i - max_len:i + 1]
            # [i-max_len-1, i] 一共max_len+2个元素
```

```

odd = s[i - max_len - 1:i + 1]
if i - max_len - 1 >= 0 and odd == odd[::-1]:
    start = i - max_len - 1
    max_len += 2
elif i - max_len >= 0 and even == even[::-1]:
    start = i - max_len
    max_len += 1
return s[start:start + max_len]

```

正则表达式

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。

根据分析其实可以得出 p 的第一个字符只能为 '.' 或者 string 字符，不能是 '*'

正则匹配的遍历条件是两者同时匹配到终点。

如果两者同时到达结尾，则能匹配上，如果其中一个 p 已经遍历结束，但是 s 还没有遍历完则匹配不上

由于模式里面第一个不可能是 '*'。判断的时候我们只需要从左往右判断就行。

如果 p 的第二字符是*，则需要判断第一个字符是否能匹配上，如果能匹配上。则，模式有三种：

- 字符串向下移动，但是 p 不变。
- p 移动两位，即*的中间匹配位数为0， s 不动
- p 不变， s 向后移动。

如果 p 的第二个字符不是*，判断就很简单，只需要一个一个往后面移动。

状态方程有：

1、当 p 的第 j 个时小写字母时， s 中必须为相同的小写字母有

$$f[i][j] = \begin{cases} f[i-1][j-1] & p[j]=s[i] \\ false & p[j] \neq s[i] \end{cases}$$

2、当第 p 个为 '.' 的时候，可以匹配 s 中任意小写字母，此时

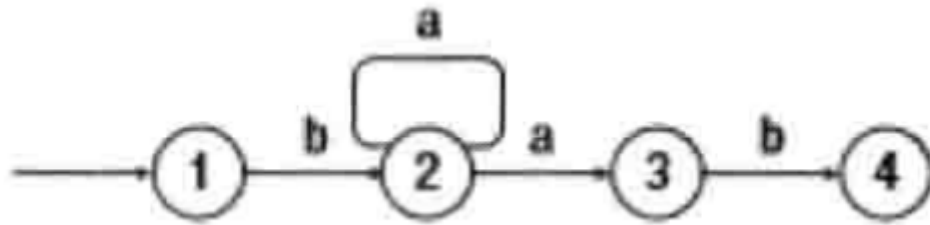
$$f[i][j] = f[i-1][j-1]$$

3、当 p 的第 j 个字符为 '*'，可以匹配前面一个字符0次或者无数次。

当 $s[i] = p[j-1]$ 的时候，有

$$f[i][j] = \begin{cases} f[i][j-1] & s[i]=p[j-1], \text{ 匹配一个} \\ f[i][j-2] & s[i] \neq p[j-1], \text{ 不匹配} \end{cases}$$

$$f[i][j] = \begin{cases} f[i-1][j] \text{ or } f[i][j-2] & p[j]='*', s[i]=p[j-1] \\ f[i][j-2] & s[i] \neq p[j-1] \end{cases}$$



直接对照代码更容易理解

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)

        def matches(i: int, j: int) -> bool:
            if i == 0:
                return False
            if p[j - 1] == '.':
                return True
            return s[i - 1] == p[j - 1]

        f = [[False] * (n + 1) for _ in range(m + 1)]
        f[0][0] = True
        for i in range(m + 1):
            for j in range(1, n + 1):
                # 如果p[j]='*'
                if p[j - 1] == '*':
                    # 匹配0个的情况
                    f[i][j] |= f[i][j - 2]
                    # 匹配一个的情况
                    if matches(i, j - 1):
                        f[i][j] |= f[i - 1][j]
                else:
                    if matches(i, j):
                        f[i][j] |= f[i - 1][j - 1]
        return f[m][n]
```

子序列系列

[53. 最大子序和](#)

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

示例 1：

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

思路：动态规划，关键点在于当和小于0时，需要重新选择序列头

```
class Solution:
    def maxSubArray(self, nums):
```

```

cur = 0
sum_all = -float('inf')
for i in range(len(nums)):

    if cur > 0:
        cur += nums[i]
    else:
        cur = nums[i]

    if cur > sum_all:
        sum_all = cur

return sum_all

```

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。

进阶：你可以设计并实现时间复杂度为 $O(n)$ 的解决方案吗？

示例 1：

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。

思路：

一：排序+双指针

思路二：哈希

代码：

```

#
class Solution:
    """
    思路一：排序 + 双指针
    """
    def longestConsecutive(self, nums):

        nums = sorted(nums)
        length = len(nums)

        if length < 2:
            return length

        start = 0
        end = 0
        cur = 1
        max_len = 0
        while cur < length:
            if nums[end] + 1 == nums[cur] or nums[end] == nums[cur]:
                end = cur
            else:
                end = cur
                start = end
            cur += 1
        return max_len

```

```

        l = len(set(nums[start:end+1]))

        if l > max_len:
            max_len = l

        cur += 1

    return max_len

class Solution:
    def longestConsecutive(self, nums):
        longest_streak = 0
        num_set = set(nums)

        for num in num_set:
            if num - 1 not in num_set:
                current_num = num
                current_streak = 1

                while current_num + 1 in num_set:
                    current_num += 1
                    current_streak += 1

                longest_streak = max(longest_streak, current_streak)

        return longest_streak

```

300. 最长递增子序列

给你一个整数数组 `nums`，找到其中最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。

示例 1:

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。

思路：定义 $dp[i]$ 为前 i 个元素中，以 i 为数字结尾的最长上升序列的长度。因此我们可以从小到大计算 dp 的值。在计算 $dp[i]$ 之前，已经算出了 $dp[0, \dots, i-1]$ 的值，则状态转移方程为

$$dp[i] = \max(dp[j]) + 1$$

其中 $0 \leq j < i$ 并且 $nums[j] < nums[i]$ 。最后整个上升子序列的最大值为：

$$LIS_{length} = \max(dp[i]), \text{ 其中 } 0 \leq i < n$$

```

class Solution:
    """
    思路一：比较简单，动态规划，O(n^2), 从后往前遍历
    思路二：
    """
    def lengthOfLIS(self, nums):

```

```

length = len(nums)

dis = [1] * length
dis[0] = 1
max_len = 1

for i in range(1, length):
    j = i - 1
    while j >= 0:
        if nums[i] > nums[j]:
            dis[i] = max(dis[j]+1, dis[i])
        j -= 1
    if dis[i] > max_len:
        max_len = dis[i]

return max_len

```

1143. 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长公共子序列的长度。

一个字符串的 *子序列* 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1:

输入: `text1 = "abcde"`, `text2 = "ace"`
 输出: 3
 解释: 最长公共子序列是 `"ace"`，它的长度为 3。

思路: 建立一个二维数组 $dp[m][n]$ 。

边界条件

- 1、两个都是空串, $dp[0][0] = 0$
- 2、其中一个为空串, $dp[i][0] = 0$ or $dp[0][j] = 0$ 。

如果不是空串, 有两种情况

一种是 $text[i] \neq text[j]$, 此时状态转移方程为 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

当 $text[i] = text[j]$ 时, 状态方程为 $dp[i][j] = dp[i-1][j-1] + 1$ 。

代码:

```

class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        m, n = len(text1), len(text2)
        dp = [[0] * (n+1) for _ in range(m+1)]

        for i in range(m+1):
            dp[i][0] = 0

```

```

for j in range(n+1):
    dp[0][j] = 0

for i in range(1, m+1):
    for j in range(1, n+1):
        if text1[i-1] == text2[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

return dp[m][n]

```

路径类问题

路径类的问题，通常有个很明显的特点，就是从左上往右下慢慢异动，通常考虑最短路径或者路径数。思路基本是一样的。需要注意初始化边界。

[62. 不同路径](#)

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

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

问总共有多少条不同的路径？

示例 1:



输入: $m = 3, n = 2$

输出: 3

解释:

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

思路：动态规划，这类路径类的问题，有个最大

```

class Solution:
    def uniquePaths(self, m, n):

        res = [(n * [0]) for i in range(m)]
        for i in range(m):
            res[i][0] = 1

```

```

for j in range(n):
    res[0][j] = 1

for i in range(1, m):
    for j in range(1, n):
        res[i][j] = res[i][j-1] + res[i-1][j]

return res[m-1][n-1]

```

64. 最小路径和

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

说明：每次只能向下或者向右移动一步。

示例 1：

| | | |
|---|---|---|
| 1 | 3 | 1 |
| 1 | 5 | 1 |
| 4 | 2 | 1 |

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

输出: 7

解释: 因为路径 1→3→1→1→1 的总和最小。

思路：动态规划

$$\min_dist[i][j] = \min(\min_dist[i-1][j], \min_dist[i][j-1]) + dis[i][j]$$

```

class Solution:
    def minPathSum(self, grid):
        m = len(grid)
        n = len(grid[0])

        if m == 0:
            return 0

        # 这种写法存在浅拷贝的坑
        # min_path = [[0] * n] * m

        min_path = [[0] * n for i in range(m)]
        # 初始化第一行与第一列
        init_dis = 0
        for i in range(m):
            init_dis += grid[i][0]
            min_path[i][0] = init_dis

```



```

init_dis = grid[0][0]
for i in range(1, n):
    init_dis += grid[0][i]
    min_path[0][i] += init_dis

for i in range(1, m):
    for j in range(1, n):
        min_path[i][j] = min(min_path[i-1][j], min_path[i][j-1]) +
grid[i][j]

return min_path[m-1][n-1]

```

斐波那契类

$$f[0] = f[1] = 1$$

$$f[n] = f[n-1] + f[n-2], n \geq 2$$

[70. 爬楼梯](#)

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

```

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

```

进阶：剑指offer超级爬楼梯

$$f[0] = 1$$

$$f[1] = 1$$

$$f[2] = 2$$

$$f(n) = \sum_{i=1}^n f(i)$$

[55. 跳跃游戏](#)

给定一个非负整数数组 `nums`，你最初位于数组的 **第一个下标**。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1：

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 1 步, 从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。

思路: 动态规划, 关键点, 计算每个位置能够跳到的最远位置

$$right = \max(right, i + \text{nums}[i])$$

```
class Solution:
    """
    关键点, index+ value
    计算每个位置能够到达的最远位置
    """

    def canJump(self, nums):
        count = len(nums)
        flag = [False] * count
        flag[0] = True
        i = 0
        right_most = 0
        while i < count:
            if i <= right_most:
                right_most = max(right_most, i+nums[i])
                if right_most >= count - 1:
                    return True
            i += 1

        return False
```

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。

示例 1:

输入: $n = 12$

输出: 3

解释: $12 = 4 + 4 + 4$

思路: 动态规划, 和前面哈希的题目优点类似。

```
import math
import sys

class Solution:
    """
    动态规划
    """
```

```
def numSquares(self, n: int) -> int:

    square_nums = [i**2 for i in range(int(math.sqrt(n))+1)]
    dp = [sys.maxsize] * (n+1)
    dp[0] = 0

    for i in range(1, n+1):
        # 从前到后遍历完全平方数
        for square in square_nums:
            # 完全平方数大于目标和，直接跳过
            if i < square:
                break
            # 更新当前所需要完全平方数最小值。
            dp[i] = min(dp[i], dp[i-square]+1)
    return 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。

示例 1:

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

```
class Solution:
    def coinChange(self, coins, amount):
        import sys
        n = len(coins)
        dp = [sys.maxsize] * (amount+1)
        dp[0] = 0

        for i in coins:
            for j in range(i, amount+1):
                dp[j] = min(dp[j], dp[j-i]+1)

        return dp[amount] if dp[amount] != sys.maxsize else -1
```

编辑距离

72. 编辑距离](<https://leetcode-cn.com/problems/edit-distance/>)

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

示例 1:

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')

思路: 动态规划, 关键点, 空字符串是能够匹配的。

1、定义二维数组, dp , $dp[i][j]$ 表示word1中前*i*个字符换到word2中前*j*个字符需要的最短路径。

2、初始化, 当word1 或者word2位空串的时候, 需要插入对应的元素数。

3、状态转移方程

其实对于编辑距离问题, 可以将其中一个字符固定, 只对另一个进行操作。因为在word1中增加一个字符和在word2中删除一个字符其实是等价的。


- 增, 相当于word1不动, word2删除一个。 $dp[i][j] = dp[i][j - 1] + 1$
- 删, word1删除一个, word2不动。 $dp[i][j] = dp[i - 1][j] + 1$
- 改, word1和word2都不动。 $dp[i][j] = dp[i - 1][j - 1] + 1$

所以最短编辑距离为

$$dp[i][j] = \min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1$$

| | | word2 | | | |
|-------|---|-------|---|---|---|
| | | | r | o | s |
| word1 | | 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 |

这是啥?



```
class Solution:
    def minDistance(self, word1: str, word2: str):
        m = len(word1)
        n = len(word2)

        distance = [[0] * (n+1) for i in range(m+1)]

        # 初始化
        for i in range(m+1):
            distance[i][0] = i
        for i in range(n+1):
            distance[0][i] = i
```

```

    for i in range(1, m+1):
        for j in range(1, n+1):
            if word1[i-1] == word2[j-1]:
                distance[i][j] = distance[i-1][j-1]
            else:
                distance[i][j] = min(distance[i-1][j], distance[i][j-1],
distance[i-1][j-1]) + 1

    return distance[m][n]

```

买卖股票

[121. 买卖股票的最佳时机](#)

给定一个数组 `prices`，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0`。

示例 1:

输入: `[7,1,5,3,6,4]`

输出: `5`

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6 - 1 = 5。

注意利润不能是 7 - 1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

思路：只买一次，然后卖出。所以选择最低点`min_price`买入，然后在低点之后的搞点卖出就行了。。

```

class Solution:
    def maxProfit(self, prices):
        max_profit = 0

        min_price = prices[0]

        for i in range(len(prices)):
            if prices[i] < min_price:
                min_price = prices[i]

            if prices[i] - min_price > max_profit:
                max_profit = prices[i] - min_price

        return max_profit

```

[122. 买卖股票的最佳时机 II](#)

给定一个数组，它的第 `i` 个元素是一支给定股票第 `i` 天的价格。

设计一个算法来计算你能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：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）。

示例 1:

输入: [7,1,5,3,6,4]

输出: 7

解释: 在第 2 天 (股票价格 = 1) 的时候买入, 在第 3 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 = $5-1=4$ 。

随后, 在第 4 天 (股票价格 = 3) 的时候买入, 在第 5 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = $6-3=3$ 。

思路一: 贪心, 找到上升的趋势, 计算结果。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:

        max_profit = 0

        for i in range(1, len(prices)):
            max_profit += max(0, prices[i]-prices[i-1])

        return max_profit
```

309. 最佳买卖股票时机含冷冻期

难度中等698收藏分享切换为英文接收动态反馈

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)。

示例:

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]

思路: 由于存在冷冻期, 所以有三种状态

- $f[i][0]$ 表示当前拥有一支股票, 对应的最大累计收益
- $f[i][1]$ 表示当前不持有任何股票且处在冷冻期对应的最大收益
- $f[i][2]$ 表示当前不持有任何股票且没有处在冷冻期对应的最大收益

注意 f 表示是当天收盘结果。

对于一个当天结束持有股票, 其持有的股票可以是前一天持有的没有卖出, 也可以是当天买入的(前一天为非冷冻期), 因此最大收益 $f[i][0]$ 为

$$f[i][0] = \max(f[i-1][0], f[i-1][2] - \text{prices}[i])$$

前一天有股票, 今天卖出了, 因此最大收益 $f[i][1]$ 为

$$f[i][1] = f[i][0] + \text{prices}[i]$$

$i-1$ 天卖出股票或者 $i-2$ 天股票为空, 都能使当天不持有任何股票, 其最大收益 $f[i][2]$ 为:

$$f[i][2] = \max(f[i-1][1], f[i-2][2])$$

最后求三种状态的最大值，由于空仓收益更高，其实看两种状态就行了。

$$\max_profit = \max(f[n-1][1], f[n-1][2])$$

```
class Solution:
    """
    动态规划：一共三种状态
    1、当前持有一直股票，有股票且没有卖
    2、当前不持有股票且处在冷冻期
    3、当前不持有股票且不处在冷冻期，即没有股票或者有股票卖了
    """
    def maxProfit(self, prices):

        n = len(prices)

        if n == 0:
            return 0

        f = [[0] * 3 for i in range(n)]
        f[0][0] = -prices[0]

        for i in range(1, n):
            # 关键点i表示今天结束的状态，也就是买入卖出操作执行结束
            # 前一天有股票，今天不属于冷冻期
            # 当前最大收益的状态时卖出或者保持不变
            f[i][0] = max(f[i-1][0], f[i-1][2] - prices[i])
            # 当前一天不持有股票且处在冷冻期
            f[i][1] = f[i-1][0] + prices[i]
            # 注意冷冻期是指
            # 当前不持有股票且不再冷冻期，有以下几种情况
            # 前一天有股票，且没有卖
            # 股票已经卖完了
            f[i][2] = max(f[i-1][1], f[i-1][2])

        return max(f[n-1][1], f[n-1][2])
```

相似题目：

[188. 买卖股票的最佳时机 IV](#)

小偷问题

[198. 打家劫舍](#)

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

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

输入：[1,2,3,1]

输出：4

解释：偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。

偷窃到的最高金额 = 1 + 3 = 4。

```

class Solution:
    """和309一个类型的题目
    判断每个状态偷还是不偷
    """
    def rob(self, nums):

        n = len(nums)

        if n == 0 or n == 1:
            return sum(nums)
        f = [[0] * 2 for _ in range(n)]
        f[0][0] = 0 # 不偷
        f[0][1] = nums[0] # 偷

        for i in range(1, n):
            f[i][0] = max(f[i-1][1], f[i-1][0])
            f[i][1] = f[i-1][0] + nums[i]

        return max(f[n-1][0], f[n-1][1])

```

213. 打家劫舍 II

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 **围成一圈**，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

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

示例 1:

输入: nums = [2,3,2]

输出: 3

解释: 你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

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。

示例 1:

输入: [3,2,3,null,3,null,1]



输出: 7

解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.


```

class Solution:
    """
    思路一: dfs+深度优先计算
    """

    def rob(self, root) -> int:

        def dfs(root):

            if not root:
                return 0, 0 # 投、不偷

            left = dfs(root.left)
            right = dfs(root.right)
            rob_value = left[1] + right[1] + root.val
            skip_value = max(left[0], left[1]) + max(right[0], right[1])
            return [rob_value, skip_value]

        value = dfs(root)

        return max(value[0], value[1])

```

矩阵问题

[240. 搜索二维矩阵 II](#)

编写一个高效的算法来搜索 $m \times n$ 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。

示例 1:

| | | | | |
|----|----|----|----|----|
| 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
```

思路：和剑指offer二维数组查找差不多。选择右上或者左下遍历。以右上角为例，对于矩阵中的点 $matrix[i][j]$ ，其左上角的点的值都小于 $matrix[i][j]$ ，选择左下角和右上角的目的是可以沿着边走，找到最后的结果。

```
import numpy as np

class Solution:
    """
    剑指offer原题：二维数组查找
    选择右上角往左下角移动，右上角和左下角的特点为
    """
    def searchMatrix(self, matrix, target):
        matrix = np.array(matrix)
        m = matrix.shape[0]
        n = matrix.shape[1]

        flag = False

        i, j = 0, n-1
        while i < m and j >= 0:
            if matrix[i][j] == target:
                flag = True
                break
            elif matrix[i][j] < target:
                i += 1
            else:
                j -= 1

        return flag
```

[221. 最大正方形](#)

难度中等701收藏分享切换为英文接收动态反馈

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。

示例 1：

| | | | | |
|---|---|---|---|---|
| 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: 4

思路: 构建状态矩阵, $dp[i][j]$ 表示以 (i, j) 为右下角且只包含1的正方形边长最大值。

对于每个位置 (i, j) , 如果该位置的值为0, 则 $dp[i][j] = 0$, 因为当前位置不可能出现在1组成的正方形中

如果当前位置是1, 状态方程为

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1$$

原始矩阵

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 |

3×3 表示 $dp[2][3]$

2×2 表示 $dp[3][4]$

1×1 表示 $dp[4][2]$

dp

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 2 | 2 | 0 |
| 2 | 0 | 1 | 2 | 3 | 1 |
| 3 | 0 | 1 | 2 | 3 | 2 |
| 4 | 0 | 0 | 1 | 2 | 3 |

$$dp(2, 3) = \min(dp(1, 3), dp(1, 2), dp(2, 2)) + 1 = 3$$

$$dp(3, 4) = \min(dp(2, 4), dp(2, 3), dp(3, 3)) + 1 = 2$$

$$dp(4, 2) = \min(dp(3, 2), dp(3, 1), dp(4, 1)) + 1 = 1$$

```
class Solution:
    def maximalSquare(self, matrix) -> int:

        if len(matrix) == 0 or len(matrix[0]) == 0:
            return 0

        max_len = 0
```

```

row, col = len(matrix), len(matrix[0])

dp = [[0] * col for _ in range(row)]
for i in range(row):
    for j in range(col):
        if matrix[i][j] == '1':
            if i == 0 or j == 0:
                dp[i][j] = 1
            else:
                dp[i][j] = min(dp[i][j-1], dp[i-1][j-1], dp[i-1][j]) + 1
            max_len = max(dp[i][j], max_len)

max_square = max_len * max_len

return max_square

```

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。

示例 1：

输入： $s = \text{"leetcode"}$, $wordDict = [\text{"leet"}, \text{"code"}]$
 输出：true
 解释：返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

思路：

1、创建一个dp，初始化 $dp[0] = \text{True}$ ，表示串空且合法

2、对于字符串 $s[0 \dots i - 1]$ ，我们只需要判断前面的单词和最后一个单词是否合法，假设分割点为 j ，因此只需要判断 $s[0 : j]$ 和 $s[j + 1 : i - 1]$ 是否合法就行了，因此状态转移方程为

$$dp[i] = dp[j] \quad \& \quad s[j:i] \text{ in word}$$

```

class Solution:
    # 动态规划
    def wordBreak(self, s: str, wordDict):
        word_set = set(wordDict)
        length = len(s)
        flag = [False] * (length+1)
        flag[0] = True # 空串为合法

        for i in range(1, length+1):
            for j in range(0, i):
                if s[j:i] in word_set and flag[j]:
                    flag[i] = True

        return flag[length]

```

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。

```
class Solution:
    """
    两个数组，最大值和最小值
    """
    def maxProduct(self, nums):
        n = len(nums)
        max_res = [1] * (n+1)
        min_res = [1] * (n+1)

        for i in range(1, n+1):
            max_res[i] = max(max_res[i-1] * nums[i-1], min_res[i-1] * nums[i-1],
                             nums[i-1])
            min_res[i] = min(max_res[i-1] * nums[i-1], min_res[i-1] * nums[i-1],
                             nums[i-1])

        return max(max_res[1:])
```

背包问题

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。

注意：

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

示例 1：

输入：[1, 5, 11, 5]

输出：true

解释：数组可以分割成 [1, 5, 5] 和 [11]。

思路：对于这个问题，如果数组长度小于2，无法分成两个数组，如果数组和sum为奇数，那么一定无法分成两个，如果最大元素max_num > sum//2，也无法分成两个相等的数组。

创建二维数组dp，包含n行，target+1列，其中 $dp[i][j]$ 表示从数组[0, i]下标范围内选择若干个正整数，是否存在一种方案使得被选的正整数等于j。初始化时dp元素全部为false。

边界条件

- $dp[i][0] = true$ ，不选择任何正整数或者选择的正整数等于0
- $dp[0][nums[i]] = true$ ，当只有一个正整数可以选择时。

在确定 $dp[i][j]$ 的时候，需要确认下面两种情况。

1、 $j \geq \text{nums}[i]$ 时，对于当前的数字 $\text{nums}[i]$ ，有两种情况，可以取，可以不取，两种情况只要有一种为 $true$ ，就有 $dp[i][j] = true$ 。

- 不选 $\text{nums}[i]$, $dp[i][j] = dp[i - 1][j]$
- 选择 $\text{nums}[i]$, $dp[i][j] = dp[i - 1][j - \text{nums}[i]]$

2、对于 $j < \text{nums}[i]$, $dp[i][j] = dp[i - 1][j]$

```
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        n = len(nums)
        if n < 2:
            return False

        total = sum(nums)
        maxNum = max(nums)
        if total & 1:
            return False

        target = total // 2
        if maxNum > target:
            return False

        dp = [[0] * (target + 1) for _ in range(n)]
        for i in range(n):
            # 如果不选择任何整数，则被选择的正整数等于0
            dp[i][0] = True

        dp[0][nums[0]] = True
        for i in range(1, n):
            num = nums[i]
            for j in range(1, target + 1):
                if j >= num:
                    dp[i][j] = dp[i - 1][j] | dp[i - 1][j - num]
                else:
                    dp[i][j] = dp[i - 1][j]

        return dp[n - 1][target]
```

回溯

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。

回溯算法的基本思想是：从一条路往前走，能进则进，不能进则退回来，换一条路再试。

一般采用dfs

最优解

17. 电话号码的字母组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

输入: digits = "23"

输出: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

思路: 回溯, dfs

关键在于终止条件, 以及如何继续想下遍历

```
class Solution:
    """
    常规题, 采用dfs
    """
    def letterCombinations(self, digits: str):

        num_dict = {'2': 'abc',
                    '3': 'def',
                    '4': 'ghi',
                    '5': 'jkl',
                    '6': 'mno',
                    '7': 'pqrs',
                    '8': 'tuv',
                    '9': 'wxyz'}

        res = []
        n = len(digits)

        if n == 0:
            return res

        str_list = [num_dict[i] for i in digits]

        def dfs(str_list, s, cur):
            nonlocal res
            if cur == n:
                res.append(s)
            else:
                for c in str_list[cur]:
                    s += c
                    dfs(str_list, s, cur+1)
                    s = s[:cur]

        dfs(str_list, "", 0)
        return res
```

22. 括号生成

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

示例:

输入: $n = 3$

输出: ["((()))","(()())","(())()","()(())","()()()"]

```

class Solution:
    def generateParenthesis(self, n: int):
        res = []

        def dfs(str, left, right):

            if left == 0 and right == 0:
                res.append(str)

            if right < left:
                return

            if left > 0:
                dfs(str+'(', left-1, right)
            if right > 0:
                dfs(str+')', left, right-1)
        dfs("", n, n)
        return res

```

39. 组合总和

给定一个无重复元素的数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

所有数字（包括 target）都是正整数。解集不能包含重复的组合。示例 1：

```

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

```

思路：dfs，不断遍历不同的情况。

关键点：

- 终止条件
- 遍历是是否能够重复选取

```

class Solution:
    def __init__(self):
        self.res_all = []

    def dfs(self, candidates, res, target):

        if sum(res) == target:
            self.res_all.append(res)
            return
        elif sum(res) > target:
            return

        for i in range(len(candidates)):

```



```

        self.dfs(candidates[i:], res + [candidates[i]], target)

    def combinationSum(self, candidates, target):

        self.dfs(candidates, [], target)

        return self.res_all

```

排列问题

46. 全排列

给定一个 **没有重复** 数字的序列，返回其所有可能的全排列。

示例:

```

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

```

```

class Solution:
    def permute(self, nums):
        m = len(nums)
        visited = [False] * (m+1)
        res = []

        def dfs(linklist, n):
            if n == 0:
                res.append(linklist)

            for i in range(1, m+1):
                if not visited[i]:
                    visited[i] = True
                    dfs(linklist + [nums[i-1]], n - 1)
                    visited[i] = False

        dfs([], m)
        return res

```

78. 子集

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1:

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

思路: 回溯

```
class Solution:
    """
    第一反应是回溯
    """
    def subsets(self, nums):

        count = len(nums)
        res_all = []

        def dfs(data, target, res):

            if target == 0:
                res_all.append(res.copy())
                return

            for i in range(len(data)):
                res.append(data[i])
                dfs(data[i+1:], target-1, res)
                res.pop()

        res_all.append([])

        for i in range(1, count+1):
            dfs(nums, i, [])

        return res_all
```

[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。

示例：

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。

```
class Solution:
```

```

"""
思路: dfs+剪枝, Java版通过, python版超时
"""

def findTargetSumWays(self, nums, S):

    count = 0
    n = len(nums)

    def dfs(nums, start, target, S):
        nonlocal count, n
        if start == n:
            if target == S:
                count += 1
        else:
            if start < len(nums):
                dfs(nums, start+1, target+nums[start], S)
                dfs(nums, start+1, target-nums[start], S)

    if S > sum(nums) or S < -sum(nums):
        return count
    else:
        dfs(nums, 0, 0, S)

    return count

```

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

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

双指针

11.盛水最多的容器

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

输入: $[1, 8, 6, 2, 5, 4, 8, 3, 7]$

输出: 49

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

<https://leetcode-cn.com/problems/container-with-most-water/>

思路：这是一道典型的双指针题目，题目意思是利用从数组中选择两个位置，当做挡板，看中间能够容纳多少水。所以思路很简单，建立一个双指针，从左右往中间移动，每次移动左右指针中值较小的那个，每次更新的时候判断面积是否大于最大面积，如果大于，就更新。

```

class Solution:
    def maxArea(self, height) -> int:

```

```

n = len(height)
max_area = 0
l, r = 0, n-1

while l < r:
    area = min(height[l], height[r]) * (r-l)
    max_area = max(area, max_area)
    if height[l] <= height[r]:
        l += 1
    else:
        r -= 1

return max_area

```

15. 三数之和

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

注意：答案中不可以包含重复的三元组。

输入: `nums = [-1,0,1,2,-1,-4]`
 输出: `[[-1,-1,2],[-1,0,1]]`

思路：排序+双指针变体

设置三个指针，首先将数组从小到大进行排序，然后从前往后一次选择一个数作为三元组的第一个数，然后设置两个左右指针，判断节点是否小于0，如果小于0，左指针往右移动，如果大于0，右指针往左移动。每一次保存上一次满足条件的结果，然后下一次满足条件时与上一次结果对比，如果不一样，保存结果。

```

class Solution:
    def threeSum(self, nums):

        nums.sort()
        count = len(nums)
        result = []
        # 保留上一次结果，做去重使用
        first = -1
        second = -1
        third = -1
        for i in range(count-1):
            j = i + 1 # 左指针
            k = count - 1

            while j < k:
                target = nums[i] + nums[j] + nums[k]
                if target < 0:
                    j += 1
                elif target > 0:
                    k -= 1
                else:
                    # 判断是否已经存在改结果
                    if third == -1 or (nums[j] > nums[second] or nums[k] <
nums[third]
                                or nums[i] > nums[first]):
                        # 第一次满足条件

```

```

        result.append([nums[i], nums[j], nums[k]])
        first = i
        second = j
        third = k

        j += 1

    return result

```

19. 删除链表的倒数第 N 个结点

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

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

思路：双指针，一个先向后移动 n 步，然后两个指针一起移动。当前面一个移动到第 n 个时，后面的指针则指向第 k 个结点。

```

def get_length(head):
    # 获取链表长度
    p = head
    count = 0
    while p is not None:
        count += 1
        p = p.next

    return count

class Solution:
    def removeNthFromEnd(self, head, n):
        m = get_length(head)

        # 返回头结点
        if m == n:
            return head.next
        k = m - n - 1
        p = head
        for i in range(k):
            p = p.next

        # 删除第n个结点
        q = p.next
        if q is not None:
            p.next = q.next
        else:
            p.next = q

        return head

```

31. 下一个排列

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

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须 [原地](#) 修改，只允许使用额外常数空间。

示例：

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

思路：双指针

注意到下一个排列总是比当前排列要大，除非该排列已经是最大的排列。我们希望找到一种方法，能够找到一个大于当前序列的新序列，且变大的幅度尽可能小。具体地：

我们需要将一个左边的「较小数」与一个右边的「较大数」交换，以能够让当前排列变大，从而得到下一个排列。

从有往左遍历。找到第一个 $nums[i] < nums[i+1]$ 的数，此时可以将数组分成两个部分， $nums[0, i]$, $nums[i+1, n-1]$ 。

可以知道此时 $nums[i+1, n-1]$ 此时是降序。交换 $nums[i]$ 和 $nums[j]$ ，然后将 $nums[i+1, n-1]$ 进行升序排序就能得到结果

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

        i = len(nums) - 2
        while i >= 0:
            if nums[i] < nums[i+1]:
                break
            i -= 1

        if i < 0:
            nums.reverse()
        else:
            j = len(nums) - 1
            while j > i:
                if nums[j] > nums[i]:
                    nums[j], nums[i] = nums[i], nums[j]
                    break
            j -= 1
            left = i + 1
            right = len(nums) - 1
            while left < right:
                nums[left], nums[right] = nums[right], nums[left]
                left += 1
                right -= 1

        return nums
```

[75. 颜色分类](#)

难度中等802收藏分享切换为英文接收动态反馈

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

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

示例 1:

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

思路：双指针

```
class Solution:
    """
    思路一: sort函数
    思路二: 单指针，第一次遍历，放0的位置，第二次遍历，放一的位置
    思路二: 双指针，如果是0异动到前面，2移动到后面
    """
    def sortColors(self, nums):
        """
        Do not return anything, modify nums in-place instead.
        """
        length = len(nums)
        p, q = 0, length - 1
        i = 0
        while i <= q:
            if nums[i] == 0:
                nums[p], nums[i] = nums[i], nums[p]
                p += 1

            if nums[i] == 2:
                nums[q], nums[i] = nums[i], nums[q]
                q -= 1

            i += 1
```

[581. 最短无序连续子数组](#)

难度中等493收藏分享切换为英文接收动态反馈

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。

示例 1:

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。

思路：排序+双指针

首先使用快排对结果进行排序，然后设置头指针，和尾指针

- 头指针，找到第一个和target位置不一样的元素。

- 尾指针，找到最后一个和target位置不一样的元素。

```
class Solution:
    def findUnsortedSubarray(self, nums):
        target = sorted(nums)

        start = len(nums)
        end = 0
        max_len = 0
        for i in range(len(nums)):

            if nums[i] != target[i]:
                start = min(i, start)
                end = max(i, end)

            if start > end:
                max_len = 0
            else:
                max_len = end - start + 1

        return max_len
```

移动数组

[283. 移动零](#)

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

```
class Solution:
    """
    思路一：将
    """
    def moveZeroes(self, nums):
        """
        Do not return anything, modify nums in-place instead.
        """
        index = 0
        count = len(nums)
        for i in range(count):
            if nums[i] != 0:
                nums[index] = nums[i]
                index += 1
        while index < count:
            nums[index] = 0
            index += 1

        return nums
```

[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，也会被视为不同的子串。

示例 1:

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

示例 2:

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

思路: 双指针, 以每个字符作为中心节点, 从左右往两边扩展,

```
class Solution:
    """从每一个中间往两边遍历"""
    def countSubstrings(self, s: str):

        ans = 0
        n = len(s)
        for i in range(2*n):
            l = int(i/2)
            r = l + i % 2
            while l >= 0 and r < n and s[l] == s[r]:
                ans += 1
                l -= 1
                r += 1

        return ans
```

快慢指针

主要可以用来判断链表是否存在环, 设置两个指针, 快指针fast和慢指针slow。fast指针每次移动两步, slow指针每次移动一步。如果存在环, 那么他们一定会相遇。

[141. 环形链表](#)

给定一个链表, 判断链表中是否有环。

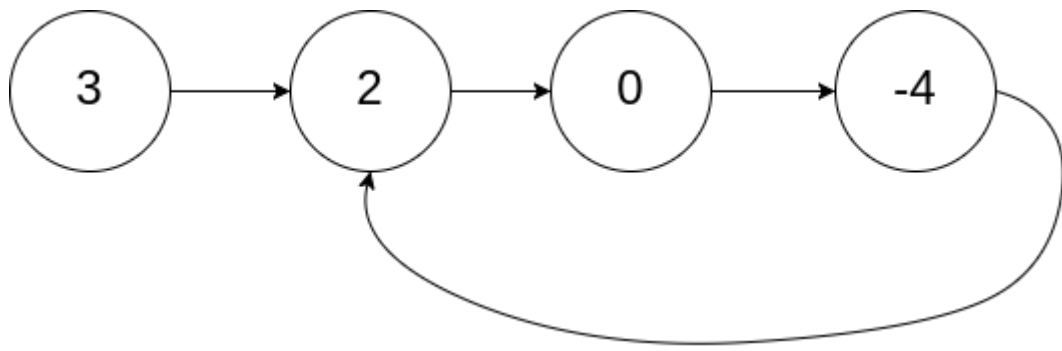
如果链表中有某个节点, 可以通过连续跟踪 `next` 指针再次到达, 则链表中存在环。为了表示给定链表中的环, 我们使用整数 `pos` 来表示链表尾连接到链表中的位置 (索引从 0 开始)。如果 `pos` 是 `-1`, 则在该链表中没有环。**注意: `pos` 不作为参数进行传递**, 仅仅是为了标识链表的实际情况。

如果链表中存在环, 则返回 `true`。否则, 返回 `false`。

进阶:

你能用 $O(1)$ (即, 常量) 内存解决此问题吗?

示例 1:



输入: head = [3,2,0,-4], pos = 1

输出: true

解释: 链表中有一个环, 其尾部连接到第二个节点。

思路: 设置快慢指针, 相交则说明有环。

```
class Solution:
    # 基础题, 以前做过, 快慢指针
    def hasCycle(self, head):

        if not head or not head.next:
            return False

        low = head
        high = head.next

        flag = False

        while low and high:

            if low == high:
                return True
            low = low.next
            high = high.next
            if high:
                high = high.next

        return False
```

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`。

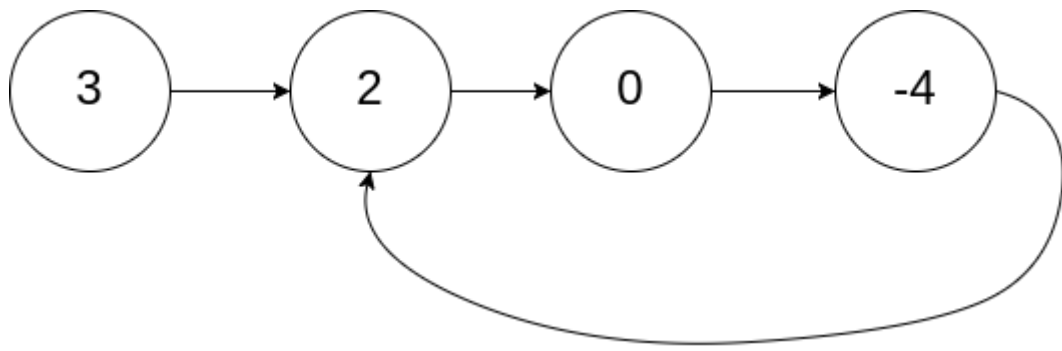
为了表示给定链表中的环, 我们使用整数 `pos` 来表示链表尾连接到链表中的位置 (索引从 0 开始)。如果 `pos` 是 `-1`, 则在该链表中没有环。注意, `pos` 仅仅是用于标识环的情况, 并不会作为参数传递到函数中。

说明: 不允许修改给定的链表。

进阶:

- 你是否可以使用 `O(1)` 空间解决此题?

示例 1:



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。

```
class solution:
    def detectCycle(self, head):

        if not head:
            return None

        point_slow = head
        point_fast = head
        count = 0

        while point_slow and point_fast:

            if point_slow == point_fast and count != 0:
                break

            point_slow = point_slow.next

            point_fast = point_fast.next
            if point_fast:
                point_fast = point_fast.next

            count += 1

        # 如果不存在环
        if not point_slow or not point_fast:
            return None

        point_fast = head
        while point_fast != point_slow:
            point_fast = point_fast.next
            point_slow = point_slow.next
        return point_fast
```

滑动窗口

滑动窗口问题算是双指针的一种变形

3. 无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。

示例 1:

输入: s = "abcabcbb"

输出: 3

思路: 设置首尾指针, 都从0开始, 然后尾指针end_index每向后移动一位, 判断当前的字符是否在滑窗里面, 如果不在

将尾指针字符加入滑窗, 如果在, 头指针向后移动一位, 加入尾指针, 然后更新滑窗的长度。如果当前长度大于最大长

, 则更新最长子串。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        start_index = 0
        end_index = start_index
        max_len = 0
        n = len(s)

        while start_index < n and end_index < n:

            if end_index == 0:
                max_len = 1
            if s[end_index] not in s[start_index: end_index]:
                end_index += 1
            else:
                start_index += 1
            if end_index - start_index > max_len:
                max_len = end_index - start_index

        return max_len
```

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 |

思路：

一、暴力求解，直接遍历每个元素组成的不同滑动窗口。超时

二、双端队列+滑动窗口

利用双端队列实现单调队列，每次里面保存最大的结果的索引，这样做的好处是即能取到对应的值，也可以通过index来判断当前元素距离队列头部保存index的距离，从而得到当前滑窗滑窗的大小。

```
from collections import deque
class Solution:
    """
    思路一：暴力求解，直接找出方框里面最大的值，基本超时
    思路二：双端队列，其实就是利用双端队列实现最大堆，每次在结果里面保存可能最大的值
    """
    def maxSlidingWindow(self, nums, k):

        n = len(nums)

        q = deque()
        res = []
        for i in range(k):
            if not q or nums[i] <= nums[q[-1]]:
                q.append(i)
            else:
                while q and nums[i] > nums[q[-1]]:
                    q.pop()
                q.append(i)

        res.append(nums[q[0]])
        for i in range(k, n):
            while q and nums[i] > nums[q[-1]]:
                q.pop()
            q.append(i)
            if i - q[0] == k:
                q.popleft()

            res.append(nums[q[0]])
        return res
```

76. 最小覆盖子串

难度困难987收藏分享切换为英文接收动态反馈

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 `""`。

注意：如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

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

思路：滑动窗口

```
class Solution:
    def minWindow(self, s: str, t: str) -> str:
```

```

from collections import defaultdict
hash_table = defaultdict(int)
for c in t:
    hash_table[c] += 1

start = 0
end = 0
min_len = float('inf') # 包含子串的最小长度
count = 0 # 用于记录当前滑动窗口包含目标字符的个数，当count = len(t)，t为子串
res = ''
while end < len(s):
    # 当前元素在子串中，包含子串字符长度+1
    # 同时对应子串个数应该-1，目的是为了防止同一个字符重复使用
    if hash_table[s[end]] > 0:
        count += 1
    hash_table[s[end]] -= 1
    end += 1
    while count == len(t):
        if min_len > end - start:
            min_len = end - start
            res = s[start: end]
        # 如果头部不在子串中，则包含子串长度-1
        if hash_table[s[start]] == 0:
            count -= 1
        hash_table[s[start]] += 1
        start += 1

return res

```

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。

示例 1:

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 |

```

from collections import deque

```

```

class Solution:

```

```

    """

```

思路一：暴力求解，直接找出方框里面最大的值，基本超时

思路二：双端队列，其实就是利用双端队列实现最大堆，每次在结果里面保存可能最大的值

```
"""
def maxSlidingWindow(self, nums, k):

    n = len(nums)

    q = deque()
    res = []
    # 保存前k个最大值
    for i in range(k):
        if not q or nums[i] <= nums[q[-1]]:
            q.append(i)
        else:
            while q and nums[i] > nums[q[-1]]:
                q.pop()
            q.append(i)
    # 弹出第一个最大值
    res.append(nums[q[0]])
    for i in range(k, n):
        # 如果当前元素大于队列顶部元素，弹出，然后加入新的新的元素
        while q and nums[i] > nums[q[-1]]:
            q.pop()
        q.append(i)
        # 如果队列已经满了，弹出
        if i - q[0] == k:
            q.popleft()
        # 每次遍历时加入最大结果
        res.append(nums[q[0]])
    return res
```

类似题目：

3. 无重复字符的最长子串
4. 串联所有单词的子串
5. 最小覆盖子串
6. 至多包含两个不同字符的最长子串
7. 长度最小的子数组
8. 滑动窗口最大值
9. 字符串的排列
10. 最小区间
11. 最小窗口子序列

栈

括号问题

借助栈的特性，来判断括号是否合法

[20. 有效的括号](#)

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

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

思路：借助辅助栈，如果当前字符是左括号，入栈。如果当前字符是右括号，出栈。如果栈空，则表明都是合法的，如果非空，表示不合法。

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = list()
        lens = len(s)
        for i in range(lens):
            if len(stack) == 0:
                stack.append(s[i])
            else:
                if (s[i] == ')' and stack[-1] == '(') or (s[i] == '}' and
stack[-1] == '{') or \
                    (s[i] == ']' and stack[-1] == '['):
                    stack.pop()
                else:
                    stack.append(s[i])

        return len(stack) == 0
```

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。

示例：

输入：s = "()"
输出：2
解释：最长有效括号子串是 "()"

思路：借助栈，由于字符串只包含两种字符，因此对于左括号进展，对于右括号，出栈时，当前元素和栈顶元素的差则为有效括号的长度，不断更新有效括号，就能得到最后结果。

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:

        stack = []
        max_len = 0
        length = 0

        if len(s) == 0:
            return 0

        for i in range(len(s)):

            if not stack or s[i] == '(' or s[stack[-1]] == ')':
                stack.append(i)
            else:
                stack.pop()
                length = i - (stack[-1] if stack else -1)

            max_len = max(max_len, length)
```



```
return max_len
```

56. 合并区间

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

示例 1:

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

思路: 排序+栈

```
class Solution:
    """
    借助栈
    """
    def merge(self, intervals):

        intervals = sorted(intervals, key=lambda x: (x[0], x[1]))

        stack = []
        for i in intervals:
            if stack and stack[-1][1] >= i[0]:
                cur = stack.pop()
                union = [min(i[0], cur[0]), max(i[1], cur[1])]
                stack.append(union)
            else:
                stack.append(i)

        return stack
```

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。

示例 1:

输入: `s = "3[a]2[bc]"`
输出: `"aaabcbc"`

思路: 采用两个栈数字占和字符栈，算法计算步骤

- 将数字和字母分别保存
- 遇到 '['，入栈
- 遇到 ']'，出栈，计算当前结果
- 然后将结果拼接到字符串后面。

```
class Solution:
    """
```

采用栈： 输入栈和输出栈

"""

```
def decodeString(self, s: str) -> str:
    num_stack = []
    str_stack = []
    multi = 0
    res = ""

    for c in s:
        if '0' <= c <= '9':
            multi = multi * 10 + int(c)
        elif 'a' <= c <= 'z':
            res += c
        elif c == '[':
            num_stack.append(multi)
            str_stack.append(res)
            res = ""
            multi = 0
        else:
            current_multi = num_stack.pop()
            current_char = str_stack.pop()
            res = current_char + current_multi * res

    return res
```

单调栈

32. 最长有效括号(<https://leetcode-cn.com/problems/longest-valid-parentheses/>)

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。

判断入栈条件

- 如果栈空，入栈
- 当前字符为 '('，入栈
- 当前栈为 ')'

上述三种情况无法进行匹配出栈，所以直接消除

出栈：

通过出栈时 ')' 的下标与队列中下标进行对比。

其实有两种情况

- 当栈为空的时候，表示前面的括号都是合法的，所以合法长度为 $i+1$
- 当栈不为空的时候，表示前面存在不合法括号，所以直接用 $i - index_{not\ legal}$

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:

        stack = []
        max_len = 0
        length = 0

        if len(s) == 0:
            return 0
```

```

for i in range(len(s)):
    if not stack or s[i] == '(' or s[stack[-1]] == ')':
        stack.append(i)
    else:
        stack.pop()
        length = i - (stack[-1] if stack else -1)

    max_len = max(max_len, length)

return max_len

```

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）。

来源: 力扣 (LeetCode)

链接: <https://leetcode-cn.com/problems/trapping-rain-water>

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

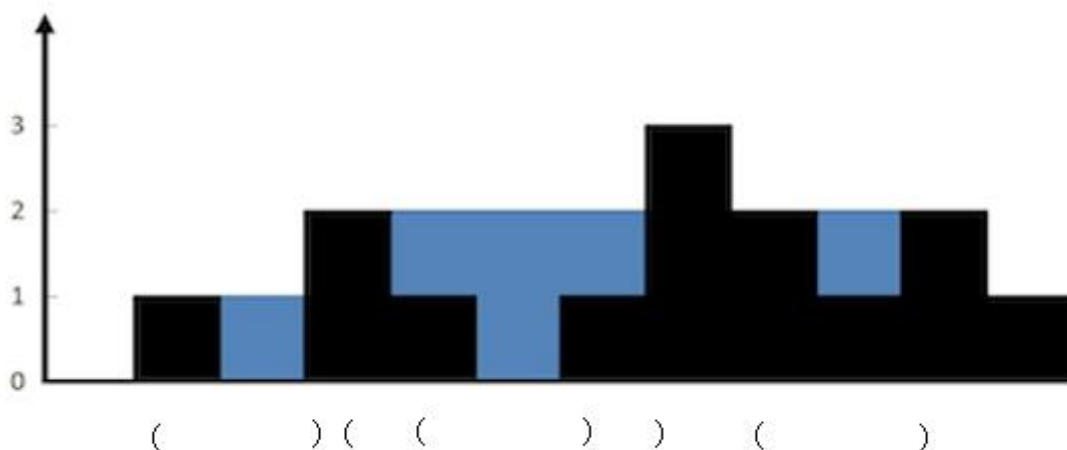
思路: 单调栈

根据下面的图，我们知道接雨水的时候，其实是由两个边界决定的。因此我们构建一个单调递减的栈 `stack` 用于保存当前数组元素的边界。

遍历数组:

- 当栈非空且当前元素大于栈顶元素，弹出栈顶元素 `left`
- 计算当前元素到栈顶元素的距离 $distance = current - stack[-1] - 1$
- 找到界定高度 $max_height = \min(height[stack[-1]], height[current]) - height[left]$
- 计算结果 $res = max_height * distance$

其实上述过程简单来讲，就是每次弹出栈时，按层计算结果。



```

class solution:

```

```

"""
思路：最小栈
"""

def trap(self, height) -> int:

    n = len(height)
    stack = []
    current = 0
    res = 0

    if n == 0:
        return 0

    while current < n:
        # 如果栈为空或者当前位置高度小于栈顶元素，入栈
        while stack and height[current] >= height[stack[-1]]:
            # 弹出栈顶元素
            left = stack.pop()
            if not stack:
                break
            distance = current - stack[-1] - 1
            max_height = min(height[stack[-1]], height[current]) -
height[left]
            res += max_height * distance
            stack.append(current)
            current += 1

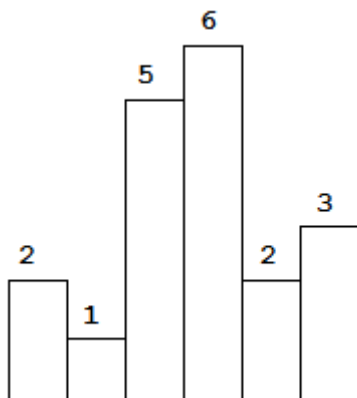
    return res

```

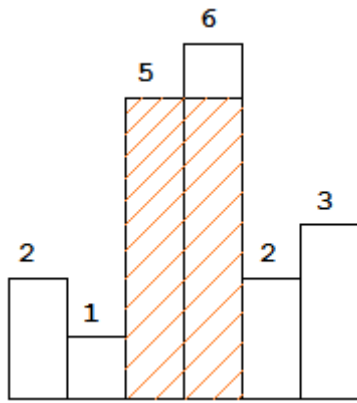
84. 柱状图中最大的矩形

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。

示例:

输入: [2,1,5,6,2,3]
输出: 10

思路: 单调栈, 和接雨水思路差不多。构建一个单调递增栈。

- 如果栈为空或者当前元素小于栈顶元素, 入栈
- 如果栈不为空, 并且当前元素大于栈顶元素, 出栈, $cur_height = nums[stack.pop()]$
- 当前宽度 $cur_width = i - stack[-1] - 1$
- 更新面积

```
class Solution:
    """
    思路一: 单调栈, 当前元素大于栈顶元素, 出栈, 出栈时计算面积
    """
    def largestRectangleArea(self, heights):

        stack = []
        max_area = 0
        heights = [0] + heights + [0]
        n = len(heights)
        for i in range(n):

            if not stack or heights[i] >= heights[stack[-1]]:
                stack.append(i)
            else:
                # 采用单调栈保存最后的结果
                while stack and heights[stack[-1]] > heights[i]:
                    cur_height = heights[stack.pop()]
                    # 关键点, 前面加上一个0, 使得首位形式一样
                    cur_width = i - stack[-1] - 1
                    max_area = max(max_area, cur_height * cur_width)
                stack.append(i)

        return max_area
```

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。

示例 1:

| | | | | |
|---|---|---|---|---|
| 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
解释: 最大矩形如上图所示。

思路: 和前面两题一样

```
class Solution:

    def largestRectangleArea(self, heights) -> int:
        stack = []
        max_area = 0
        heights = [0] + heights + [0]
        n = len(heights)
        for i in range(n):

            if not stack or heights[i] >= heights[stack[-1]]:
                stack.append(i)
            else:
                # 采用单调栈保存最后的结果
                while stack and heights[stack[-1]] > heights[i]:
                    cur_height = heights[stack.pop()]
                    # 关键点, 前面加上一个0, 使得首位形式一样
                    cur_width = i - stack[-1] - 1
                    max_area = max(max_area, cur_height * cur_width)
                stack.append(i)

        return max_area

    def maximalRectangle(self, matrix) -> 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

```

739. 每日温度

难度中等669收藏分享切换为英文接收动态反馈

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。

例如，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]`。

提示： `气温` 列表长度的范围是 `[1, 30000]`。每个气温的值的均为华氏度，都是在 `[30, 100]` 范围内的整数。

思路：保存一个递减单调栈，当当前栈元素大于栈顶元素，弹出栈。

```

class Solution:
    """
    思路一：暴力，超时
    思路二：借助栈
    """
    def dailyTemperatures(self, T):

        n = len(T)
        stack = []
        ans = []
        ans_dict = {}
        for i in range(len(T)):
            if not stack:
                stack.append(i)
            else:
                while stack:
                    j = stack[-1]
                    if T[i] > T[j]:
                        stack.pop()
                        ans_dict[j] = i - j
                    else:
                        break
                stack.append(i)

        while stack:
            i = stack.pop()
            ans_dict[i] = 0

        for i in range(n):
            ans.append(ans_dict[i])

```

```
return ans
```

迷宫求解

采用dfs或者栈，得到迷宫最优解。

[200. 岛屿数量](#)

给你一个由 '1'（陆地）和 '0'（水）组成的二维网格，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。

示例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
```

```
class Solution:
    def numIslands(self, grid):

        m = len(grid)
        n = len(grid[0])

        visited = [[0] * n for _ in range(m)]

        from collections import deque
        q = deque()
        count = 0
        for i in range(m):
            for j in range(n):
                flag = False
                if not q and grid[i][j] == '1' and visited[i][j] == 0:
                    q.append((i, j))
                    visited[i][j] = 1

                while q:
                    pos = q[0]
                    x, y = pos[0], pos[1]

                    # 向左移动
                    if x - 1 >= 0:
                        if grid[x-1][y] == '1' and visited[x-1][y] == 0:
                            q.append((x-1, y))
                            visited[x-1][y] = 1

                    # 向上移动
                    if y - 1 >= 0:
```



```

        if grid[x][y-1] == '1' and visited[x][y-1] == 0:
            q.append((x, y-1))
            visited[x][y-1] = 1

        # 向下移动
        if x + 1 < m:
            if grid[x+1][y] == '1' and visited[x+1][y] == 0:
                q.append((x+1, y))
                visited[x+1][y] = 1

        # 向右移动
        if y + 1 < n:
            if grid[x][y+1] == '1' and visited[x][y+1] == 0:
                q.append((x, y+1))
                visited[x][y+1] = 1

        q.popleft()
        flag = True

    if flag:
        count += 1

    return count

```

树

树的遍历

1、先序遍历

非递归版先序遍历

```

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        ret = []
        stack = []
        while root or stack:
            while root:
                ret.append(root.val)
                stack.append(root)
                root = root.left
            if stack:
                t = stack.pop()
                root = t.right
        return ret

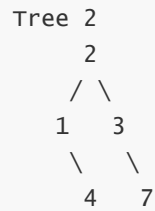
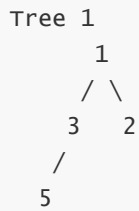
```

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

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

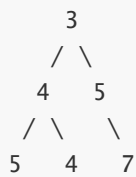
示例 1:

输入:



输出:

合并后的树:



```
class Solution:
```

```
    """
```

```
    二叉树构建:
```

```
    先序遍历，创建二叉树
```

```
    """
```

```
    def create_tree(self, t1, t2):
```

```
        if not t1 and not t2:
```

```
            return None
```

```
        root = TreeNode(0)
```

```
        if t1 and t2:
```

```
            root.val = t1.val + t2.val
```

```
            root.left = self.create_tree(t1.left, t2.left)
```

```
            root.right = self.create_tree(t1.right, t2.right)
```

```
        elif t1:
```

```
            root.val = t1.val
```

```
            root.left = self.create_tree(t1.left, None)
```

```
            root.right = self.create_tree(t1.right, None)
```

```
        elif t2:
```

```
            root.val = t2.val
```

```
            root.left = self.create_tree(None, t2.left)
```

```
            root.right = self.create_tree(None, t2.right)
```

```
        return root
```

```
    def mergeTrees(self, t1, t2):
```

```
        return self.create_tree(t1, t2)
```

2、中序遍历

```
def inorderTraversal(root_node):
```

```
    if not root_node:
```

```

        return
    node_stack = []
    node = root_node
    while node_stack or node:
        # 从跟节点开始，把它的左子树找出来
        while node:
            node_stack.append(node)
            node = node.left
        # 上面 while 的结束就是 node 为空的时候，也就是前一个节点没有左子树的了
        node = node_stack.pop()
        print node.val
        # 这个时候就开始查看右子树了
        node = node.right

```

3、后续遍历

```

def postorderTraversal(root_node):
    if not root_node:
        return
    stack1 = [root_node]
    stack2 = []
    while stack1:
        # 在这个 while 循环里面找到后序遍历的逆序，存在在 stack2 中
        node = stack1.pop()
        if node.left:
            stack1.append(node.left)
        if node.right:
            stack1.append(node.right)
        stack2.append(node)
    while stack2:
        print stack2.pop().val

```

4、层序遍历

[102. 二叉树的层序遍历](#)

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

示例： 二叉树： [3,9,20,null,null,15,7]，

```

    3
   /\
  9 20
 /\  /\
15 7

```

```

from queue import Queue

class Solution:
    def levelOrder(self, root):
        res = []
        if root is None:
            return res

        q = Queue()
        q.put(root)

```

```

while q.qsize() != 0:
    level = []
    size = q.qsize()
    while size > 0:
        root = q.get()
        level.append(root.val)

        if root.left is not None:
            q.put(root.left)

        if root.right is not None:
            q.put(root.right)
        size -= 1
    res.append(level)

return res

```

5、二叉树的深度

```

class Solution:
    def maxDepth(self, root: TreeNode) -> int:

        if root is None:
            return 0
        else:
            left = self.maxDepth(root.left) + 1
            right = self.maxDepth(root.right) + 1
            return max(left, right)

```

6、二叉树的宽度

[543. 二叉树的直径](#)

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

示例：给定二叉树



返回 **3**，它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x

```

```

#         self.left = None
#         self.right = None

class Solution:

    def __init__(self):
        self.max_depth = 0

    def sumTreeDepth(self, root):
        if not root:
            return 0

        left = self.sumTreeDepth(root.left)
        right = self.sumTreeDepth(root.right)
        current = max(left, right)

        if left + right > self.max_depth:
            self.max_depth = left + right

        return current + 1

    def diameterOfBinaryTree(self, root):

        if not root:
            return 0

        self.sumTreeDepth(root)
        return self.max_depth

```

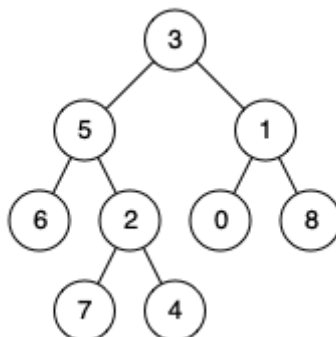
二叉树公共先祖

[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**）。”

示例 1:



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 。

```

class Solution:

    """
    求最近公共祖先，将右节点指向父节点，就能转换成链表求父节点的问题
    """

    def length(self, node):
        count = 0
        while node:
            count += 1
            node = node.right

        return count

    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':

        def inorder(node, parent):

            if node:
                inorder(node.left, node)
                inorder(node.right, node)
                node.right = parent

        inorder(root, None)

        l1 = self.length(p)
        l2 = self.length(q)

        if l1 > l2:
            i = 0
            while i < l1-l2:
                p = p.right
                i += 1
        else:
            i = 0
            while i < l2 - l1:
                q = q.right
                i += 1

        while p and q:
            if p == q:
                return p
            p = p.right
            q = q.right

```

二叉树序列化

[297. 二叉树的序列化与反序列化](#)

```

from collections import deque

class Codec:

    def serialize(self, root):

```

```

        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        if not root:
            return []

        res = []
        q = deque()
        q.append(root)
        while q:
            node = q.popleft()

            if node is None:
                res.append('null')
            else:
                res.append(str(node.val))
                q.append(node.left)
                q.append(node.right)

        return res

def deserialize(self, data):
    """Decodes your encoded data to tree.

    :type data: str
    :rtype: TreeNode
    """
    if not data or len(data) == 0:
        return None
    self.root = TreeNode(data[0])
    queue = deque([self.root])
    leng = len(data)
    nums = 1
    while nums < leng:
        node = queue.popleft()
        if node:
            node.left = TreeNode(data[nums]) if data[nums] else None
            queue.append(node.left)
            if nums + 1 < leng:
                node.right = TreeNode(data[nums + 1]) if data[nums + 1] else
None
                queue.append(node.right)
            nums += 1
        nums += 1
    return self.root

```

字典树

[208. 实现 Trie \(前缀树\)](#)

实现一个 Trie (前缀树), 包含 `insert`, `search`, 和 `startswith` 这三个操作。

示例:

```
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
```

```
class TrieNode:

    def __init__(self, val=None, children=[], end=False):
        self.val = val
        self.children = children
        self.end = end

class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        # 创建一个根节点
        self.node = {}

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        node = self.node
        for w in word:
            if w not in node:
                node[w] = {}
            node = node[w]

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        node = self.node

        for w in word:
            if w not in node:
                return False
            node = node[w]

        return True

    def startswith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given
        prefix.
        """
        node = self.node
```



```

for p in prefix:
    if p not in node:
        return False
    node = node[p]

return True

```

搜索二叉树

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

解析：二叉搜索数的定义是，所有对于所有节点都有

- 根节点大于左节点的
- 根节点小于右节点
- 先序遍历二叉搜索树，得到有序单调递增序列

定义两个函数

- $G(n)$: 长度为 n 的序列能构成不同的二叉搜索树的个数
- $F(i, n)$: 以 i 为根、序列长度为 n 的不同二叉搜索树的个数

以不同的 i 作为节点，可得到

$$G(n) = \sum_{i=1}^n F(i, n)$$

边界条件： $G(0)=G(1)=1$

$F(i, n)$ 的定点固定，所以能够组成的二叉搜索树个数是由两个子节点决定的。即

$[1, i]$ 和 $[i+1, n]$ 决定。而 $[i+1, n]$ 等价于 $[1, n-i-1]$

所以有 $F(i, n) = G(i)G(n-i)$

所以最后的递推公式可以转换成

$$G(n) = \sum_{i=1}^n G(i)G(n-i)$$

该公式可以归纳为卡特兰数

$$C(n+1) = \frac{2(2n+1)}{n+2} C(n)$$

```

def numTrees(self, n):

    G = [0] * (n+1)
    G[0] = G[1] = 1

    for i in range(2, n+1):
        for j in range(1, i+1):
            G[i] += G[j-1] * G[i-j]

    return G[n]

```

类似题目：

[538. 把二叉搜索树转换为累加树](#)

难度中等487收藏分享切换为英文接收动态反馈

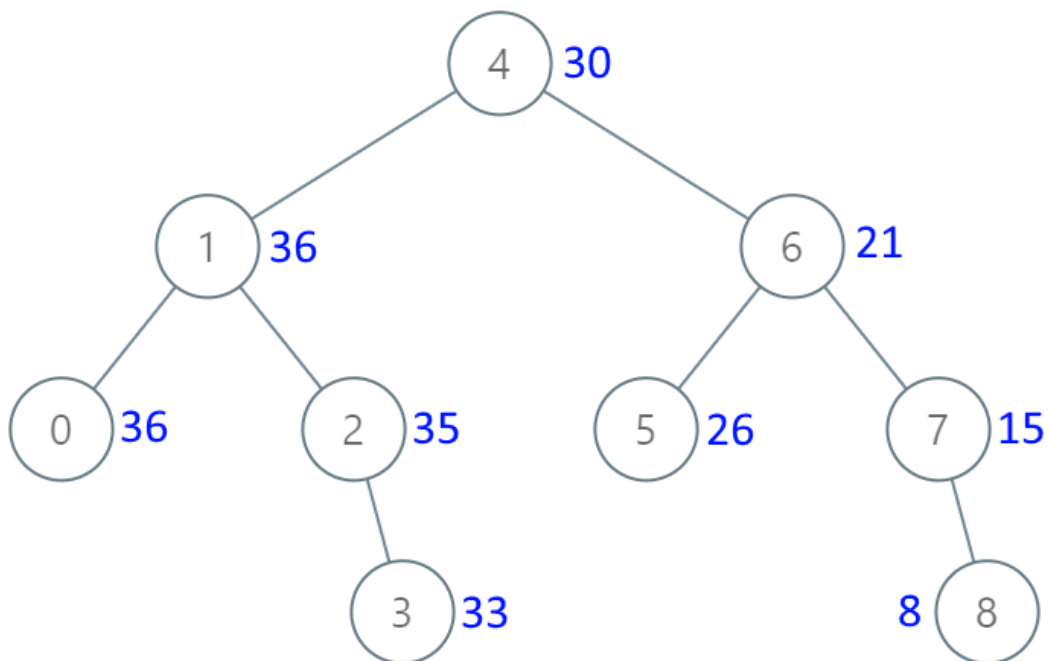
给出二叉 **搜索** 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下，二叉搜索树满足下列约束条件：

- 节点的左子树仅包含键 **小于** 节点键的节点。
- 节点的右子树仅包含键 **大于** 节点键的节点。
- 左右子树也必须是二叉搜索树。

注意：本题和 1038: <https://leetcode-cn.com/problems/binary-search-tree-to-greater-sum-tree/> 相同

示例 1：



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]

位运算

[136. 只出现一次的数字](#)

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

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

输入: [2,2,1]
输出: 1

思路: 位运算, 当两个数相同时, 异或运算结果为0。

```
class Solution:
    def singleNumber(self, nums):

        s = 0
        for i in nums:
            s ^= i

        return s
```

[338. 比特位计数](#)

给定一个非负整数 **num**。对于 $0 \leq i \leq \text{num}$ 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。

示例 1:

输入: 2
输出: [0,1,1]

思路: 转换成二进制后, 是形如这样的数字: aa...aa10...00, 从右向左数有任意多个0, 直到遇见第一个1, 字母a用来占位, 代表1左边的任意数字。

x-1转换成二进制后, 是形如这样的数字: aa...aa01...11, 从右向左数, 原来的任意多个0都变成1, 原来的第一个1, 变成0, 字母a部分不变。

对x 和 x-1 进行 按位与 计算, 会得到: aa...aa00...00, 从右向左数, 原来的第一个1变成了0, 字母a部分不变。

所以 $x \& (x-1)$ 相当于消除了 x 从右向左数遇到的第一个1。

```
class Solution:
    """位运算"""

    def countBits(self, num: int):
        int count_bit(x):
            count = 0
            while x:
                count += 1
                x &= x-1

        res = [count_bit(x) for x in range(0, num+1)]

        return res
```

进阶版:

```
class Solution:
    """位运算"""

    def countBits(self, num: int):
        res = [0] * (num + 1)

        for i in range(1, num + 1):
            res[i] = res[i & i - 1] + 1

        return res
```

461. 汉明距离

难度简单379收藏分享切换为英文接收动态反馈

两个整数之间的[汉明距离](#)指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y ，计算它们之间的汉明距离。

注意： $0 \leq x, y < 231$ 。

示例：

输入： $x = 1, y = 4$

输出：2

解释：

```
1   (0 0 0 1)
4   (0 1 0 0)
    ↑   ↑
```

上面的箭头指出了对应二进制位不同的位置。

思路：先异或运算，再为运算。

```
class Solution:
    """
    汉明距离，数字中有多少个1的变形
    """

    def hammingDistance(self, x, y):
        nums = x ^ y

        count = 0
        while nums != 0:
            if nums & 1:
                count += 1
            nums >>= 1

        return count
```

查找算法

二分查找

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**。

示例 1:

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

思路一：将两个序列进行合并，类似链表的操作方式，然后找到中位数。

```
class Solution:
    def findMedianSortedArrays(self, nums1, nums2):
        nums = []
        len1 = len(nums1)
        len2 = len(nums2)
        i = 0
        j = 0
        while i < len1 and j < len2:
            if nums1[i] < nums2[j]:
                nums.append(nums1[i])
                i += 1
            else:
                nums.append(nums2[j])
                j += 1

        if i < len1:
            nums += nums1[i: len1]
        elif j < len2:
            nums += nums2[j: len2]

        nums_size = len(nums)
        middle_index = int(nums_size/2)
        if nums_size == 0:
            return []
        elif nums_size % 2 == 0:
            return (nums[middle_index - 1] + nums[middle_index]) / 2
        else:
            return nums[middle_index]
```

思路二：二分查找。暂时没看懂

二分查找解法：

[33. 搜索旋转排序数组](#)

整数数组 `nums` 按升序排列，数组中的值 **互不相同**。

在传递给函数之前，`nums` 在预先未知的某个下标 k ($0 \leq k < \text{nums.length}$) 上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标从 0 开始计数）。例如，`[0,1,2,4,5,6,7]` 在下标 3 处经旋转后可能变为 `[4,5,6,7,0,1,2]`。

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

思路: 二分查找, 比较简单, 直接贴代码。

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        if not nums:
            return -1
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] == target:
                return mid
            if nums[0] <= nums[mid]:
                if nums[0] <= target < nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            else:
                if nums[mid] < target <= nums[len(nums) - 1]:
                    l = mid + 1
                else:
                    r = mid - 1
        return -1
```

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**

思路: 归并排序

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    """
    方法一: 冒泡排序 时间复杂度O(n^2)
    方法二: 归并排序
    """

    def sortList(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head
        slow = head
        fast = head
        # 用快慢指针分成两部分
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        # 找到左右部分, 把左部分最后置空
        mid = slow.next
        slow.next = None
        # 递归下去
```

```

    left = self.sortList(head)
    right = self.sortList(mid)
    # 合并
    return self.merge(left, right)

def merge(self, left, right):
    dummy = ListNode(0)
    p = dummy
    l = left
    r = right

    while l and r:
        if l.val < r.val:
            p.next = l
            l = l.next
            p = p.next
        else:
            p.next = r
            r = r.next
            p = p.next
    if l:
        p.next = l
    if r:
        p.next = r
    return dummy.next

```

哈希表法

分块查找

图算法

拓扑排序

[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`。

示例 1：

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。

思路：

本质是有向图是否存在环，拓扑排序问题

- 1、首先找到度入度为0的点
- 2、然后以改点为起点，找到指向的点，度-1
- 3、然后找到指向的度为0的点，重复1/2操作。

```
import collections

class Solution:

    def canFinish(self, numCourses: int, prerequisites):
        # 采用邻接链表
        edges = collections.defaultdict(list)
        indeg = [0] * numCourses

        # 构建边与入度
        for info in prerequisites:
            edges[info[1]].append(info[0])
            indeg[info[0]] += 1

        zero_indeg = [i for i in range(numCourses) if indeg[i] == 0]

        q = collections.deque(zero_indeg)

        count = 0
        while q:
            node = q.popleft()
            count += 1

            for i in edges[node]:
                indeg[i] -= 1
                if indeg[i] == 0:
                    q.append(i)

        return count == numCourses
```

复杂数据结构

并查集

[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 $A_i / B_i = \text{values}[i]$ 。每个 A_i 或 B_i 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 $C_j / D_j = ?$ 的结果作为答案。

示例 1：

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]`

```
class UnionFind:
    # 定义一个并查集类
    # 特点: 一边查询, 一边修改节点指向

    def __init__(self):
        self.parent = {}
        self.weight = {}

    def find(self, x):
        # 路径压缩
        # 找到根节点

        root = x
        multi = 1

        # if x in self.parent:
        # 判断查询语句是否有效
        while self.parent[root] != root:
            # 计算路径上的权值
            multi *= self.weight[root]
            root = self.parent[root]
        while x != root:
            last_parent = self.parent[x]
            cur_weight = self.weight[x]
            self.weight[x] = multi
            multi /= cur_weight
            self.parent[x] = root
            x = last_parent

        return root

    def merge(self, x, y, val):
        # 合并并查集
        parent_x = self.find(x)
        parent_y = self.find(y)

        if parent_x == parent_y:
            return

        if parent_x != parent_y:
            self.parent[parent_x] = parent_y
            # 合并之后更新权值
            self.weight[parent_x] = self.weight[y] * val / self.weight[x]

    def is_connected(self, x, y):
        # 判断两点是否相连
```

```

        return x in self.parent and y in self.parent and self.find(x) ==
self.find(y)

    def add(self, x):

        if x not in self.parent:
            self.parent[x] = x
            self.weight[x] = 1.0

class Solution:
    """
    思路：带权重并查集
    """

    def calcEquation(self, equations, values, queries):
        uf = UnionFind()
        # 构建并查集
        for (a, b), val in zip(equations, values):
            uf.add(a)
            uf.add(b)
            uf.merge(a, b, val)

        res = []

        for (a, b) in queries:
            if uf.is_connected(a, b):
                res.append(uf.weight[a]/uf.weight[b])
            else:
                res.append(-1.0)

        return res

```

LRU

[146. LRU 缓存机制](#)

难度中等1219收藏分享切换为英文接收动态反馈

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。

进阶：你是否可以在 `O(1)` 时间复杂度内完成这两种操作？

示例：

```

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
```

思路：哈希表+双向链表

```
import collections

class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache(collections.OrderedDict):
    """
    思路一：采用OrderedDict，面试一般需要自己实现
    思路二：哈希表+双向链表
    """

    def __init__(self, capacity: int):
        self.cache = {}
        self.head = DLinkedNode()
        self.tail = DLinkedNode()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.capacity = capacity
        self.size = 0

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1

        node = self.cache[key]
        self.moveToHead(node)
        return node.value

    def put(self, key: int, value: int) -> None:

        if key not in self.cache:
            node = DLinkedNode(key, value)
            self.cache[key] = node
```

```

        if self.size >= self.capacity:
            removed = self.removeTail()
            self.cache.pop(removed.key)
        self.addToHead(node)
    else:
        # 如果存在哈希冲突，修改结果并将其移动到链表头部
        node = self.cache[key]
        node.value = value
        self.moveToHead(node)
        print("哈希冲突")

def addToHead(self, node):
    node.next = self.head.next
    node.prev = self.head
    self.head.next.prev = node
    self.head.next = node
    self.size += 1

def removeNode(self, node):
    node.prev.next = node.next
    node.next.prev = node.prev
    self.size -= 1

def moveToHead(self, node):
    self.removeNode(node)
    self.addToHead(node)

def removeTail(self):
    node = self.tail.prev
    self.removeNode(node)
    return node

```

Your LRUCache object will be instantiated and called as such:

LFU

排序算法

在[计算机科学与数学](#)中，一个**排序算法**（英语：Sorting algorithm）是一种能将一串资料依照特定排序方式进行排列的一种[算法](#)。最常用到的排序方式是数值顺序以及[字典顺序](#)。有效的排序算法在一些算法（例如[搜索算法](#)与[合并算法](#)）中是重要的，如此这些算法才能得到正确解答。排序算法也用在处理文字资料以及产生人类可读的输出结果。基本上，排序算法的输出必须遵守下列两个原则：

1. 输出结果为递增序列（递增是针对所需的排序顺序而言）
2. 输出结果是原输入的一种[排列](#)、或是重组

| 排序算法 | 时间复杂度 | 稳定性 |
|------|---------------|-----|
| 冒泡排序 | $O(n^2)$ | 稳定 |
| 选择排序 | $O(n^2)$ | 不稳定 |
| 插入排序 | $O(n^2)$ | 稳定 |
| 快速排序 | $O(n \log n)$ | 不稳定 |
| 归并排序 | $O(n \log n)$ | 稳定 |
| 堆排序 | $O(n \log n)$ | 不稳定 |
| 计数排序 | $O(n)$ | 稳定 |
| 基数排序 | $O(n)$ | 稳定 |

归并排序

```

void Merge(int *a,int s,int m,int n)
{
    int temp[20];
    int i=s,q=s;
    int j=m+1;
    while(i<=m&& j<=n)
    {
        if(a[i]<a[j])
            temp[q++]=a[i++];
        else
            temp[q++]=a[j++];
    }

    while(i<=m)
        temp[q++]=a[i++];
    while(j<=n)
        temp[q++]=a[j++];
    for(int k=s;k<=n;k++)
        a[k]=temp[k];
}

void MSort(int *a,int s,int t)
{
    int m;
    if(s==t)
        return;
    else
    {
        m=(s+t)/2;
        MSort(a,s,m);
        MSort(a,m+1,t);
        Merge(a,s,m,t);
    }
}

```

快速排序

```
int Partition(SqList &L,int low,int high)//最后返回的位置为枢轴的位置
{
    RedType t;
    L.r[0]=L.r[low]; //设置枢轴
    while(low<high)
    {
        while(L.r[high].key>=L.r[0].key&&low<high)
            high--;
        L.r[low]=L.r[high];
        while(L.r[low].key<L.r[0].key&&low<high)
            low++;
        L.r[high]=L.r[low];
    }
    L.r[low]=L.r[0];
    return low;
}

void Qsort(SqList &L,int low,int high)
{
    int i;
    if(low<high)
    {
        i=Partition(L,low,high);
        Qsort(L,low,i-1);
        Qsort(L,i+1,high);
        count++;
        printf("第%d趟排序:",count);
        Print(L);
    }
}
```

[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。

示例 1:

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

选择排序

冒泡排序

堆排序

大根堆：父节点的值大于子节点的值。

小根堆：父节点的值小于子节点值。

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。

思路一：利用堆排序

构建一个小根堆，将元素一次插入其中，然后将链表一次放入其中。

然后一次将链表弹出，最后得到的就是有序的链表。

思路二：将链表结果放到数组里面，然后利用快排。

```
class Solution(object):
    def mergeKLists(self, lists):
        import heapq
        head = point = ListNode(0)
        heap = []
        for l in lists:
            while l:
                heapq.heappush(heap, l.val)
                l = l.next
        while heap:
            val = heapq.heappop(heap)
            point.next = ListNode(val)
            point = point.next
        point.next = None
        return head.next
```

桶排序

基数排序

希尔排序