

Coding

1. Add Binary 二进制求和

<https://leetcode-cn.com/problems/add-binary/>

给你两个二进制字符串 a 和 b，以二进制字符串的形式返回它们的和。

示例 1:

输入:a = "11", b = "1"

输出:"100"

示例 2:

输入:a = "1010", b = "1011"

输出:"10101"

提示:

- $1 \leq a.length, b.length \leq 10^4$
- a 和 b 仅由字符 '0' 或 '1' 组成
- 字符串如果不是 "0"，就不含前导零

02/16/22, 02/22, 03/01, 03/16

可以用One Pointer跟踪目前位数上的和:

```
def addBinary(a, b):
    digit = 0
    result = []
    for i in range(1, max(len(a), len(b))+1):
        if i <= len(a):
            digit += int(a[-i])
        if i <= len(b):
            digit += int(b[-i])
        result.append(str(digit%2))
        digit = digit // 2
    if digit == 1:
        result.append(str(digit))
    return ''.join(result[::-1])
```

time complexity: $O(n)$
space complexity: $O(n)$

2. Add Strings 字符串相加

<https://leetcode-cn.com/problems/add-strings/>

给定两个字符串形式的非负整数 `num1` 和 `num2`，计算它们的和并同样以字符串形式返回。

你不能使用任何内建的用于处理大整数的库（比如 `BigInteger`），也不能直接将输入的字符串转换为整数形式。

示例 1:

输入: `num1 = "11"`, `num2 = "123"`
输出: `"134"`

示例 2:

输入: `num1 = "456"`, `num2 = "77"`
输出: `"533"`

示例 3:

输入: `num1 = "0"`, `num2 = "0"`
输出: `"0"`

提示:

- $1 \leq \text{num1.length}, \text{num2.length} \leq 10^4$
- `num1` 和 `num2` 都只包含数字 0-9
- `num1` 和 `num2` 都不包含任何前导零

02/16/22, 02/22, 03/01, 03/16

```
def addStrings(self, num1: str, num2: str) -> str:
    digit = 0
    result = []
    for i in range(1, max(len(num1), len(num2))+1):
        if i <= len(num1):
            digit += int(num1[-i])
        if i <= len(num2):
            digit += int(num2[-i])
        result.append(str(digit%10))
```

```

        digit = digit // 10
    if digit == 1:
        result.append(str(digit))
    return ".join(result[::-1])

```

time complexity: $O(n)$
 space complexity: $O(n)$

3. Intersection of Two Arrays 两个数组的交集

<https://leetcode-cn.com/problems/intersection-of-two-arrays/>

给定两个数组 `nums1` 和 `nums2`，返回 它们的 交集 。输出结果中的每个元素一定是 唯一的。
 我们可以 不考虑输出结果的顺序 。

示例 1：

输入：`nums1 = [1,2,2,1]`, `nums2 = [2,2]`
 输出：`[2]`

示例 2：

输入：`nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`
 输出：`[9,4]`
 解释：`[4,9]` 也是可通过的

提示：

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

02/16/22, 02/21, 03/01, 03/16

先用set去掉重复的元素：

```

def intersection(nums1, nums2):
    nums1 = set(nums1)
    nums2 = set(nums2)
    if len(nums1) <= len(nums2):
        return [num for num in nums1 if num in nums2]
    else:
        return [num for num in nums2 if num in nums1]

```

4. Sliding Window Maximum 滑动窗口最大值

<https://leetcode-cn.com/problems/sliding-window-maximum/>

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

示例 2:

输入: `nums = [1]`, `k = 1`

输出: `[1]`

提示:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $1 \leq k \leq \text{nums.length}$

02/18/22, 02/21, 03/01, 03/16, 03/28

最直接的方法是，建立一个sliding window list，然后取最大值，每移动一次就更新这个list，然后接着求最大值。

```
def maxSlidingWindow(nums, k):  
    slide = nums[:k]  
    res = [max(slide)]  
    i = k  
    while i < len(nums):  
        slide.pop(0)
```

```

        slide.append(nums[i])
        res.append(max(slide))
        i += 1
    return res

```

time complexity: $O(kn)$

space complexity: $O(n)$

然后, 为了使程序运行在linear time, 我们可以用一个deque表示滑动窗口的索引, 同时保持deque的左端为最大元素索引。

```

from collections import deque

def maxSlidingWindow(nums, k):
    deq = deque()
    res = []
    for i in range(len(nums)):
        while deq and nums[i] > nums[deq[-1]]:
            deq.pop()
        deq.append(i)
        if deq[0] == i-k:
            deq.popleft()
        if i >= k-1:
            res.append(nums[deq[0]])
    return res

```

time complexity: $O(n)$

space complexity: $O(n)$

贪心算法 (5 - 9):

5. Assign Cookies 分发饼干

<https://leetcode-cn.com/problems/assign-cookies/>

假设你是一位很棒的家长, 想要给你的孩子们一些小饼干。但是, 每个孩子最多只能给一块饼干。

对每个孩子 i , 都有一个胃口值 $g[i]$, 这是能让孩子们满足胃口的饼干的最小尺寸; 并且每块饼干 j , 都有一个尺寸 $s[j]$ 。如果 $s[j] \geq g[i]$, 我们可以将这个饼干 j 分配给孩子 i , 这个孩子会得到满足。你的目标是满足尽可能多的孩子, 并输出这个最大数值。

示例 1:

输入: $g = [1,2,3]$, $s = [1,1]$

输出: 1

解释:

你有三个孩子和两块小饼干, 3 个孩子的胃口值分别是: 1,2,3。

虽然你有两块小饼干, 由于他们的尺寸都是 1, 你只能让胃口值是 1 的孩子满足。

所以你应该输出 1。

示例 2:

输入: $g = [1,2]$, $s = [1,2,3]$

输出: 2

解释:

你有两个孩子和三块小饼干, 2 个孩子的胃口值分别是 1,2。

你拥有的饼干数量和尺寸都足以让所有孩子满足。

所以你应该输出 2。

提示:

- $1 \leq g.length \leq 3 \times 10^4$
- $0 \leq s.length \leq 3 \times 10^4$
- $1 \leq g[i], s[j] \leq 2^{31} - 1$

02/23/22, 03/01, 03/16

首先排序, 然后使用贪心算法, 贪心策略是, 给剩余孩子里最小饥饿度的孩子分配最小的能饱腹的饼干:

```
def findContentChildren(g, s):
    g.sort()
    s.sort()
    i = j = 0
    res = 0
    while i < len(g) and j < len(s):
        if g[i] <= s[j]:
            res += 1
            i += 1
            j += 1
        else:
            j += 1
    return res
```

time complexity: $O(n \log n)$

space complexity: $O(1)$

6. Candy 分发糖果

<https://leetcode-cn.com/problems/candy/>

n 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子中，评分更高的那个会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的 最少糖果数目 。

示例 1：

输入：`ratings = [1,0,2]`

输出：5

解释：你可以分别给第一个、第二个、第三个孩子分发 2、1、2 颗糖果。

示例 2：

输入：`ratings = [1,2,2]`

输出：4

解释：你可以分别给第一个、第二个、第三个孩子分发 1、2、1 颗糖果。

第三个孩子只得到 1 颗糖果，这满足题面中的两个条件。

提示：

- $n == ratings.length$
- $1 \leq n \leq 2 * 10^4$
- $0 \leq ratings[i] \leq 2 * 10^4$

02/23/22, 03/01, 03/16

我们可以做two pass, 先从左往右遍历, 再从右往左遍历。这里的贪心策略是, 在每次遍历中, 只考虑并更新相邻一侧的大小关系：

```
def candy(ratings):  
    n = len(ratings)  
    res = [1] * n  
    for i in range(n-1):  
        if ratings[i+1] > ratings[i]:
```

```

        res[i+1] = ratings[i] + 1
    for i in range(n-2, -1, -1):
        if ratings[i] > ratings[i+1]:
            res[i] = max(res[i], res[i+1] + 1)
    return sum(res)

```

time complexity: $O(n)$

space complexity: $O(n)$

7. Non-overlapping Intervals 无重叠区间

<https://leetcode-cn.com/problems/non-overlapping-intervals/>

给定一个区间的集合 `intervals`，其中 `intervals[i] = [starti, endi]`。返回 需要移除区间的最小数量，使剩余区间互不重叠。

注意 只在一点上接触的区间是 不重叠的。例如 `[1, 2]` 和 `[2, 3]` 是不重叠的。

示例 1:

输入: `intervals = [[1,2],[2,3],[3,4],[1,3]]`
 输出: 1
 解释: 移除 `[1,3]` 后，剩下的区间没有重叠。

示例 2:

输入: `intervals = [[1,2], [1,2], [1,2]]`
 输出: 2
 解释: 你需要移除两个 `[1,2]` 来使剩下的区间没有重叠。

示例 3:

输入: `intervals = [[1,2], [2,3]]`
 输出: 0
 解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

提示:

- $1 \leq \text{intervals.length} \leq 10^5$
- $\text{intervals}[i].\text{length} == 2$
- $-5 \times 10^4 \leq \text{start}_i < \text{end}_i \leq 5 \times 10^4$

02/23/22, 03/01, 03/16

首先排序。求最少移除区间个数，等价于尽量多保留不重叠的区间。在选择要保留的区间时，选择的区间的结尾越小，余留给其他区间的空间就越大，就越能保留更多的区间：

```
def eraseOverlapIntervals(intervals):
    intervals.sort(key=lambda x: x[1])
    count = 0
    end = intervals[0][1]
    for interval in intervals[1:]:
        if interval[0] >= end:
            end = interval[1]
        else:
            count += 1
            end = min(end, interval[1])
    return count
```

time complexity: $O(n \log n)$

space complexity: $O(1)$

8. Merge Intervals 合并区间

<https://leetcode-cn.com/problems/merge-intervals/>

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [start, end]`。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]`。

示例 2:

输入: `intervals = [[1,4],[4,5]]`

输出: `[[1,5]]`

解释: 区间 `[1,4]` 和 `[4,5]` 可被视为重叠区间。

示例 3:

输入: `intervals = [[4,7],[1,4]]`

输出: `[[1,7]]`

解释: 区间 `[1,4]` 和 `[4,7]` 可被视为重叠区间。

提示:

- $1 \leq \text{intervals.length} \leq 10^4$
- `intervals[i].length == 2`

- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^4$

02/18/22, 02/21, 03/01, 03/16

首先按照interval第一个元素排序：

```
def merge(intervals):
    intervals.sort(key=lambda x: x[0])
    res = []
    for interval in intervals:
        if not res:
            res.append(interval)
        elif interval[0] > res[-1][1]:
            res.append(interval)
        else:
            res[-1][1] = max(res[-1][1], interval[1])
    return res
```

time complexity: $O(n \log n)$

space complexity: $O(n)$

9. Best Time to Buy and Sell Stock II 买卖股票的最佳时机 II

<https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/>

给你一个整数数组 prices，其中 prices[i] 表示某支股票第 i 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。然而，你可以在 同一天 多次买卖该股票，但要确保你持有的股票不超过一股。

返回 你能获得的 最大 利润。

示例 1：

输入：prices = [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 = $6 - 3 = 3$ 。

最大总利润为 $4 + 3 = 7$ 。

示例 2：

输入：prices = [1,2,3,4,5]

输出：4

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

最大总利润为 4。

示例 3:

输入: prices = [7,6,4,3,1]

输出: 0

解释: 在这种情况下, 交易无法获得正利润, 所以不参与交易可以获得最大利润, 最大利润为 0。

提示:

- $1 \leq \text{prices.length} \leq 3 \times 10^4$
- $0 \leq \text{prices}[i] \leq 10^4$

02/24/22, 03/01, 03/17

two pointers: buy和sell, 遍历prices, 当前price小于buy, 就更新为buy, price大于buy, 就sell, 再更新为buy:

```
maxProfit(prices):
    profit = 0
    buy = prices[0]
    for price in prices[1:]:
        if price < buy:
            buy = price
        elif price > buy:
            sell = price
            profit += sell - buy
            buy = price
    return profit
```

time complexity: $O(n)$

space complexity: $O(1)$

Two Pointers (10 - 15):

10. Partition Labels 划分字母区间

<https://leetcode-cn.com/problems/partition-labels/>

给你一个字符串 s。我们要把这个字符串划分为尽可能多的片段, 同一字母最多出现在一个片段中。例如, 字符串 "ababcc" 能够被分为 ["abab", "cc"], 但类似 ["aba", "bcc"] 或 ["ab", "ab", "cc"] 的划分是非法的。

注意, 划分结果需要满足: 将所有划分结果按顺序连接, 得到的字符串仍然是 s 。

返回一个表示每个字符串片段的长度的列表。

示例 1:

输入: $s = \text{"ababcbacadefegdehijhklij"}$

输出: $[9,7,8]$

解释:

划分结果为 "ababcbaca" 、 "defegde" 、 "hijhklij" 。

每个字母最多出现在一个片段中。

像 $\text{"ababcbacadefegde"}$ 、 "hijhklij" 这样的划分是错误的, 因为划分的片段数较少。

示例 2:

输入: $s = \text{"eccbbbbbdec"}$

输出: $[10]$

提示:

- $1 \leq s.length \leq 500$
- s 仅由小写英文字母组成

02/23/22, 02/24, 02/25, 03/01, 03/17

首先, 创建一个字典, 字典里存储的是每一个字母在句子里最后一次出现的index。然后用 two pointers, 分别代表这个section的首尾index。从左到右遍历, 尾端index不断更新, 直到遇到相同的字母为止。

```
def partitionLabels(s):  
    rightmost = {letter:idx for idx, letter in enumerate(s)}  
    left = right = 0  
    res = []  
    for i, letter in enumerate(s):  
        right = max(right, rightmost[letter])  
        if i == right:  
            res.append(right-left+1)  
            left = i + 1  
    return res
```

time complexity: $O(n)$

space complexity: $O(n)$

11. Two Sum 两数之和

<https://leetcode-cn.com/problems/two-sum/>

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 `target` 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, 并且你不能使用两次相同的元素。

你可以按任意顺序返回答案。

示例 1:

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]` 。

示例 2:

输入: `nums = [3,2,4]`, `target = 6`

输出: `[1,2]`

示例 3:

输入: `nums = [3,3]`, `target = 6`

输出: `[0,1]`

提示:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- 只会存在一个有效答案

02/15/22, 02/21, 02/24, 03/01, 03/18

Array不是sorted的话, 我们可以用一个Hash Table:

```
def two_sum(numbers, target):
    numbers_dict = {}
    for i in range(len(numbers)):
        if numbers[i] in numbers_dict:
            return [i, numbers_dict[numbers[i]]]
        else:
            numbers_dict[target-numbers[i]] = i
```

time complexity: $O(n)$

space complexity: $O(n)$

12. Two Sum II 两数之和 II - 输入有序数组

<https://leetcode-cn.com/problems/two-sum-ii-input-array-is-sorted/>

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 非递减顺序排列，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$ 。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 只对应唯一的答案，而且你 不可以 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

示例 1:

输入: `numbers = [2,7,11,15]`, `target = 9`

输出: `[1,2]`

解释: 2 与 7 之和等于目标数 9。因此 `index1 = 1`, `index2 = 2`。返回 `[1, 2]`。

示例 2:

输入: `numbers = [2,3,4]`, `target = 6`

输出: `[1,3]`

解释: 2 与 4 之和等于目标数 6。因此 `index1 = 1`, `index2 = 3`。返回 `[1, 3]`。

示例 3:

输入: `numbers = [-1,0]`, `target = -1`

输出: `[1,2]`

解释: -1 与 0 之和等于目标数 -1。因此 `index1 = 1`, `index2 = 2`。返回 `[1, 2]`。

提示:

- $2 \leq \text{numbers.length} \leq 3 \times 10^4$
- $-1000 \leq \text{numbers}[i] \leq 1000$
- `numbers` 按 非递减顺序 排列
- $-1000 \leq \text{target} \leq 1000$
- 仅存在一个有效答案

02/15/22, 02/21, 02/24, 03/01, 03/19

Array是sorted的话，可以用Two Pointers，来降低space complexity:

```
def two_sum(numbers, target):  
    left, right = 0, len(numbers)-1  
    while left < right:  
        if numbers[left] + numbers[right] == target:
```

```
        return [left+1, right+1]
    elif numbers[left] + numbers[right] < target:
        left += 1
    else:
        right -= 1
```

time complexity: $O(n)$

space complexity: $O(1)$

13. Valid Palindrome 验证回文串

<https://leetcode-cn.com/problems/valid-palindrome/>

如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样。则可以认为该短语是一个回文串。

字母和数字都属于字母数字字符。

给你一个字符串 s ，如果它是回文串，返回 `true`；否则，返回 `false`。

示例 1:

输入: $s = \text{"A man, a plan, a canal: Panama"}$

输出: `true`

解释: "amanaplanacanalpanama" 是回文串。

示例 2:

输入: $s = \text{"race a car"}$

输出: `false`

解释: "raceacar" 不是回文串。

示例 3:

输入: $s = \text{" "}$

输出: `true`

解释: 在移除非字母数字字符之后, s 是一个空字符串 ""。

由于空字符串正着反着读都一样, 所以是回文串。

提示:

- $1 \leq s.length \leq 2 * 10^5$
- s 仅由可打印的 ASCII 字符组成

02/15/22, 02/22, 02/24, 03/01, 03/22

我们可以用Two Pointers:

```
import re

def palindrome(s):
    re_pattern = r'^a-zA-Z0-9'
    s = re.sub(re_pattern, "", s).lower()
    left, right = 0, len(s)-1
    while left < right:
        if s[left] != s[right]:
            return False
        else:
            left, right = left+1, right-1
    return True
```

time complexity: $O(n)$

space complexity: $O(1)$

为了使代码更简洁, 还可以利用Python列表切片:

```
import re

def palindrome(s):
    re_pattern = r'^a-zA-Z0-9'
    s = re.sub(re_pattern, "", s)
    return s == s[::-1]
```

time complexity: $O(n)$

space complexity: $O(1)$

14. Valid Palindrome II 验证回文字符串 II

<https://leetcode-cn.com/problems/valid-palindrome-ii/>

给你一个字符串 s , 最多 可以从中删除一个字符。

请你判断 s 是否能成为回文字符串: 如果能, 返回 `true` ; 否则, 返回 `false` 。

示例 1:

输入: $s = "aba"$

输出: `true`

示例 2:

输入:s = "abca"

输出:true

解释:你可以删除字符 'c' 。

示例 3:

输入:s = "abc"

输出:false

提示:

- $1 \leq s.length \leq 10^5$
- s 由小写英文字母组成

02/15/22, 02/22, 02/24, 03/01, 03/22

我们可以用Two Pointers:

```
def palindrome(s):
    left, right = 0, len(s)-1
    while left < right:
        if s[left] != s[right]:
            s1 = s[left:right]
            s2 = s[left+1:right+1]
            return (s1==s1[::-1]) or (s2==s2[::-1])
        else:
            left, right = left+1, right-1
    return True
```

time complexity: $O(n)$

space complexity: $O(n)$

15. Merge Sorted Array 合并两个有序数组

<https://leetcode-cn.com/problems/merge-sorted-array/>

给你两个按 非递减顺序 排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。

请你 合并 nums2 到 nums1 中，使合并后的数组同样按 非递减顺序 排列。

注意:最终,合并后数组不应由函数返回,而是存储在数组 nums1 中。为了应对这种情况,nums1 的初始长度为 m + n,其中前 m 个元素表示应合并的元素,后 n 个元素为 0,应忽略。nums2 的长度为 n。

示例 1:

输入:nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

输出:[1,2,2,3,5,6]

解释:需要合并 [1,2,3] 和 [2,5,6]。

合并结果是 [1,2,2,3,5,6], 其中斜体加粗标注的为 nums1 中的元素。

示例 2:

输入:nums1 = [1], m = 1, nums2 = [], n = 0

输出:[1]

解释:需要合并 [1] 和 []。

合并结果是 [1]。

示例 3:

输入:nums1 = [0], m = 0, nums2 = [1], n = 1

输出:[1]

解释:需要合并的数组是 [] 和 [1]。

合并结果是 [1]。

注意,因为 m = 0,所以 nums1 中没有元素。nums1 中仅存的 0 仅仅是为了确保合并结果可以顺利存放到 nums1 中。

提示:

- nums1.length == m + n
- nums2.length == n
- 0 <= m, n <= 200
- 1 <= m + n <= 200
- -10⁹ <= nums1[i], nums2[j] <= 10⁹

02/24/22, 02/25, 03/01, 03/22

用Two Pointers, 分别指向array m和array n, 因为array是排好序的, 所以每次比较最后的一个数, 然后移动指针, 复制数字, 直到复制完为止:

```
def merge(nums1, m, nums2, n):  
    while m > 0 and n > 0:  
        if nums1[m-1] >= nums2[n-1]:  
            nums1[m+n-1] = nums1[m-1]  
            m -= 1  
        else:
```

```

                                nums1[m+n-1] = nums2[n-1]
                                n -= 1
                                if n > 0:
                                    nums1[:n] = nums2[:n]

```

time complexity: $O(m+n)$

space complexity: $O(1)$

滑动窗口 (16 - 20)

16. Minimum Window Substring 最小覆盖子串

<https://leetcode-cn.com/problems/minimum-window-substring/>

给定两个字符串 s 和 t , 长度分别是 m 和 n , 返回 s 中的 最短窗口 子串, 使得该子串包含 t 中的每一个字符(包括重复字符)。如果没有这样的子串, 返回空字符串 ""。

测试用例保证答案唯一。

示例 1:

输入: $s = \text{"ADOBECODEBANC"}, t = \text{"ABC"}$

输出: "BANC"

解释: 最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

示例 2:

输入: $s = \text{"a"}, t = \text{"a"}$

输出: "a"

解释: 整个字符串 s 是最小覆盖子串。

示例 3:

输入: $s = \text{"a"}, t = \text{"aa"}$

输出: ""

解释: t 中两个字符 'a' 均应包含在 s 的子串中, 因此没有符合条件的子字符串, 返回空字符串。

提示:

- $m == s.length$
- $n == t.length$
- $1 \leq m, n \leq 10^5$

- s 和 t 由英文字母组成

02/25/22, 03/01, 03/17

我们可以利用滑动窗口的思想，创建两个字典，dict_s存储s中滑动窗口中各个字符出现的次数，dict_t存储t中各个字符出现的次数。如果dict_s中包含dict_t中的所有字符，并且对应的个数都不小于dict_t各个字符的个数，那么说明当前窗口是可行的，可行窗口中长度最短的滑动窗口就是答案。为了解决这个问题，我们可以用two pointers表示滑动窗口，另外定义一个计数器counter，counter记录的是s中滑动窗口中满足t的所有字符的个数。因为两个指针都严格递增，最多移动n次，所以总的时间复杂度是O(n):

```
from collections import defaultdict

def minWindow(s, t):
    if len(s) < len(t):
        return ""
    dict_s = defaultdict(int)
    dict_t = defaultdict(int)
    for letter in t:
        dict_t[letter] += 1
    left = right = 0
    counter = 0
    res = ""
    while right < len(s):
        dict_s[s[right]] += 1
        if dict_s[s[right]] <= dict_t[s[right]]:
            counter += 1
        while left <= right and dict_s[s[left]] > dict_t[s[left]]:
            dict_s[s[left]] -= 1
            left += 1
        if counter == len(t):
            if not res or right-left+1 < len(res):
                res = s[left:right+1]
            right += 1
    return res
```

time complexity: O(n)

space complexity: O(n)

17. Longest Word in Dictionary through Deleting 通过删除字母匹配到字典里最长单词

<https://leetcode-cn.com/problems/longest-word-in-dictionary-through-deleting/>

给你一个字符串 s 和一个字符串数组 $dictionary$ ，找出并返回 $dictionary$ 中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。

如果答案不止一个，返回长度最长且字母序最小的字符串。如果答案不存在，则返回空字符串。

示例 1:

输入: $s = \text{"abpcplea"}$, $dictionary = [\text{"ale"}, \text{"apple"}, \text{"monkey"}, \text{"plea"}]$
输出: "apple"

示例 2:

输入: $s = \text{"abpcplea"}$, $dictionary = [\text{"a"}, \text{"b"}, \text{"c"}]$
输出: "a"

提示:

- $1 \leq s.length \leq 1000$
- $1 \leq dictionary.length \leq 1000$
- $1 \leq dictionary[i].length \leq 1000$
- s 和 $dictionary[i]$ 仅由小写英文字母组成

02/25/22, 02/26/22, 03/01, 03/18

先对 $dictionary$ 排序，且长度最长为第一优先级，字典序为第二优先级。然后我们只需对 $dictionary$ 进行顺序查找，找到的第一个符合条件的 $word$ 即是答案。我们用 Two Pointers 来进行查找，使用 i , $char$ 代表检查到 $word$ 和 s 中的哪个字符。当 $char == word[i]$ 时，说明找到一个字符，然后再继续找下一个字符，直到找到为止：

```
def findLongestWord(s, dictionary):  
    dictionary.sort(key=lambda x: (-len(x), x))  
    for word in dictionary:  
        i = 0  
        for char in s:  
            if i < len(word) and char == word[i]:  
                i += 1  
        if i == len(word):  
            return word  
    return ""
```

time complexity: $O(m \log m + mn)$

space complexity: $O(1)$

18. Longest Substring with At Most K Distinct Characters

<https://blog.csdn.net/qq508618087/article/details/51049767>

02/25/22, 02/26, 03/01, 03/19

使用Two Pointers滑动窗口：

```
from collections import defaultdict

def lengthOfLongestSubstringKDistinct(s, k):
    dict = defaultdict(int)
    left = right = 0
    res = 0
    while right < len(s):
        dict[s[right]] += 1
        while left <= right and len(dict) > k:
            dict[s[left]] -= 1
            if dict[s[left]] == 0:
                dict.pop(s[left])
            left += 1
        res = max(res, right-left+1)
        right += 1
    return res
```

time complexity: $O(n)$

space complexity: $O(n)$

19. Longest Substring Without Repeating Characters 无重复字符的最长子串

<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

给定一个字符串 s ，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: $s = \text{"abcabcbb"}$

输出: 3

解释: 因为无重复字符的最长子串是 "abc", 所以其长度为 3。注意 "bca" 和 "cab" 也是正确答案。

示例 2:

输入: s = "bbbbbb"

输出: 1

解释: 因为无重复字符的最长子串是 "b", 所以其长度为 1。

示例 3:

输入: s = "pwwkew"

输出: 3

解释: 因为无重复字符的最长子串是 "wke", 所以其长度为 3。

请注意, 你的答案必须是 子串 的长度, "pwke" 是一个子序列, 不是子串。

提示:

- $0 \leq s.length \leq 5 \times 10^4$
- s 由英文字母、数字、符号和空格组成

02/26/22, 03/01, 03/22

```
from collections import defaultdict
```

```
def lengthOfLongestSubstring(s):  
    w = defaultdict(int)  
    left, right = 0  
    res = 0  
    while right < len(s):  
        w[s[right]] += 1  
        while left <= right and right-left+1 > len(w):  
            w[s[left]] -= 1  
            if w[s[left]] == 0:  
                w.pop(s[left])  
            left += 1  
        res = max(res, right-left+1)  
        right += 1  
    return res
```

time complexity: $O(n)$

space complexity: $O(n)$

20. Minimum Size Subarray Sum 长度最小的子数组

<https://leetcode-cn.com/problems/minimum-size-subarray-sum/>

给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其总和大于等于 target 的长度最小的 子数组 $[\text{nums}_l, \text{nums}_{l+1}, \dots, \text{nums}_{r-1}, \text{nums}_r]$ ，并返回其长度。如果不存在符合条件的子数组，返回 0。

示例 1:

输入: target = 7, nums = [2,3,1,2,4,3]

输出: 2

解释: 子数组 [4,3] 是该条件下的长度最小的子数组。

示例 2:

输入: target = 4, nums = [1,4,4]

输出: 1

示例 3:

输入: target = 11, nums = [1,1,1,1,1,1,1,1]

输出: 0

提示:

- $1 \leq \text{target} \leq 10^9$
- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^4$

02/26/22, 03/01, 03/22, 03/28

可以用Two pointers滑动窗口, $\text{res} = \min(\text{res}, \text{right} - \text{left} + 1)$ 这一步要放在while循环里面是因为有可能left减1之后sum不满足条件了, 所以我们要先比较结果:

```
import numpy as np
```

```
def minSubArrayLen(target, nums):
    w_sum = 0
    left = right = 0
    res = np.inf
    while right < len(nums):
        w_sum += nums[right]
        while left <= right and w_sum >= target:
            w_sum -= nums[left]
            res = min(res, right-left+1)
            left += 1
        right += 1
    if res == np.inf:
        return 0
    return res
```

time complexity: $O(n)$
space complexity: $O(1)$

二分查找 (21 - 28):

做二分查找的思路/模版是, 首先定义left, right两个指针, 然后用一个while循环, 循环结束条件初始设置为while left<right。然后计算mid, 用地板除: $mid = left + (right - left) // 2$, 然后根据条件更新left和right。最后, 很重要的一点是, 考虑区间只剩下两个和一个数时, 整个计算逻辑是不是正确的, 然后根据结果调整循环条件, 例如是不是该为while left<=right, 或left, right更新条件对不对, 会不会让while陷入死循环。最后返回结果, 一般是left或right。

21. First Bad Version 第一个错误的版本

<https://leetcode-cn.com/problems/first-bad-version/>

02/15/22, 02/22, 02/26, 03/17

我们可以用Binary Search:

```
def firstBadVersion(n):  
    left, right = 1, n  
    while left < right:  
        mid = left + (right-left) // 2  
        if isBadVersion(mid):  
            right = mid  
        else:  
            left = mid + 1  
    return left
```

time complexity: $O(\log n)$
space complexity: $O(1)$

22. Sqrt(x) x 的平方根

<https://leetcode-cn.com/problems/sqrtx/>

02/15/22, 02/22, 02/26, 03/18

可以用二分法。为什么while循环结束条件是 $left \leq right$ 呢。我们初始时是写的while $left < right$, 然后我们考虑最后区间只剩下两个数时, 例如对8求平方根, 最后区间只剩下2, 3, 这时由于mid是地板除, mid是2, left变为3, 如果这时就结束循环, 得到的结果是不正确的。因此我们把条件设置成while $left \leq right$, 并返回right:

```
def mySqrt(x):
    left, right = 0, x
    while left <= right:
        mid = left + (right-left) // 2
        if mid*mid == x:
            return mid
        elif mid*mid > x:
            right = mid - 1
        else:
            left = mid + 1
    return right
```

time complexity: $O(\log n)$

space complexity: $O(1)$

23. 求解一个数的立方根

02/22/22, 02/26, 03/19

类似于上一题, 可以用二分法:

```
def cubeRoot(x):
    left, right = 0, x
    while left <= right:
        mid = left + (right-left)//2
        if mid*mid*mid == x:
            return mid
        elif mid*mid*mid > x:
            right = mid - 1
        else:
            left = mid + 1
    return right
```

time complexirty: $O(\log n)$

space complexity: $O(1)$

类似的, 如果是要求解一个数的m方根, 同样可用类似的二分法:

```
def Root(x, m):
    left, right = 0, x
    while left <= right:
        mid = left + (right-left)//2
        if mid ** m == x:
            return mid
        elif mid ** m > x:
            right = mid - 1
        else:
            left = mid + 1
    return right
```

24. Find Minimum in Rotated Sorted Array 寻找旋转排序数组中的最小值

<https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/>

02/16/22, 02/22, 02/26, 03/22

用二分法：

```
def findMin(nums):
    left, right = 0, len(nums)-1
    while left < right:
        mid = left + (right-left) // 2
        if nums[mid] > nums[right]:
            left = mid + 1
        elif nums[mid] < nums[right]:
            right = mid
    return nums[left]
```

time complexity: $O(\log n)$

space complexity: $O(1)$

25. Find Minimum in Rotated Sorted Array II 寻找旋转排序数组中的最小值 II

<https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array-ii/>

02/26/22, 03/22

可以用二分法。与上一题不同，当数组中元素可重复时，当 $\text{nums}[\text{mid}] == \text{nums}[\text{right}]$ 时，我们难以判断分界点的指针区间。例如 $[1, 0, 1, 1, 1]$ 和 $[1, 1, 1, 0, 1]$ ，我们无法判断mid该怎么分

界。当`nums[mid]==nums[right]`，我们可以进行`right -= 1`的操作。原因是，第一，此操作不会使数组越界，因为`right>left>=0`。第二，此操作不会使最小值丢失。假设`nums[right]`是最小值，那么它可能是唯一的最小值，也可能不是唯一的。当`nums[right]`是唯一的最小值时，因为我们是做的地板除，是向下取整的，因此`mid!=right`。当它不是唯一的最小值时，由于`mid<right`，那么 我们进行`right -= 1`的操作后，最小值还是在区间中。时间复杂度还是 $O(\log n)$ ，但是在worst case下，时间复杂度是 $O(n)$ ，例如数组中所有元素值都相等：

```
def findMin(nums):
    left, right = 0, len(nums)-1
    while left < right:
        mid = left + (right-left) // 2
        if nums[mid] > nums[right]:
            left = mid + 1
        elif nums[mid] < nums[right]:
            right = mid
        else:
            right -= 1
    return nums[left]
```

time complexity: $O(\log n)$, worst case $O(n)$

space complexity: $O(1)$

26. Search Insert Position 搜索插入位置

<https://leetcode-cn.com/problems/search-insert-position/>

02/16/22, 02/22, 02/26, 03/22

看到sorted array的时候，敏感的想到能不能用二分法。

```
def searchInsert(self, nums: List[int], target: int) -> int:
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = left + (right-left) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            right = mid - 1
        else:
            left = mid + 1
    return left
```

time complexity: $O(\log n)$

space complexity: $O(1)$

27. Find First and Last Position of Element in Sorted Array 在排序数组中查找元素的第一个和最后一个位置

<https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

02/26/22, 03/22

我们可以用二分法来分别查找左右边界：

```
def searchRange(nums, target):
    def searchLeft(nums, target):
        left, right = 0, len(nums)-1
        while left < right:
            mid = left + (right-left) // 2
            if nums[mid] >= target:
                right = mid
            else:
                left = mid + 1
        return left

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

    if not nums or nums[searchLeft(nums, target)] != target:
        return [-1, -1]
    else:
        return [searchLeft(nums, target), searchRight(nums, target)]
```

time complexity: $O(\log n)$

space complexity: $O(1)$

28. Median of Two Sorted Arrays 寻找两个正序数组的中位数

<https://leetcode-cn.com/problems/median-of-two-sorted-arrays/>

02/27/22, 03/22, 03/28

用二分法, 找median等同于找第k小的数, 那么就取 $p1 = \text{nums1}[k/2-1]$, $p2 = \text{nums2}[k/2-1]$ 进行比较。取 $\text{pivot} = \min(p1, p2)$, 两个数组中小于等于pivot的元素总计不会超过 $(k/2-1) + (k/2-1) = k-2$ 个, 这样, pivot本身最大也只能是第k-1小的元素。如果 $\text{pivot} = p1$, 那么 $\text{nums1}[0, \dots, k/2-1]$ 都不可能是第k小的元素, 我们可以把这些元素都删掉, 剩下的作为新的 nums1 数组。对 nums2 同理:

```
def findMedianSortedArrays(nums1, nums2):
    def getKthSmallestElement(k):
        i1, i2 = 0, 0
        while True:
            if i1 == m: return nums2[i2+k-1]
            if i2 == n: return nums1[i1+k-1]
            if k == 1: return min(nums1[i1], nums2[i2])

            p1 = min(i1 + k//2 - 1, m-1)
            p2 = min(i2 + k//2 - 1, n-1)
            if nums1[p1] <= nums2[p2]:
                k -= p1 - i1 + 1
                i1 = p1 + 1
            else:
                k -= p2 - i2 + 1
                i2 = p2 + 1

        m, n = len(nums1), len(nums2)
        if (m+n) % 2 == 1:
            kth = (m+n+1)//2
            return getKthSmallestElement(kth)
        else:
            kth = (m+n)//2
            return (getKthSmallestElement(kth) + getKthSmallestElement(kth+1)) / 2
```

time complexity: $O(\log(m+n))$

space complexity: $O(1)$

29. Majority Element 多数元素

<https://leetcode-cn.com/problems/majority-element/>

30. Search a 2d Matrix II 搜索二维矩阵 II

<https://leetcode-cn.com/problems/search-a-2d-matrix-ii/>

动态规划 (29 -):

31. Climbing Stairs 爬楼梯

<https://leetcode-cn.com/problems/climbing-stairs/>

02/27/22, 03/17

可用DP动态规划。定义一个数组dp, dp[i]表示走到第i阶的方法数。因为每一次可以走一步或两步到达, 因此状态转移方程是: $dp[i] = dp[i-1] + dp[i-2]$:

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

time complexity: $O(n)$

space complexity: $O(n)$

进一步的, 我们可以对动态规划的空间进行压缩, 因为dp[i]只与dp[i-1]和dp[i-2]有关, 因此可以只用两个变量来存储dp[i-1]和dp[i-2], 从而使得空间复杂度优化为 $O(1)$:

```
def climbStairs(n):
    pre1, pre2 = 1, 1
    cur = 1
    for i in range(2, n+1):
        cur = pre1 + pre2
        pre1 = pre2
        pre2 = cur
    return cur
```

time complexity: $O(n)$

space complexity: $O(1)$

32. House Robber 打家劫舍

<https://leetcode-cn.com/problems/house-robber/>

02/27/22, 03/18

可以用 DP 动态规划。dp[i] 表示抢劫到第 i 个房子时，可以抢劫的最大数量。我们考虑 dp[i]，此时可以抢劫的最大数量有两种可能：一种是我们选择不抢劫这个房子，此时累计的金额为 dp[i-1]。另一种是我们选择抢劫这个房子，因为我们不能抢劫第 i-1 个房子，此时累计的最大金额是 nums[i-1]+dp[i-2]：

```
def rob(nums):
    n = len(nums)
    if n == 1: return nums[0]
    dp = [0] * (n+1)
    dp[0], dp[1] = 0, nums[0]
    for i in range(2, n+1):
        dp[i] = max(dp[i-1], dp[i-2]+nums[i-1])
    return dp[n]
```

time complexity: O(n)

space complexity: O(n)

进一步的，我们可以对动态规划的空间进行压缩，因为 dp[i] 只与 dp[i-1] 和 dp[i-2] 有关，因此可以只用两个变量来存储 dp[i-1] 和 dp[i-2]，从而使得空间复杂度优化为 O(1)：

```
def rob(nums):
    n = len(nums)
    if n == 1: return nums[0]
    pre1, pre2 = 0, nums[0]
    cur = 0
    for i in range(2, n+1):
        cur = max(pre2, pre1+nums[i-1])
        pre1 = pre2
        pre2 = cur
    return cur
```

time complexity: O(n)

space complexity: O(1)

33. Minimum Path Sum 最小路径和

<https://leetcode-cn.com/problems/minimum-path-sum/>

02/27/22, 03/19

可以采用 DP 动态规划：

```
def minPathSum(grid):
```

```

r, c = len(grid), len(grid[0])
dp = [[0] * c for _ in range(r)]
for i in range(r):
    for j in range(c):
        if i == 0 and j == 0:
            dp[i][j] = grid[i][j]
        elif i == 0:
            dp[i][j] = dp[i][j-1] + grid[i][j]
        elif j == 0:
            dp[i][j] = dp[i-1][j] + grid[i][j]
        else:
            dp[i][j] = min(dp[i][j-1], dp[i-1][j]) + grid[i][j]
return dp[r-1][c-1]

```

time complexity: $O(mn)$

space complexity: $O(mn)$

34. Maximal Square 最大正方形

<https://leetcode-cn.com/problems/maximal-square/>

02/27/22, 03/22

可以用DP动态规划，对于每个位置，如果该位置的值是1，则 $dp(i,j)$ 的值由其上方、左方和左上方的三个相邻位置的dp值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加1：

```

import numpy as np

def maximalSquare(matrix):
    r, c = len(matrix), len(matrix[0])
    dp = [[0]*c for _ in range(r)]
    for i in range(r):
        for j in range(c):
            if matrix[i][j] == "1":
                if i == 0 or j == 0:
                    dp[i][j] = 1
                else:
                    dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])+1
    return np.max(dp) ** 2

```

time complexity: $O(mn)$

space complexity: $O(mn)$

35. Word Break 单词拆分

<https://leetcode-cn.com/problems/word-break/>

03/02/22, 03/22, 03/28

我们可以用DP。dp[i]表示前面i个字符组成的字符串能不能用wordDict里的单词表示：

```
def wordBreak(s, wordDict):
    n = len(s)
    dp = [False] * (n+1)
    dp[0] = True
    // 此处的i对应的是s里的i，因为dp的长度比s的长度要大1，
    // 所以s[i]对应的是dp[i+1]
    for i in range(n):
        // j的右侧要能确保能取到的最大值是n，因为我们要得到dp[n]的值
        for j in range(i+1, n+1):
            // dp[i]对应的是s[i-i]，表明前面i个字符组成的字符串在不在
            // 所以s[i:j]从i开始。
            if dp[i] and s[i:j] in wordDict:
                dp[j] = True
    return dp[n]
```

time complexity: $O(n^2)$

space complexity: $O(n)$

36. Maximum Subarray 最大子数组和

<https://leetcode-cn.com/problems/maximum-subarray/>

02/17/22, 02/21, 02/22, 03/22

可以用Dynamic Programming，先找到subarray的最优解，然后得到全局的最优解：

```
def maxSubArray(nums):
    cursum = maxsum = nums[0]
    for num in nums[1:]:
        cursum = max(num, num+cursum)
        maxsum = max(maxsum, cursum)
    return maxsum
```

time complexity: $O(n)$

space complexity: $O(1)$

37. Longest Increasing Subsequence 最长递增子序列

<https://leetcode-cn.com/problems/longest-increasing-subsequence/>

02/18/22, 02/21, 03/20, 03/22

我们可以用Dynamic Programming, 把大问题转化成几个小问题, 如果我们能找到以每一个元素作为结尾的数列的LIS, 然后再求所有元素LIS的最大值, 就得到了我们想要的结果。

```
def lengthOfLIS(nums):
    if len(nums) == 1:
        return 1
    maxnums = [1]*len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                maxnums[i] = max(maxnums[i], maxnums[j]+1)
    return max(maxnums)
```

time complexity: $O(n^2)$

space complexity: $O(n)$

38. Edit Distance 编辑距离

<https://leetcode-cn.com/problems/edit-distance/>

02/18/22, 02/21, 02/22, 03/22

我们可以用Dynamic Programming, 把大问题转化成几个小问题, 我们先生成一个二维数组dp, dp[i][j]表示长度分别为i和j的两个词的距离

```
def minDistance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[None]*(n+1) for _ in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif 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[m][n]

```

time complexity: $O(mn)$

space complexity: $O(mn)$

39. Longest Common Subsequence 最长公共子序列

<https://leetcode-cn.com/problems/longest-common-subsequence/>

03/02/22, 03/23

可以用DP。定义一个二维dp, $dp[i][j]$ 表示分别有i个字符和j个字符的两个字符串的最长公共子序列:

```

def longestCommonSubsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n+1) for _ in (m+1)]
    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], dp[i-1][j-1])
    return dp[m][n]

```

time complexity: $O(mn)$

space complexity: $O(mn)$

40. Coin Change 零钱兑换

<https://leetcode-cn.com/problems/coin-change/>

03/25/22, 03/28

可以用DP动态规划法。

```

def coinChange(coins, amount):
    dp = [float("inf")] * (amount+1)
    dp[0] = 0
    for i in range(1, amount+1):
        for coin in coins:
            if i >= coin:

```

```

                                dp[i] = min(dp[i], dp[i-coin] + 1)
        if dp[amount] == float("inf"):
            return -1
        else:
            return dp[amount]

```

time complexity: $O(n)$
 space complexity: $O(n)$

41. Best Time to Buy and Sell Stock 买卖股票的最佳时机

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/>

02/24/22

设置一个minprice和一个maxprofit, 从左到右遍历, 保持当前的minprice, 并计算当前的profit, 更新maxprofit:

```

def maxProfit(prices):
    maxprofit = 0
    minprice = prices[0]
    for price in prices[1:]:
        minprice = min(minprice, price)
        profit = price - minprice
        maxprofit = max(maxprofit, profit)
    return maxprofit

```

time complexity: $O(n)$
 space complexity: $O(1)$

42. Longest-palindromic-substring 最长回文子串

<https://leetcode-cn.com/problems/longest-palindromic-substring/>

03/25/22, 03/28

我们可以用动态规划来解。dp[i][j]表示字符串s[i:j+1]是否为回文子串。在写状态转移方程的时候, 我们需要先考虑边界条件: 当(j-1)-(i+1)<=0时, 只要当s[i]==s[j]时就是回文, 例如s只有两个或三个字符。然后, 当s[i]==s[j]并且dp[i+1][j-1]==True时, dp[i][j]就是True。在初始状态时, 当s的长度小于2时, 我们需要先做个判断:

```

def longestPalindrome(s):
    n = len(s)

```

```

        if n < 2:
            return s
        dp = [[False]*n for _ in range(n)]
        res = s[0]
        for j in range(1, n):
            for i in range(j):
                if (j-1)-(i+1) <= 0:
                    if s[i] == s[j]:
                        dp[i][j] = True
                else:
                    if s[i] == s[j] and dp[i+1][j-1]:
                        dp[i][j] = True
            if dp[i][j]:
                if j-i+1 > len(res):
                    res = s[i:j+1]
        return res

```

time complexity: $O(n^2)$

space complexity: $O(n^2)$

回溯算法 (-):

43. Permutations 全排列

<https://leetcode-cn.com/problems/permutations/>

03/08/22, 03/17

我们可以用回溯法，按顺序枚举出每一位数字可能出现的情况，已经选择的数字在当前要选择的数字中不能出现。比如最开始，我们可以先写以1开头的全排列，即1加上[2,3]的全排列。再写以2开头的全排列。最后写以3开头的全排列。这种搜索的方法我们可以用DFS深度优先搜索来实现。为什么不用BFS？使用广度优先遍历也可以解，但是从性能来说是不划算的。因为使用BFS就得使用队列，然后编写节点类，队列中需要存储每一步的状态信息，需要存储的数据很大，但是我们真正用到的却很少。

还有一个需要特别注意的地方是，我们在添加path到res中时，不能直接添加path，因为path变量在python传参的过程中，复制的是变量的地址，这些地址被添加到res变量中，但实际上指向的是同一块内存地址，因此我们会看到几个空的对象列表。解决的办法是，我们用path[:]来拷贝地址指向的值并加入到res中：

```
def permute(nums):
```

```

def dfs(nums, size, depth, path, used, res):
    if depth == size:
        res.append(path[:])
        return

    for i in range(size):
        if not used[i]:
            path.append(nums[i])
            used[i] = True
            dfs(nums, size, depth+1, path, used, res)
            path.pop()
            used[i] = False

    size = len(nums)
    depth = 0
    path = []
    used = [False for _ in range(size)]
    res = []
    dfs(nums, size, depth, path, used, res)
    return res

```

time complexity: $O(N \times N!)$

space complexity: $O(N \times N!)$

44. Combination Sum 组合总和

<https://leetcode-cn.com/problems/combination-sum/>

03/08/22, 03/18

我们可以用回溯法，先以target为根结点，然后用根结点的值减去数组里的元素值，得到子节点的值。通过这种方式来创建分支。减到0或者负数的时候停止。我们可以用DFS深度优先遍历来实现。有一个需要注意的问题是，由于题目中说每一个元素可以重复使用，如果我们考虑数组中所有的元素，会出现重复的列表。为了解决这个问题，我们在每一次搜索的时候设置下一轮搜索的起点。也就是说，从每一层的第二个节点开始，都不能再搜索产生同一层节点已经使用过的数组里的元素。

需要注意的是，此题使用begin而上一题不使用begin的原因是。在sum问题中，1+2和2+1是重复的，需要去重。而在combination问题中，[1,2]和[2,1]是两个不同的排列组合，不需要去重：

```

def combinationSum(candidates, target):
    def dfs(candidates, size, path, target, begin, res):
        if target == 0:
            res.append(path[:])

```

```

        return
    elif target < 0:
        return
    for i in range(begin, size):
        path.append(candidates[i])
        target -= candidates[i]
        dfs(candidates, size, path, target, i, res)
        path.pop()
        target += candidates[i]

    size = len(candidates)
    path = []
    begin = 0
    res = []
    dfs(candidates, size, path, target, begin, res)
    return res

```

45. Permutations II 全排列 II

<https://leetcode-cn.com/problems/permutations-ii/>

03/08/22, 03/19, 03/28

可以用回溯法。此题的序列中元素可重复，而且又要求返回的结果中不能用重复元素。所以我们的思路是，在遍历的过程中，一边遍历一边检测，在一定会产生重复结果的地方剪枝。我们可以在搜索之前对候选数组排序，一旦发现某个分支搜索下去可能搜索到重复的元素就停止搜索，这样结果集中不会包含重复列表：

```

def permuteUnique(nums):
    def dfs(nums, size, path, depth, used, res):
        if depth == size:
            res.append(path[:])
            return

        for i in range(size):
            if not used[i]:
                if i > 0 and nums[i] == nums[i-1] and not used[i-1]:
                    continue
                path.append(nums[i])
                used[i] = True
                dfs(nums, size, path, depth+1, used, res)
                path.pop()
                used[i] = False

    nums.sort()
    size = len(nums)

```

```

    path = []
    depth = 0
    used = [False for _ in range(size)]
    res = []
    dfs(nums, size, path, depth, used, res)
    return res

```

time complexity: $O(N \times N!)$

space complexity: $O(N \times N!)$

46. Combination Sum II 组合总和 II

<https://leetcode-cn.com/problems/combination-sum-ii/>

03/08/22, 03/22, 03/28

我们可以使用回溯法。对于求和问题使用begin因为1+2和2+1是重复的，需要去重。使用used是因为数组中每个元素只能使用一次，所以我们要track哪些元素已经被使用过。此外，因为数组中可能包含重复的元素，我们还需要在可能产生重复的地方进行剪枝，一般的做法是先排序再进行剪枝：

```

def combinationSum2(candidates, target):
    def dfs(candidates, size, path, target, begin, used, res):
        if target == 0:
            res.append(path[:])
            return
        elif target < 0:
            return

        for i in range(begin, size):
            if not used[i]:
                if i > begin and candidates[i] == candidates[i-1] and not used[i-1]:
                    continue
                path.append(candidates[i])
                used[i] = True
                target -= candidates[i]
                dfs(candidates, size, path, target, i, used, res)
                path.pop()
                used[i] = False
                target += candidates[i]

    candidates.sort()
    size = len(candidates)
    path = []
    begin = 0

```

```

        used = [False for _ in range(size)]
        res = []
        dfs(candidates, size, path, target, begin, used, res)
        return res

```

47. Subsets 子集

<https://leetcode-cn.com/problems/subsets/>

03/08/22, 03/22

每一个元素都面临着两种选择，选或者不选，我们可以用DFS求解。当depth==size时说明已经遍历到最后一层：

```

def subsets(nums):
    def dfs(nums, size, depth, path, res):
        if depth == size:
            res.append(path[:])
            return
        dfs(nums, size, depth+1, path, res)
        path.append(nums[depth])
        dfs(nums, size, depth+1, path, res)
        path.pop()

    size = len(nums)
    depth = 0
    path = []
    res = []
    dfs(nums, size, depth, path, res)
    return res

```

48. Number of Islands 岛屿数量

<https://leetcode.com/problems/number-of-islands/>

我们可以用DFS深度优先搜索来搜索整个网络。如果一个位置为1，则以其为起始节点开始进行深度优先搜索。在深度优先搜索的过程中，每个搜索到的1都会被重新标记为0：

03/08/22

```

def numIslands(grid):
    def dfs(grid, m, n, i, j):
        grid[i][j] = 0

```

```

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

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

```

time complexity: $O(MN)$

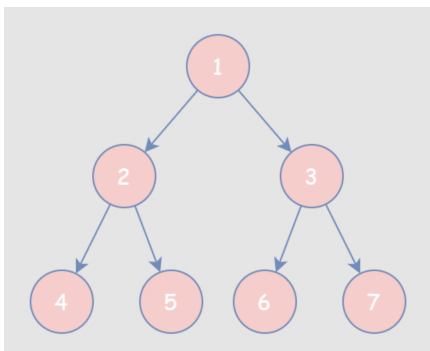
space complexity: $O(MN)$

二叉树

49. 二叉树的前序遍历

<https://leetcode-cn.com/problems/binary-tree-preorder-traversal/>

03/10/22, 03/17



前序遍历的顺序是按照 根结点->左孩子->右孩子 的方式遍历。上图按照前序遍历的结果是 1 2 4 5 3 6 7。我们可以分别采用递归和迭代的解法：

递归解法是：

```

def preorderTraversal(root):
    def preorder(root):

```

```

        if not root:
            return
        res.append(root.val)
        preorder(root.left)
        preorder(root.right)
    res = []
    preorder(root)
    return res

```

迭代解法是：

迭代解法的思路是，我们使用栈来进行迭代，过程是：我们引入一个变量cur来代表当前节点，先把根结点赋予cur，然后将根结点和所有的左孩子加入栈中并加入到结果中，直到cur为空，我们用一个while循环实现这个过程。然后，每弹出一个栈顶元素tmp，我们就到达它的右孩子，再将这个节点当作cur重新按上面的步骤来一遍，直至栈为空，我们用另一个while循环来实现这个过程：

```

def preorderTraversal(root):
    if not root:
        return []
    cur, stack, res = root, [], []
    while cur or stack:
        while cur:
            res.append(cur.val)
            stack.append(cur)
            cur = cur.left
        tmp = stack.pop()
        cur = tmp.right
    return res

```

time complexity: $O(n)$

space complexity: 最坏情况 $O(n)$

50. 二叉树的中序遍历

<https://leetcode-cn.com/problems/binary-tree-inorder-traversal/>

03/10/22, 03/18

中序遍历的顺序是按照 左孩子->根节点->右孩子 的方式遍历。上图按照中序遍历的结果是 4 2 5 1 6 3 7。我们可以分别采用递归和迭代的解法：

递归解法是：

```

def inorderTraversal(root):

```

```

def inorder(root):
    if not root:
        return
    inorder(root.left)
    res.append(root.val)
    inorder(root.right)

res = []
inorder(root)
return res

```

迭代解法是：

迭代解法的思路是，和前序遍历相似，我们使用栈来进行迭代，不同的是，只是在出栈的时候才将节点tmp的值加入到结果中。具体过程是：我们引入一个变量cur来代表当前节点，先把根节点赋予cur，然后将根节点和所有的左孩子加入栈中，直到cur为空，我们用一个while循环实现这个过程。然后，每弹出一个栈顶元素tmp，我们先将tmp的值加入到结果中，然后到达它的右孩子，再将这个节点当作cur重新按上面的步骤来一遍，直至栈为空，我们用另一个while循环来实现这个过程：

```

def inorderTraversal(root):
    if not root:
        return []
    cur, stack, res = root, [], []
    while cur or stack:
        while cur:
            stack.append(cur)
            cur = cur.left
        tmp = stack.pop()
        res.append(tmp.val)
        cur = tmp.right
    return res

```

time complexity: $O(n)$

space complexity: 最坏情况 $O(n)$

51. 二叉树的后序遍历

<https://leetcode-cn.com/problems/binary-tree-postorder-traversal/>

03/10/22, 03/19, 03/28

后序遍历的顺序是按照 左孩子->右孩子->根节点 的方式遍历。上图按照中序遍历的结果是 4 5 2 6 7 3 1。我们可以分别采用递归和迭代的解法：

递归解法是：

```
def postorderTraversal(root):
    def postorder(root):
        if not root:
            return
        postorder(root.left)
        postorder(root.right)
        res.append(root.val)
    res = []
    postorder(root)
    return res
```

迭代解法是：

迭代解法的思路是，后序遍历我们可以看作是前序遍历的变种，前序是根->左->右，后序是左->右->根。因此我们后序遍历时我们可以先做根->右->左，然后把得到的结果做翻转，就得到了后序遍历的结果。同样的，我们使用栈来进行迭代，过程是：我们引入一个变量cur来代表当前节点，先把根结点赋予cur，然后将根结点和所有的右孩子加入栈中并加入到结果中，直到cur为空，我们用一个while循环实现这个过程。然后，每弹出一个栈顶元素tmp，我们就到达它的左孩子，再将这个节点当作cur重新按上面的步骤来一遍，直至栈为空，我们用另一个while循环来实现这个过程，最后把结果做翻转，我们就得到了后序遍历的结果：

```
def postorderTraversal(root):
    if not root:
        return []
    cur, stack, res = root, [], []
    while cur or stack:
        while cur:
            res.append(cur.val)
            stack.append(cur)
            cur = cur.right
        tmp = stack.pop()
        cur = tmp.left
    return res[::-1]
```

time complexity: $O(n)$

space complexity: 最坏情况 $O(n)$

52. 二叉树的层序遍历

<https://leetcode-cn.com/problems/binary-tree-level-order-traversal/>

03/10/22, 03/22, 03/28

二叉树的层次遍历的迭代方法与前面不同，因为前面都采用了深度优先搜索的方式，而层次遍历使用了广度优先搜索。深度优先搜索用栈来实现，广度优先搜索主要用队列来实现，队列遵循先进先出的原则。具体步骤为：先初始化队列q，并将根结点加入到队列中。当队列不为空时，队列中弹出节点node，加入到结果中；如果左子树非空，左子树加入队列；如果右子树非空，右子树加入队列。由于题目要求每一层保存在一个子数组中，所以我们额外加入了level保存每层的遍历结果，并使用for循环来实现。

```
def levelOrder(root):
    if not root:
        return []
    q, res = [root], []
    while q:
        n = len(q)
        level = []
        for _ in range(n):
            node = q.pop(0)
            level.append(node.val)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
        res.append(level)
    return res
```

time complexity: $O(n)$

space complexity: $O(n)$

53. 二叉树展开为链表

<https://leetcode-cn.com/problems/flatten-binary-tree-to-linked-list/>

03/10/22, 03/22

我们可以先用先序遍历得到各节点的访问顺序之后，再更新给个节点的左右子节点的信息

```
def flatten(root):
    def preorder(root):
        if not root:
            return
        res.append(root)
```

```

        preorder(root.left)
        preorder(root.right)
    res = []
    preorder(root)
    for i in range(1, len(res)):
        pre, cur = res[i-1], res[i]
        pre.left = None
        pre.right = cur

```

time complexity: $O(n)$

space complexity: $O(n)$

上一种方法借助前序遍历，前序遍历过程中需要使用栈存储节点，我们还可以采用另一种方法使得空间复杂度是 $O(1)$ 。我们注意到前序遍历访问各节点的顺序是根节点，左子树，右子树。如果一个节点的左子节点为空，则该节点不需要进行展开操作。如果一个节点的左子节点不为空，则该节点的左子树中的最后一个节点被访问之后，该节点的右子节点被访问。该节点的左子树中最后一个被访问的节点是左子树中的最右边的节点，也是该节点的前驱节点。因此，问题转化成了寻找当前节点的前驱节点。具体做法是，对于当前节点，如果其左子节点不为空，则在其左子树中找到最右边的节点，作为前驱节点，将当前节点的右子节点赋给前驱节点的右子节点，然后将当前节点的左子节点赋给当前节点的右子节点，并将当前节点的左子节点设为空。对当前节点处理结束后，继续处理链表中的下一个节点，直到所有节点都处理结束。

```

def flatten(root):
    cur = root
    while cur:
        if cur.left:
            pivot = cur.left
            while pivot.right:
                pivot = pivot.right
            pivot.right = cur.right
            cur.right = cur.left
            cur.left = None
        cur = cur.right

```

time complexity: $O(n)$

space complexity: $O(1)$

54. 二叉树的锯齿形层序遍历

<https://leetcode-cn.com/problems/binary-tree-zigzag-level-order-traversal/>

03/10/22, 03/22

此题为层序遍历的变体。我们可以定义一个counter, 来跟踪遍历到了哪一层, 然后把level定义成一个双端队列deque, 根据counter的奇偶来决定把这一层的节点加到deque的头还是尾:

```
from collections import deque
```

```
def zigzagLevelOrder(root):
    if not root:
        return []
    q, res = [root], []
    cnt = 1
    while q:
        n = len(q)
        level = deque()
        for _ in range(n):
            node = q.pop(0)
            if cnt%2 == 0:
                level.appendleft(node.val)
            else:
                level.append(node.val)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
        res.append(list(level))
        cnt += 1
    return res
```

time complexity: $O(n)$

space complexity: $O(n)$

55. 二叉树的右视图

<https://leetcode-cn.com/problems/binary-tree-right-side-view/>

03/10/22, 03/22

此题为层序遍历的变体。我们定义一个变量rightside, 每一层不断地更新这个变量, 直到遍历到这一层最右边为止, 此时的rightside就是这一层最右边的节点:

```
def rightSideView(root):
    if not root:
        return []
    q, res = [root], []
    while q:
```

```

        for _ in range(len(q)):
            node = q.pop(0)
            rightside = node.val
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
            res.append(rightside)
    return res

```

time complexity: $O(n)$

space complexity: $O(n)$

56. 二叉树的最近公共祖先

<https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/>

03/11/22, 03/23

最近公共祖先的定义是：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p, q的最近公共祖先。根据以上定义，如果root是p, q的最近公共祖先，则只可能是以下情况之一：p和q分别在root的左右子树中；p==root, 且q在root的左或右子树中；q==root, 且p在root的左或右子树中。因此，我们可以使用递归对二叉树进行遍历，递归终止的条件是：当越过叶节点时，返回None；当root==p或q时，直接返回root。

递归时，如果root的左子树为空，说明p和q都在右子树，其最近公共祖先也在右子树；如果root的右子树为空，说明p和q都在左子树，其最近公共祖先也在左子树；如果p和q都不为空，说明p和q分别在root的左右子树，root就是p和q的最近公共祖先：

```

def lowestCommonAncestor(root, p, q):
    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: return right
    if not right: return left
    return root

```

time complexity: $O(n)$

space complexity: $O(n)$

57. 二叉树中的最大路径和

<https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/>

03/11/22, 03/23

我们可以先实现一个递归函数来计算二叉树中每个节点的最大贡献值，也就是说，在以该节点为根节点的子树中寻找以该节点为起点的一条路径，使得该路径上的节点值之和最大。具体计算过程是：空节点的最大贡献值为0，非空节点最大贡献值等于节点值与左子节点或右子节点（取较大者）最大贡献值之和。另外需要注意的是，当左子节点或右子节点的最大贡献值是负数时，我们舍弃这个子节点以及它所链接的其他子节点。然后，我们设置一个全局变量maxsum，在递归的过程中不断地更新这个maxsum。最后得到的值就是最大值。

因为maxsum是全局变量，并且要不断地递归更新它的值，所以我们这里在主函数外面定义一个新函数，并且利用类的self属性来调用这个全局变量：

```
def maxNodeSum(node):
    if not node:
        return 0
    leftsum = max(self.maxNodeSum(node.left), 0)
    rightsum = max(self.maxNodeSum(node.right), 0)
    self.maxsum = max(self.maxsum, node.val+leftsum+rightsum)
    return node.val + max(leftsum, rightsum)

def maxPathSum(root):
    self.maxsum = float('-inf')
    self.maxNodeSum(root)
    return self.maxsum
```

time complexity: $O(n)$

space complexity: $O(n)$

58. 二叉树的最大深度

<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/>

03/11/22, 03/23

此题可以看作是层序遍历的变体。我们采用广度优先搜索的方法，定义一个队列和一个counter，每遍历一层，counter就加1，遍历完后，返回结果：

```
def maxDepth(root):
    if not root:
        return 0
    q, res = [root], 0
    while q:
```

```

        for _ in range(len(q)):
            node = q.pop(0)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)

        res += 1
    return res

```

time complexity: $O(n)$

space complexity: 空间消耗取决于存储的节点个数, 最坏情况下是 $O(n)$

59. 翻转二叉树

<https://leetcode-cn.com/problems/invert-binary-tree/>

03/11/22, 03/23

其实这道题我们要做的就是先交换当前节点的左右子树, 然后再递归交换当前节点的左子树和右子树:

```

def invertTree(root):
    if not root:
        return
    root.left, root.right = root.right, root.left
    self.invertTree(root.left)
    self.invertTree(root.right)
    return root

```

time complexity: $O(n)$

space complexity: $O(n)$

60. 从前序与中序遍历序列构造二叉树

<https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>

03/11/22, 03/23

先序遍历的顺序是根节点, 左子树, 右子树。中序遍历的顺序是左子树, 根节点, 右子树。所以我们只需要根据先序遍历得到根节点, 然后在中序遍历中找到根结点的位置, 它的左边就是左子树的节点, 右边就是右子树的节点。生成左子树和右子树后, 递归的重复同样的过程。为了在中序遍历中找到根结点的位置, 我们可以首先建立一个字典, key和value分别是节点和它对应的index:

```

def buildTree(preorder, inorder):
    def build(pre_s, pre_e, in_s, in_e):
        if pre_s > pre_e:
            return
        pre_root = pre_s
        in_root = d[preorder[pre_root]]
        root = TreeNode(preorder[pre_root])
        left_size = in_root - in_s
        root.left = build(pre_s+1, pre_s+left_size, in_s, in_root-1)
        root.right = build(pre_s+left_size+1, pre_e, in_root+1, in_e)
        return root

    d = {v:i for i, v in enumerate(inorder)}
    n = len(preorder)
    return build(0, n-1, 0, n-1)

```

time complexity: $O(n)$
space complexity: $O(n)$

61. 二叉树的直径

<https://leetcode-cn.com/problems/diameter-of-binary-tree/>

03/12/22, 03/23

一条路径的长度为该路径经过的节点数减一，而任意一条路径都可以看作由某个节点为起点，从其左子树和右子树向下遍历的路径拼接得到。假设我们知道对于某节点的左子树向下遍历经过最多的节点数L和其右子树向下遍历经过最多的节点数R，那么以该节点为起点的路径经过节点数的最大值即为L+R+1。

```

def diameterOfBinaryTree(root):
    self.res = 1
    def maxNodes(root):
        if not root:
            return 0
        left = maxNodes(root.left)
        right = maxNodes(root.right)
        self.res = max(self.res, left + right + 1)
        return max(left, right) + 1
    maxNodes(root)
    return self.res - 1

```

time complexity: $O(n)$
space complexity: $O(n)$

62. 对称二叉树

<https://leetcode-cn.com/problems/symmetric-tree/>

03/12/22, 03/23

我们可以用递归来解决这个问题。将根节点的左子树记作left, 右子树记作right。比较left是否等于right, 不等的话直接返回就可以了。如果相等, 就递归的比较left.left和right.right, 以及left.right, right.left是否相等:

```
def isSymmetric(root):
    def dfs(left, right):
        if (not left) and (not right):
            return True
        if (not left) or (not right):
            return False
        if left.val != right.val:
            return False
        return dfs(left.left, right.right) and dfs(left.right, right.left)

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

time complexity: $O(n)$

space complexity: $O(n)$ 。空间复杂度是递归的深度, 也就是跟树的高度有关, 最坏情况下树变成一个链表结构, 高度时 n 。

高频题 (-):

63. Kth Largest Element in an Array 数组中的第K个最大元素

<https://leetcode-cn.com/problems/kth-largest-element-in-an-array/>

02/22/22

我们可以采用QuickSort的分治思想, 首先随机选择一个pivot, 最终pivot的左边均小于pivot, 右边均大于pivot。由于递归只会执行其中一个分支, 在pivot随机的情况下, 平均时间复杂度是 $O(n)$, 最坏情况是 $O(n^2)$:

```
import numpy as np

def findKthLargest(nums, k):
    pivot = np.random.randint(0, len(nums))
```

```

left = [num for num in nums if num < nums[pivot]]
middle = [num for num in nums if num == nums[pivot]]
right = [num for num in nums if num > nums[pivot]]
if len(right) >= k:
    return findKthLargest(right, k)
elif len(right) + len(middle) < k:
    return findKthLargest(left, k-len(right)-len(middle))
else:
    return middle[0]

```

time complexity: $O(n)$

space complexity: $O(n)$

64. 3Sum 三数之和

<https://leetcode-cn.com/problems/3sum/>

02/28/22, 3/18

可以用排序+Two Pointers:

```

def threeSum(nums):
    n = len(nums)
    if n < 3: return []
    nums.sort()
    res = []
    for i in range(n):
        if nums[i] > 0: return res
        if i > 0 and nums[i] == nums[i-1]: continue
        left = i + 1
        right = n - 1
        while left < right:
            if nums[i]+nums[left]+nums[right] == 0:
                res.append([nums[i], nums[left], nums[right]])
                while left < right and nums[left] == nums[left+1]:
                    left += 1
                while left < right and nums[right] == nums[right-1]:
                    right -= 1
                left += 1
                right -= 1
            elif nums[i]+nums[left]+nums[right] > 0:
                right -= 1
            else:
                left += 1
    return res

```

time complexity: $O(n^2)$
space complexity: $O(n)$

65. Search in Rotated Sorted Array 搜索旋转排序数组

<https://leetcode-cn.com/problems/search-in-rotated-sorted-array/>

02/28/22, 03/22

可以用二分法，可以观察到的一个重要的点是，在旋转数组中，如果以某一个索引为分界线，那么一定有一个方向的子数列是升序排列：

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

time complexity: $O(\log n)$
space complexity: $O(1)$

66. Find Peak Element 寻找峰值

<https://leetcode-cn.com/problems/find-peak-element/>

03/01/22, 03/22

可以用二分法。首先, 因为 $\text{nums}[-1]=\text{nums}[n]=-\infty$, 所以, 如果数组严格递增, 那么峰值在最右边处, 如果数组严格递减, 那么峰值在最左边处。如果数组不是严格递增或者严格递减, 那么在 $\text{nums}[1:n-1]$ 处一定存在一个峰值。下面就简单了, 比较mid的值和左右两边值的大小: 如果mid值比左右都大, 那么mid是峰值; 如果mid值比左边小, 那么左边一定有峰值, 原因是, 因为mid比左边小, 所以左边要么递减, 要么不递减, 递减的话, 最左边是峰值, 不递减的话, 一定有峰值; 同理, 如果mid值比右边小, 那么右边一定有峰值。只有这三种情况:

```
def findPeakElement(nums):
    n = len(nums)
    if n == 1: return 0
    if nums[0] > nums[1]: return 0
    if nums[n-1] > nums[n-2]: return n-1

    left, right = 0, n-1
    while left <= right:
        mid = left + (right-left) // 2
        if mid >= 1 and nums[mid-1] < nums[mid] > nums[mid+1]:
            return mid
        elif mid >= 1 and nums[mid-1] > nums[mid]:
            right = mid - 1
        elif mid <= n-2 and nums[mid+1] > nums[mid]:
            left = mid + 1
    return -1
```

time complexity: $O(\log n)$

space complexity: $O(1)$

67. Trapping Rain Water 接雨水

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

03/18/22, 03/23

我们可以用动态规划的方法。对于下标i, 下雨后能到达的最大高度等于下标i两边的最大高度的最小值, 下标i处能接的雨水量等于下标i处的水能到达的最大高度减去 $\text{height}[i]$ 。我们可以创建两个列表leftmax和rightmax, 分别表示下标i的左边和右边位置中能到达的最大高度。我们分别正向和反向遍历数组就能得到leftmax和rightmax。

```
def trap(height):
    n = len(height)
    leftmax = [0] * n
    leftmax[0] = height[0]
    rightmax = [0] * n
```

```

rightmax[n-1] = height[n-1]
for i in range(1, n):
    leftmax[i] = max(leftmax[i-1], height[i])
for i in range(n-2, -1, -1):
    rightmax[i] = max(rightmax[i+1], height[i])
res = [min(leftmax[i], rightmax[i]) - height[i] for i in range(n)]
return sum(res)

```

time complexity: O(n)
space complexity: O(n)

68. Sort an Array 排序数组

<https://leetcode-cn.com/problems/sort-an-array/>

03/24/22, 04/05

我们可以用快速排序的方法来做。快速排序的方法简单来说就是分解，解决，合并。第一步分解，将数组nums[l, r]分解为两个可能为空的子数组nums[l, pivot-1]和nums[pivot+1, r]，使得第一个子数组中所有元素均小于nums[pivot]，第二个子数组中所有元素均大于nums[pivot]，而计算下标pivot是划分过程的一部分。第二步解决，通过递归调用快速排序，对子数组进行排序。因为子数组都是原址排序，故不需要合并。初始化的时候我们可以根据需求定义四个子空间[l, i], [i+1, j], [j+1, r-1], [r]，依次为小于nums[r]的空间，大于等于nums[r]的空间，待排序的空间以及基准元素nums[r]。我们构建一个维护过程，即给定一个任意序列，通过交换顺序来维护快速排序的性质，将序列的元素放入四个空间内。当j=r时，此时序列已经有序了。

```
import random
```

```

def quicksort(self, nums, l, r):
    if l < r:
        p = self.partition(nums, l, r)
        self.quicksort(nums, l, p)
        self.quicksort(nums, p+1, r)

def partition(self, nums, l, r):
    pivot = random.randint(l, r)
    nums[pivot], nums[r] = nums[r], nums[pivot]
    i = l - 1
    for j in range(l, r):
        if nums[j] <= nums[r]:
            i += 1
            nums[i], nums[j] = nums[j], nums[i]
    nums[i+1], nums[r] = nums[r], nums[i+1]
    return i

```

```
def sortArray(self, nums):
    n = len(nums)
    self.quicksort(nums, 0, n-1)
    return nums
```

time complexity: $O(n \log n)$
space complexity: $O(n)$

69. Valid Parentheses 有效的括号

<https://leetcode-cn.com/problems/valid-parentheses/>

03/25/22

我们可以用栈来解决。遍历给定的字符串s, 当我们遇到一个左括号时, 我们会期望在后续的遍历中, 有一个相同类型的右括号将其闭合。由于后遇到的左括号要先闭合, 因此我们可以将这个左括号放入栈顶。

```
def isValid(s):
    d = {"(": ")", "(": ")", "{": "}", "[": "]"
    stack = []
    for c in s:
        if c in d:
            if not stack or stack[-1] != d[c]:
                return False
            else:
                stack.pop()
        else:
            stack.append(c)

    return not stack
```

time complexity: $O(n)$
space complexity: $O(n)$

70. Spiral Matrix 螺旋矩阵

<https://leetcode-cn.com/problems/spiral-matrix/>

03/26/22

我们可以按层模拟。将矩阵看成若干层，首先输出最外层的元素，然后输出次外层的元素，直到输出最内层的元素。

```
def spiralOrder(matrix):
    r, c = len(matrix), len(matrix[0])
    left, right, top, bottom = 0, c-1, 0, r-1
    res = []
    while left <= right and top <= bottom:
        for i in range(left, right+1):
            res.append(matrix[top][i])
        for i in range(top+1, bottom+1):
            res.append(matrix[i][right])
        if left < right and top < bottom:
            for i in range(right-1, left, -1):
                res.append(matrix[bottom][i])
            for i in range(bottom, top, -1):
                res.append(matrix[i][left])
        left, right, top, bottom = left+1, right-1, top+1, bottom-1
    return res
```

time complexity: $O(mn)$

space complexity: $O(mn)$

71. 部分排序

<https://leetcode-cn.com/problems/sub-sort-lcci/>

03/26/22

我们可以做两次遍历。先从左到右遍历，找出左边界。先找出从左到右的递增数组，再看右边是否有更小的数，有的话左边界左移。再从右到左遍历，找出右边界。先找出从右到左的递减数组，再看左边是否有更大的数，有的话右边界右移。

```
def subSort(array):
    n = len(array)
    if n <= 1:
        return [-1, -1]

    i = 0
    while i < n-1 and array[i+1] >= array[i]:
        i += 1
    for j in range(i+1, n):
        while i >= 0 and array[j] < array[i]:
            i -= 1
```

```

        left = i + 1
    i = n - 1
    while i > 0 and array[i-1] <= array[i]:
        i -= 1
    for j in range(i-1, -1, -1):
        while i <= n-1 and array[j] > array[i]:
            i += 1
    right = i - 1

    if left < right:
        return [left, right]
    else:
        return [-1, -1]

```

72. Reverse Words in a String 颠倒字符串中的单词

<https://leetcode-cn.com/problems/reverse-words-in-a-string/>

73. Next Permutation 下一个排列

<https://leetcode-cn.com/problems/next-permutation/>

74. Restore Ip Addresses 复原 IP 地址

<https://leetcode-cn.com/problems/restore-ip-addresses/>

75. First Missing Positive 缺失的第一个正数

<https://leetcode-cn.com/problems/first-missing-positive/>

76. Generate Parentheses 括号生成

<https://leetcode-cn.com/problems/generate-parentheses/>

77. Sum Root to Leaf Numbers 求根节点到叶节点数字之和

<https://leetcode-cn.com/problems/sum-root-to-leaf-numbers/>

78. Min Stack 最小栈

<https://leetcode-cn.com/problems/min-stack/>

79. Validate Binary Search Tree 验证二叉搜索树

<https://leetcode-cn.com/problems/validate-binary-search-tree/>

80. Find the common elements from two lists of sorted integers, of lengths m and n, where duplicate numbers are possible.

- What is a naive way to search? And what is its time and space complexity?

```
def duplicate(nums1, nums2):  
    set1 = nums1  
    set2 = nums2  
    if len(set1) < len(set2):  
        return [num for num in set1 if num in set2]  
    else:  
        return [num for num in set2 if num in set1]
```

time complexity: $O(m+n)$

space complexity: $O(m+n)$

- How do you use the sorted property? What is its time and space complexity?

81. Coding - python

- Given a csv file output the observations that is > 5 std. 可以用panda read_csv
- calculate rolling mean (no package). How to optimize in terms of run time and space?

```
import pandas as pd

df = pd.read_csv("test.csv")
df = df.loc[df["obs"] > 5*df["obs"].std()]

def rolling_mean(A):
    cum_sum = 0
    moving_means = []
    for i in range(len(A)):
        cum_sum += A[i]
        moving_means.append(cum_sum/(i+1))
    return moving_means
```

82. 数列只含0和1, 并且所有0都在1之前, 比如:[0, 0, 0, 1, 1, 1, 1]。求第一个1的index。(数列可能全部含0或全部含1)

```
def find_index(A):
    if 1 not in A:
        return None
    left = 0
    right = len(A) - 1
    while left < right:
        mid = int((left + (right-left) / 2))
        if A[mid] == 0:
            left = mid + 1
        else:
            right = mid
    return left
```

83. 一个Python的算法, two sum 然后说出如何优化

```
def two_sum(nums, target):
    nums_dict = {}
    for i in range(len(nums)):
        if nums[i] not in nums_dict:
            nums_dict[target-nums[i]] = i
        else:
            return [i, nums_dict[nums[i]]]
```

time: O(n)

space: $O(n)$

84. find out prime number 各种follow up 和优化 -这种类型的题目很久没有brush了 所以基本靠自己想法写的, 小哥哥各种给提示, 最终感觉自己还是没写好lol

```
def check_prime(num):
    if num > 1:
        for i in range(2, int(num / 2) + 1):
            if num % i == 0:
                return False
        else:
            return True
    else:
        return False
```

85. python,read一个text file,split每一行line,提取line里面的关键字,然后以此关键字作为key,将后面的所有值相加作为value,再print出来

```
from collections import defaultdict

with open("data.txt") as f:
    content = f.readlines()
key_values = defaultdict(int)
for line in content:
    key = line.split(",")[0].strip()
    value = int(line.split(",")[1].strip())
    key_values[key] += value
```

86. 有一个text文本:

每一行记录A家庭有几个孩子, 然后每个孩子的年龄, 例如:

A, 3, 1|2|3

B, 2, 4|6

C, 0,

D, 1, 12

要求Parse出一个规范dataframe。无所谓语言, 重点考察parse逻辑。

```
import pandas as pd

with open("data.txt") as f:
    content = f.readlines()
houses = []
kids = []
```

```

ages = []
for line in content:
    house = line.split(",")[0].strip()
    houses.append(house)
    kid = int(line.split(",")[1].strip())
    kids.append(kid)
    age = line.split(",")[2].strip()
    age = age.split("|")
    ages.append(age)
df = pd.DataFrame({"houses": houses, "kids": kids, "ages": ages})

```

87. coding: 和word frequency有关, 常见题, 然后后面有两小问, 一个和re有关一个 hashmap 性质有关. calculate the frequency of each word in a text file.

words.txt

```

from collections import Counter

with open("words.txt") as f:
    content = f.readlines()
dict = Counter()
for line in content:
    words = line.strip().split()
    dict.update(words)

```

88. coding (python) :

(1) 写一个rotation函数, argument是一个string和一个非负int。例子:

rotation('facebook', 0): return 'facebook'

rotation('facebook', 2): return 'cebookfa'

rotation('facebook', 19): return 'ebookfac' (和19-8-8=3的情况一样)

```

def rotation(s, num):
    length = len(s)
    start_pos = num % length
    return s[start_pos:] + s[:start_pos]

```

(2) 给了一个numpy array, 包含了很多vector, 每个vector是长度为5的概率分布(加起来等于1)。计算每行的entropy(注意去掉概率为0的情况)。

$$Entropy = - \sum_i p_i \log p_i$$

```
results = []
```

```
for vec in vecs:
```

```
    entropy = np.sum([prob*np.log(prob) for prob in vec if prob > 0])
```

```
    results.append(entropy)
```

89. Inner join two tables using SQL, Pandas, Python

SQL:

```
SELECT *  
FROM table1  
JOIN table2  
ON table1.col1 = table2.col1
```

Pandas:

```
A.merge(B, left_on="col1", right_on="col1", how="inner")
```

Python: Assume we know the kth column is the common column in both tables

```
with open("table1.txt") as f1:  
    tbl1 = f1.readlines()  
with open("table2.txt") as f2:  
    tbl2 = f2.readlines()  
tbl1 = tbl1[1:]  
tbl2 = tbl2[2:]  
result = []  
for i in range(len(tbl1)):  
    col_tbl1 = tbl1[i].split().strip()[k]  
    for j in range(len(tbl2)):  
        col_tbl2 = tbl2[j].split().strip()[k]  
        if col_tbl1 == col_tbl2:  
            result.append(tbl1[i] + " " + tbl2[j])
```