Day61: 模拟行走机器人

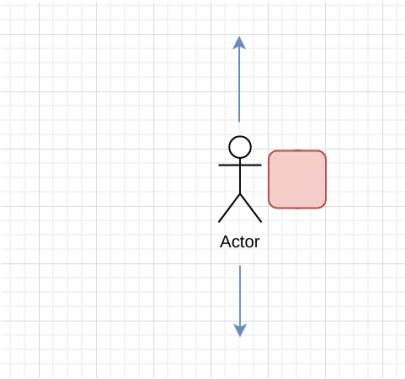
题目

机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。该机器人可以接收以下三种类型的命令：

-2：向左转 90 度
-1：向右转 90 度
1 <= x <= 9：向前移动 x 个单位长度
在网格上有一些格子被视为障碍物。

第 i 个障碍物位于网格点 (`obstacles[i][0]` , `obstacles[i][1]`)

机器人无法走到障碍物上，它将会停留在障碍物的前一个网格方块上，但仍然可以继续该路线的其余部分。



返回从原点到机器人所有经过的路径点（坐标为整数）的**最大欧式距离的平方**。

例子

示例 1：

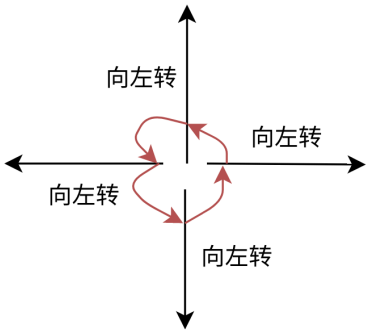
输入: commands = [4,-1,3], obstacles = []
输出: 25
解释: 机器人将会到达 (3, 4)
示例 2：

输入: commands = [4,-1,4,-2,4], obstacles = 2,4
输出: 65
解释: 机器人在左转走到 (1, 8) 之前将被困在 (1, 4) 处

提示：

0 <= commands.length <= 10000
0 <= obstacles.length <= 10000
-30000 <= obstacle[i][0] <= 30000
-30000 <= obstacle[i][1] <= 30000
答案保证小于 2 ^ 31

分析



大家注意观察上图，假定起始箭头指向正上方，向左旋转90度指向正左方，再旋转90度指向正下方，再旋转90度指向正右方。

再旋转90度，又指向正上方，循环往复，周期为 4 .

这道题目有一些很好的技巧，可能搞自动驾驶的朋友看到这道题，可能一下都能想到。

设定以上四个方向的单位向量分别为：`[0,1]`，`[-1,0]`，`[0,-1]`，`[1,0]`

根据题意，若指令等于`-2`，表示向左旋转90，即变为正左方，对应`[-1,0]`，再向左旋转90对应`[0,-1]`，再旋转90对应`[1,0]`

上面这些操作，可以总结为如下代码：

```
d = 0
vectors = [[0,1], [-1,0], [0,-1], [1,0]]
if command == -2: # 向左旋转
    d = (d + 1) % 4
    vector = vectors[d]
```

同理，若收到指令`-1`，表示向又旋转90，同理可得：

```
d = 0
vectors = [[0,1], [-1,0], [0,-1], [1,0]]
if command == -1: # 向右旋转
    d = (d - 1) % 4
    vector = vectors[d]
```

收到其他指令，只需沿着`vector`方向一步一步前行即可。若遇到障碍物，停下来，直到收到下一条指令。

代码

基于上面的分析，写出代码就不会太难：

```
def robotSim(commands,obstacles):
    # 预处理
    obstacles = set(map(tuple,obstacles))
    d = 0
    vectors = [[0,1], [-1,0], [0,-1], [1,0]]
    x,y,maxd = 0,0,0
    for command in commands:
        if command == -2: # 向左旋转
            d = (d + 1) % 4
        elif command == -1: # 向右旋转
            d = (d - 1) % 4
        else:
            vector = vectors[d]
            # 需沿着 `vector` 方向一步一步前行即可。若遇到障碍物，停下来，直到收到下一条指令。
            while command > 0:
                if (x + vector[0], y + vector[1]) not in obstacles: # 无障碍
                    x, y = x + vector[0], y + vector[1]
                command -= 1
            # 贪心
            maxd = max(maxd, x**2+y**2)
     return maxd
```

leetcode上验证一遍，以上代码未发现问题。

Day62：两地调度

题目

公司计划面试 `2N` 人。第 `i` 人飞往 A 市的费用为 `costs[i][0]`，飞往 B 市的费用为 `costs[i][1]`

返回将每个人都飞到某座城市的最低费用，要求每个城市都有 `N` 人抵达。

示例

```
输入：[[10,20],[30,200],[400,50],[30,20]]
输出：110
解释：
第一个人去 A 市，费用为 10。
第二个人去 A 市，费用为 30。
第三个人去 B 市，费用为 50。
第四个人去 B 市，费用为 20。

最低总费用为 10 + 30 + 50 + 20 = 110，每个城市都有一半的人在面试。
```

来源：力扣（LeetCode）

补充下面代码：

```python
class Solution(object):
    def twoCitySchedCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
```

分析

如何从零开始分析这道题？如何构思？

最基本的思路：选择`[ACost,BCost]`成本较小的飞往对应城市，哪个城市人数先到`N`，剩余的人都分配另一个城市。这个思路有问题吗？考虑下面的例子：

`[[10,20],[30,200],[50,400],[30,20]]`

根据上述思路：

第一个应该飞往A，成本为10

第二个飞往A，成本为30

A城市率先到`N`人，所以剩余的都飞B城市 ，成本分别为400,20

总成本：10 + 30 + 400 + 20 = 460

很明显这不是最优解，因为选择400成本已有问题。

这种维度的贪心问题出在哪里？只根据元素`[ACost,BCost]`的两者大小选择是有问题的，上面的例子交换第二、三个元素的顺序：

`[[10,20],[50,400],[30,200],[30,20]]`

第一个应该飞往A，成本为10

第二个飞往A，成本为50

A城市率先到`N`人，所以剩余的都飞B城市 ，成本分别为200,20

总成本：10 + 50 + 200 + 20 = 280

同样思路，仅仅交换元素顺利，但整个问题没有变化，得到结果必须相同的才是正确的思路。

这种变化也让我们意识到，`[ACost,BCost]`的差值是不受列表内元素出现顺序影响的，所以取得差值绝对值：`abs(ACost-BCost)`，相差越大的越是应该要优先选择成本更小的。所以，对成本差的绝对值降序排序，按照此顺序，贪心的选择成本较小者飞往对应城市，哪个城市率先到`N`人，剩余的就都安排到另一个城市。

综上所述：维度`abs(ACost-BCost)`的贪心 ，确保取得最优解。

代码

根据上述思路，翻译为下面的代码：

```
import numpy as np
class Solution(object):
    def twoCitySchedCost(self,costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        diff = [abs(a - b) for a, b in costs]
        indices = np.argsort(diff)[::-1] # 按照diff降序
        ACount, N, minc = 0, len(costs) // 2, 0
        for i, indice in enumerate(indices):
            c = costs[indice]
            if ACount >= N: # 都选B城市
                minc += c[1]
            elif i - ACount >= N: # 都选A城市
                minc += c[0]

            elif c[0] < c[1]:
                ACount, minc = ACount+1, minc+c[0]
            else:
                ACount, minc = ACount, minc + c[1]
        return minc
```

进一步分析

上面的思路根据维度`abs(ACost-BCost)`贪心，实际上根据`ACost-BCost`贪心更加简单。根据`ACost-BCost`升序排序，排序后的序列前半部分飞往A城，后半部分飞往B城，妙！

代码：

```
class Solution(object):
    def twoCitySchedCost(self,costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        costs.sort(key=lambda cost: cost[0]- cost[1]) #就地排序
        N, sumc = len(costs) // 2, 0
        for i in range(N):
            sumc += costs[i][0] + costs[i+N][1]
        return sumc
```

相比于第一种贪心解法，代码更加简洁，空间复杂度降低为O(1)

Day63：换酒问题

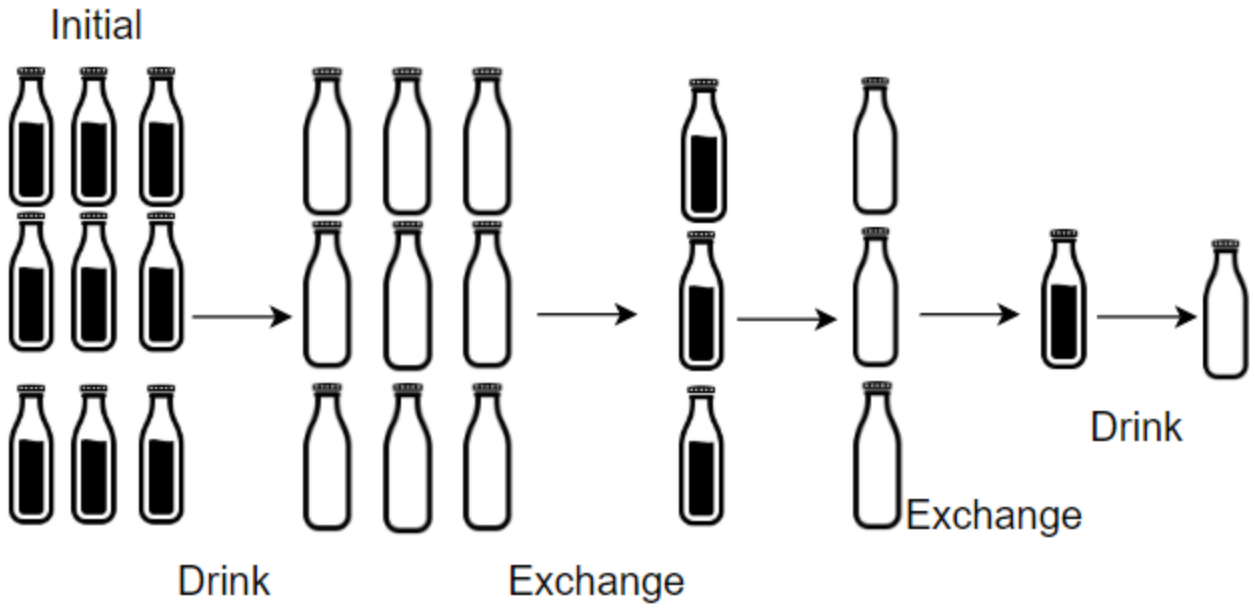
题目

小区便利店正在促销，用 numExchange 个空酒瓶可以兑换一瓶新酒。你购入了 numBottles 瓶酒。

如果喝掉了酒瓶中的酒，那么酒瓶就会变成空的。

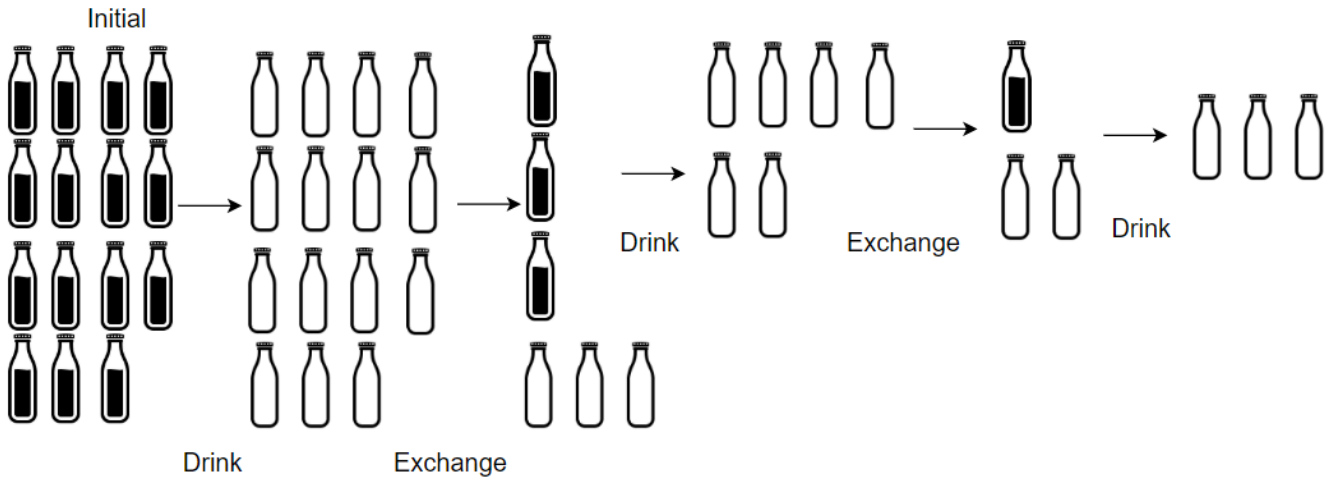
请你计算**最多**能喝到多少瓶酒。

示例 1



输入：numBottles = 9, numExchange = 3
输出：13
解释：你可以用 3 个空酒瓶兑换 1 瓶酒。
所以最多能喝到 9 + 3 + 1 = 13 瓶酒。

示例 2



输入：numBottles = 15, numExchange = 4
输出：19
解释：你可以用 4 个空酒瓶兑换 1 瓶酒。
所以最多能喝到 15 + 3 + 1 = 19 瓶酒。

分析

这道题贪心的维度非常明显，直接暴露在题目表面，即当前喝完所有饮料后变为空瓶加上已有空瓶后，最大限度的、贪心的、去兑换饮料，依次类推，直到手上的空瓶不足以兑换出一瓶饮料止。

代码

根据上述分析，兑现为代码如下：

```python
class Solution(object):
    def numWaterBottles(self, numBottles, numExchange):
        """
        :type numBottles: int
        :type numExchange: int
        :rtype: int
        """
        sumb = numBottles
        empty = numBottles
        while empty // numExchange:
            bottle = empty // numExchange # 兑酒数
            empty = bottle + empty % numExchange # 空瓶子数
            sumb += bottle

        return sumb
```

Day64：买卖股票的最佳时机

题目

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所能获取的最大利润。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）。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii

例子

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 。

示例 2:

输入: [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 。
注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:

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

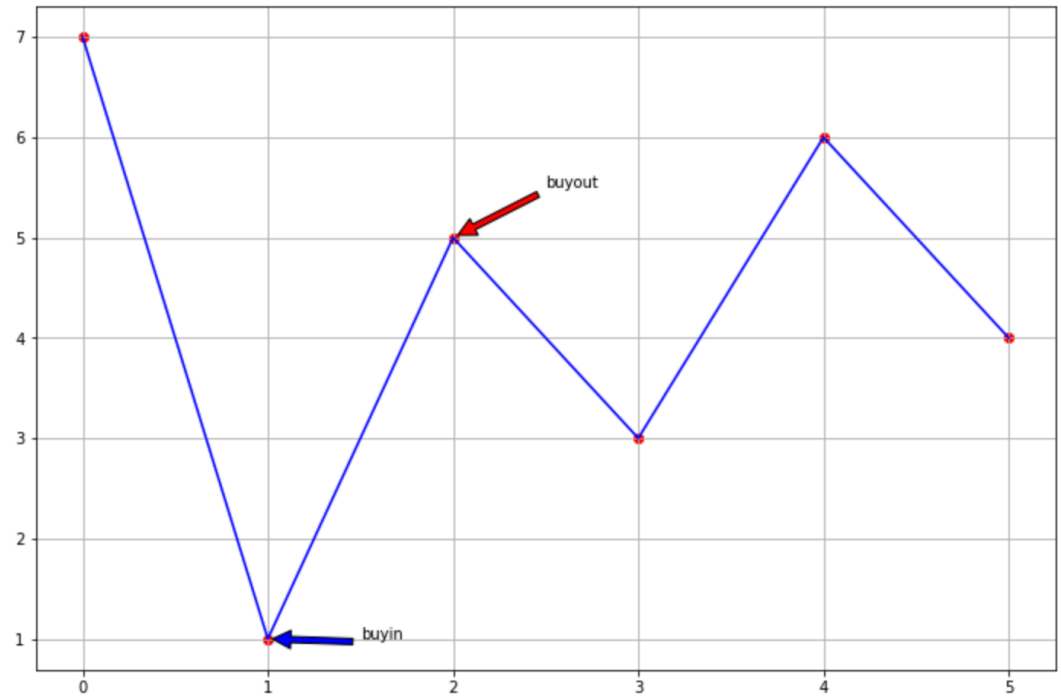
分析

以交易序列 prices = [7,1,5,3,6,4] 为例，分析交易获得最大收益。使用`matplotlib`绘制交易图：

```
import matplotlib.pyplot as plt

prices = [7,1,5,3,6,4]
index = range(len(prices))
fig = plt.figure(figsize=(12,8))
plt.scatter(index,prices,c='r')
plt.plot(index,prices,c='b')
plt.annotate('buyin', xy=(1, 1),
             xytext=(1.5, 1),\
            xycoords='data',\
            arrowprops=dict(facecolor='blue', shrink=0.05))
plt.annotate('buyout', xy=(2, 5),
             xytext=(2.5, 5.5),\
            xycoords='data',\
            arrowprops=dict(facecolor='red', shrink=0.05))

plt.grid()
plt.show()
```



观察图形，首先要买入股票，买入点当然越低越好，所以一直寻找最低点，然后买入，寻找方法：

```
buyin, buyout = 10**4+1,0
i,c,sump = 0,len(prices),0

while i < c and prices[i] < buyin:
    buyin = prices[i]
    i += 1
```

退出`while`循环后，找到最佳买入点`buyin`

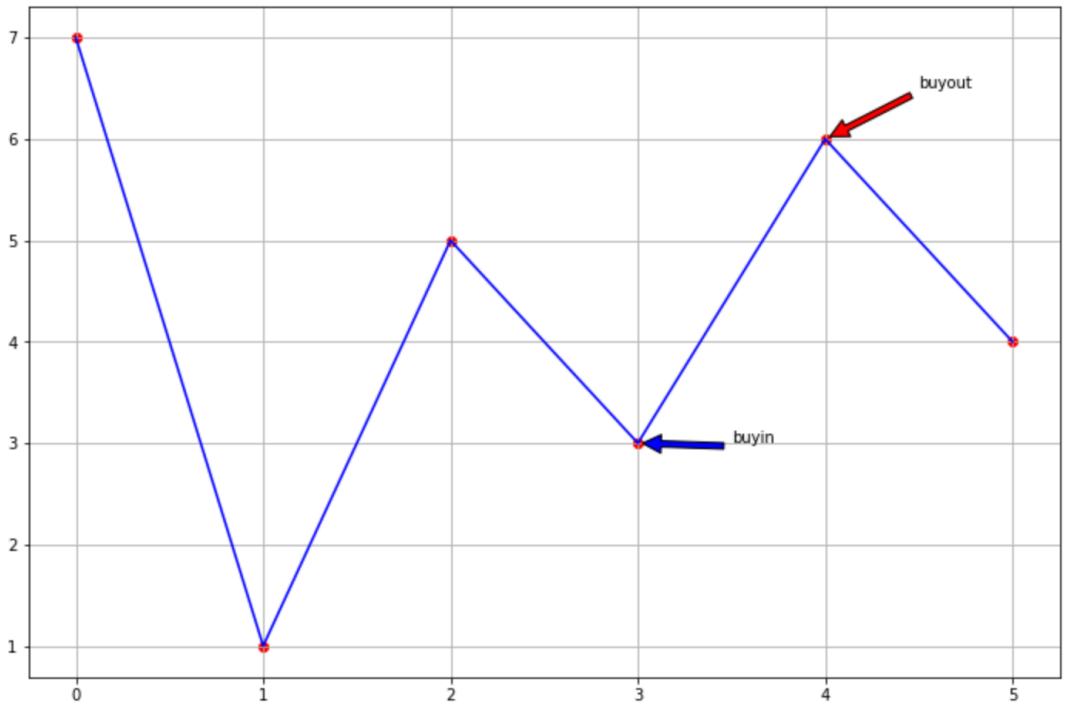
买入后，最佳卖出点当然是越高越好，一直寻找，知道找到局部最高点，其方法：

```
while i < c and buyout < prices[i]:
    buyout = prices[i]
    i += 1
```

这样得到截止最高点时的最大收益：

```
sump += max(0,buyout - buyin)
```

同样原理，寻找下一次交易的波谷和波峰，累加到`sump`上。此例子寻找到另一对波谷和波峰：



代码

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if not prices:
            return 0
        buyin, buyout = 10**4+1,0
        i,c,sump = 0,len(prices),0
        while i < c:
            while i < c and prices[i] < buyin:
                buyin = prices[i]
                i += 1
            while i < c and buyout < prices[i]:
                buyout = prices[i]
                i += 1
            sump += max(0,buyout - buyin)
            buyin, buyout = 10**4+1,0
        return sump
```

Day65：柠檬水找零

题目

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

示例 1：

输入：[5,5,5,10,20]
输出：true
解释：
前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
由于所有客户都得到了正确的找零，所以我们输出 true。

示例 2：

输入：[5,5,10]
输出：true

示例 3：

输入：[10,10]
输出：false

示例 4：

输入：[5,5,10,10,20]
输出：false

解释：
前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
由于不是每位顾客都得到了正确的找零，所以答案是 false。

提示：

0 <= bills.length <= 10000
bills[i] 不是 5 就是 10 或是 20

代码

```python
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        five = 0
        ten = 0
        for b in bills:
            if b == 5:
                five += 1
            elif b == 10:
                # 收到一张10，找出一张5
                five -= 1
                ten += 1
            else:
                # 收到20，找出15；
                # 如果有10，优先将10找出
                if ten > 0:
                    ten -= 1
                    five -= 1
                else:
                    five -= 3

            if ten < 0 or five < 0:
                return False

        return True
```

Day66：跳跃游戏

题目

给定一个非负整数数组，你最初位于数组的第一个位置。

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位置。

例子

示例 1:

输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

来源：力扣（LeetCode）


```
class Solution(object):
def canJump(self, nums):
"""
:type nums: List[int]
:rtype: bool
"""
```

题目分析

首先明确数组中每个元素的意义：代表你在该位置可以跳跃的**最大**长度，大家一定注意是能够跳

跃的**最大**长度，而不是每次跳跃恰好都是这个长度值。

考虑你已跳跃到index等于i的位置，那么下一跳你能到达**可能的**最远位置等于：$f(i) = i + nums[i]$，表明你可以到达0 $f(i)$的任何一个位置！

如果 $f(i) \geq len(nums) - 1$，表明能够达到最后一个位置，此处跳过这个位置也代表能到达。

如果 $f(i) < len(nums) - 1$，表明还未到达最后一个位置，此处大家要注意，此时的最远距离**一定**等于 f(i)？

这是不一定的，需要比较 f(i) 和 前面0~i-1次跳跃的最远距离 $maxd$，并选择较大者，即 $maxd = max(f(i), maxd)$.

代码

```
class Solution:
    def canJump(self, nums):
        n, maxd = len(nums), 0
        for i in range(n):
            if i > maxd:
                return False
            fi = i + nums[i]
            maxd = max(maxd, fi)
            if maxd >= n - 1:
                return True
        return False
```

Day67：最大跳跃次数2

题目

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例

示例:

输入: [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
说明:

假设你总是可以到达数组的最后一个位置。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/jump-game-ii
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。

分析

基于上一题的基本思路，$maxd$标记着跳跃的最远位置，$i + nums[i]$是从index等于i处能跳跃的最远距离。

此题要求跳跃到终点的最少跳跃次数。这是一道不太容易想出来的题目，虽然最后解题的代码只有几行。

初始状态：第一跳，最大幅度的跳，贪心的跳，即等于：$nums[i]$，并赋值给 maxd ，此时 i = 0;

然后遍历向前一步，即 i = 1，比较maxd 和 i+nums[i] 的较大者：

- 若 i 在 0~nums[0]之间能跳跃到的最远距离为 nums[0]，则第二跳的起始位置为 nums[j]
- 若 i 在 0~nums[0]之间能跳跃到的最远距离为 j + nums[j]，其中0 < j < nums[0] 则第二跳的起始位置为 j，则第一跳应该跳到 j 处，而不是nums[0] 处，同时更新 maxd = j + nums[j].

想清楚以上过程后，再求解此题仍然还有一定难度，因为直接将上述过程描述为代码，仍然不容易。

这道题贪心思想发挥的淋漓尽致，我们使用at变量标记第一次贪心跳跃的位置，当 i 遍历到at时

就认为是一次跳跃，本题我们完全并不用关心准确跳跃到哪一个最佳位置。

为什么说当 i 遍历到at时就认为是一次跳跃呢？

如果你能思考清楚上面说到的两种情况：

- 若 i 在 0~nums[0]之间能跳跃到的最远距离为 nums[0]，则第二跳的起始位置为 nums[j]
- 若 i 在 0~nums[0]之间能跳跃到的最远距离为 j + nums[j]，其中0 < j < nums[0] 则第二跳的起始位置为 j，则第一跳应该跳到 j 处，而不是nums[0] 处，同时更新 maxd = j + nums[j].

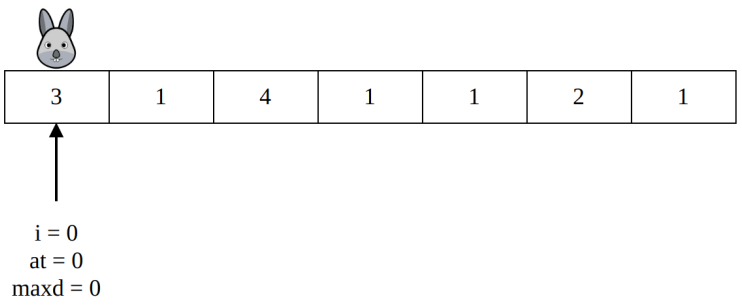
你便能找到证据，因为第一次跳跃的最佳位置 j 一定位于 0~nums[0]处。

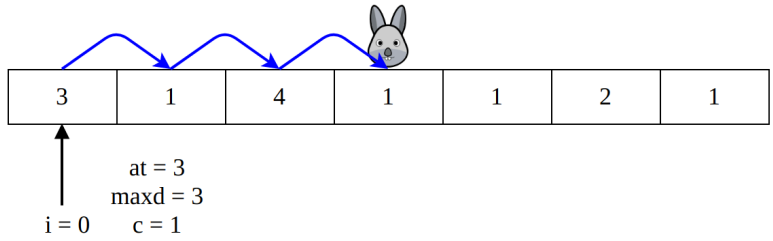
代码

```
class Solution(object):
    def jump(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        n,at,maxd = len(nums),0,0
        c = 0
        for i in range(n-1):
            maxd = max(maxd, i+nums[i]) # 贪心的跳跃

            if i == at: #at标记着上一次的位置，i等于at后，表明找到一个跳跃次数
                at = maxd
                c += 1
        return c
```

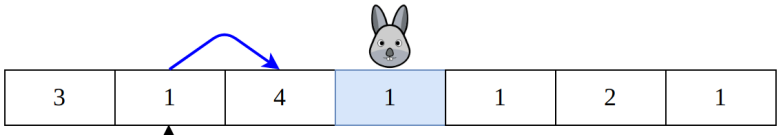
实例





i = 0

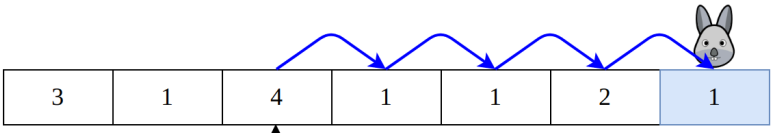
at = 3
maxd = 3
c = 1



i = 1

maxd = max(3, 1 + 1)

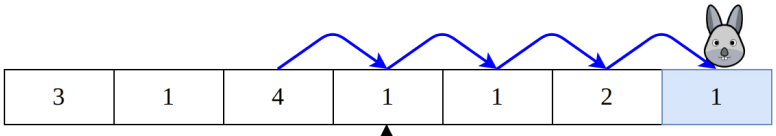
at = 3
c = 1



i = 2

maxd = max(3, 2 + 4)

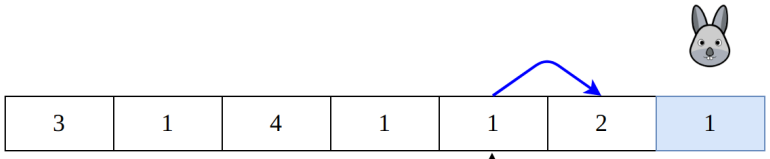
at = 6
c = 1



i = 3

max(6, 3 + 1)

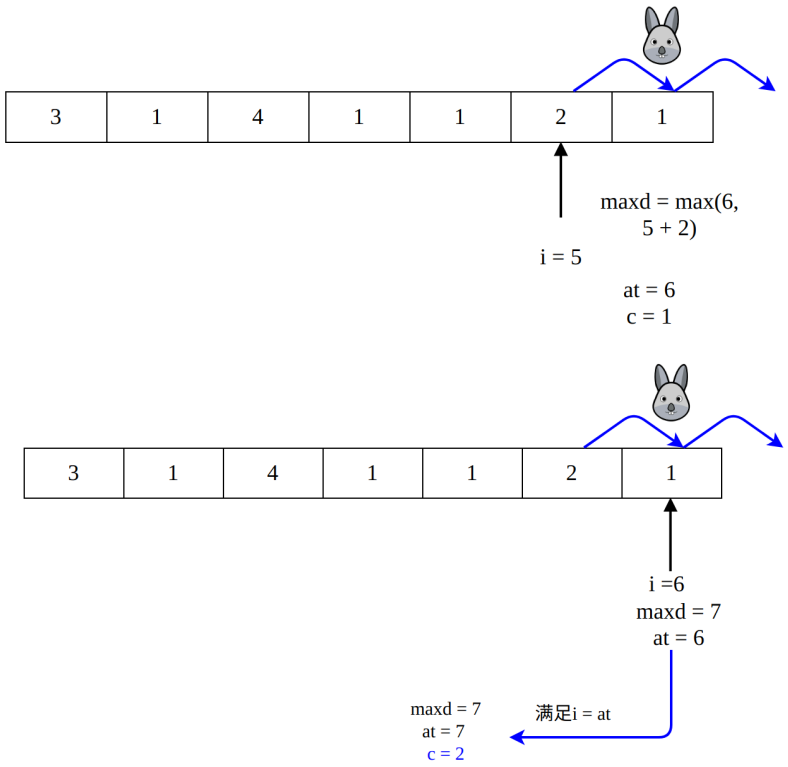
at = 6
c = 1



i = 4

max(6, 4 + 1)

at = 6
c = 1



Day68：分发饼干

题目

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] >= 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。你的目标是尽可能满足越多数量的孩子，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 <= g.length <= 3 * 10^4
0 <= s.length <= 3 * 10^4
1 <= g[i], s[j] <= 2^31 - 1

分析

为了能尽可能满足更多的孩子，如果有能满足孩子胃口的饼干，就用尽量小的饼干去满足他，大一些的就留给胃口更大的孩子。

代码

```python
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        # 排序
        g.sort()
        s.sort()

        res = 0
        # i是孩子们的索引，j是饼干的索引
        i, j = 0, 0
        m, n = len(g), len(s)

        for j in range(n):
            # 遍历所有的饼干
            if i == m:
                break
            if g[i] <= s[j]:
                # 如果当前孩子的胃口不大于饼干尺寸，将饼干分给这个孩子
                res += 1
                # 再来看下一个孩子
                i += 1
            # 而如果当前孩子的胃口大于饼干尺寸，那么后面孩子的胃口也是比此饼干尺寸大
            # 前面的孩子都有了自己的饼干，所以这个饼干就分不出去。

        return res
```

Day69：将数组拆分成斐波那契序列

题目

给定一个数字字符串 S，比如 S = "123456579"，我们可以将它分成斐波那契式的序列 [123, 456, 579]。

形式上，斐波那契式序列是一个非负整数列表 F，且满足：

0 <= F[i] <= 2^31 - 1，（也就是说，每个整数都符合 32 位有符号整数类型）；
F.length >= 3；
对于所有的0 <= i < F.length - 2，都有 F[i] + F[i+1] = F[i+2] 成立。
另外，请注意，将字符串拆分成小块时，每个块的数字一定不要以零开头，除非这个块是数字 0 本身。

返回从 S 拆分出来的任意一组斐波那契式的序列块，如果不能拆分则返回 []。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/split-array-into-fibonacci-sequence

示例

示例 1：

```
输入："123456579"
输出：[123,456,579]
```

示例 2：

```
输入：  "11235813"
输出：  [1,1,2,3,5,8,13]
```

示例 3：

```
输入：  "112358130"
输出：  []
解释：  这项任务无法完成。
```

分析

这道题想了很久，也调试了一段时间，比较容易出错。没有太好的思路，无非就是贪心的搜寻，主要两点：

1. 刚开始的两个数，决定后续的所有序列
2. 需要穷举初始状态时所有可能的前两个数，一旦找到这样的序列返回即可。

终止的状态：

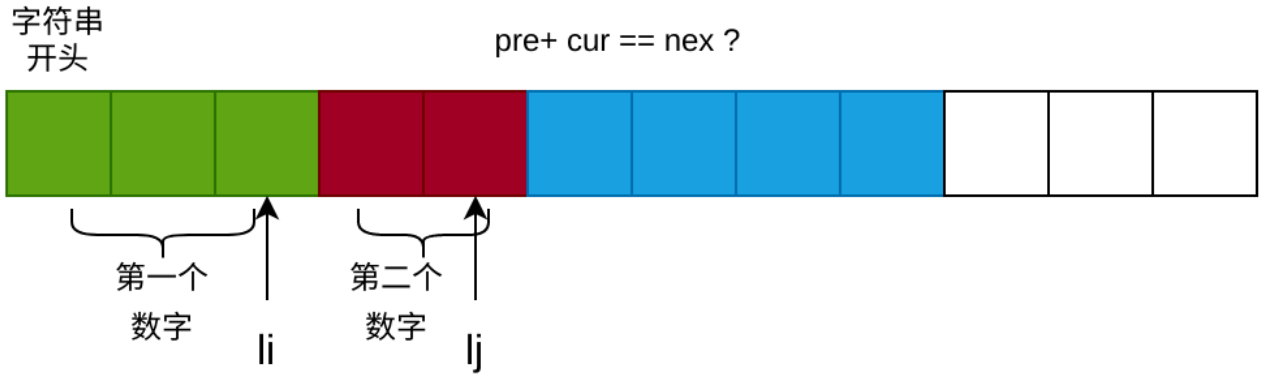
列表内元素对应字符串长度和恰好等于原字符串长度，且至少含有三个元素。

注意：

字符串以前导0开始

结合图形，给出一个基本的求解思路：

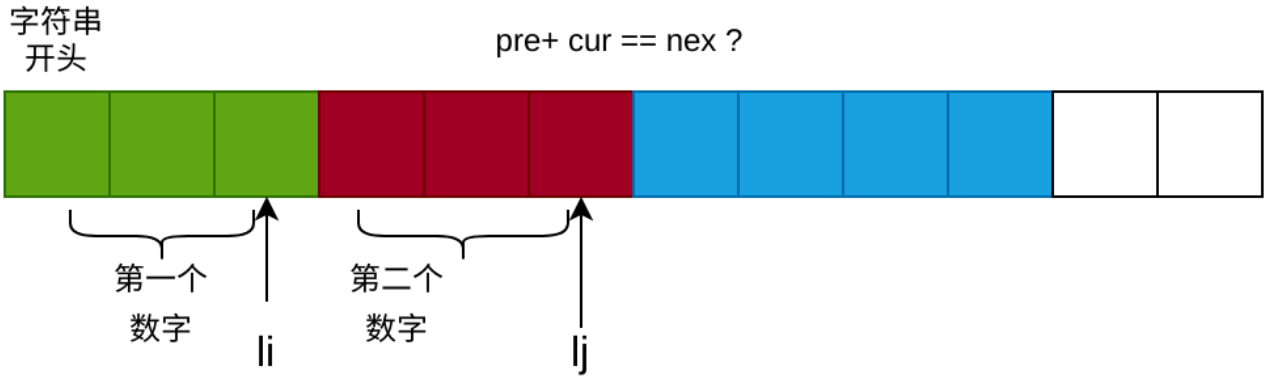
中间某个状态1：



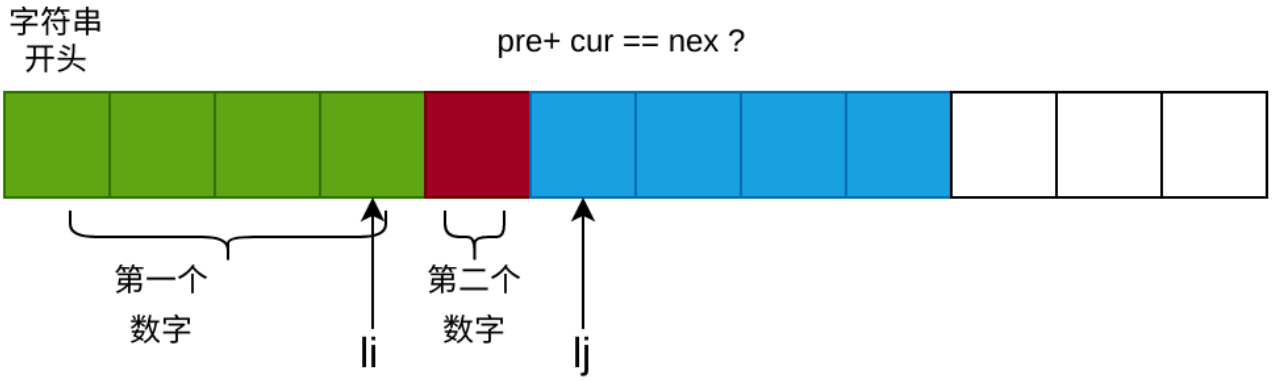
中间某个状态2：

nexi = lj

nexj = lj +len(第三个数)



中间某个状态3：


字符串
开头
pre+ cur == nex ?
第一个
数字
第二个
数字
li
lj

代码

```
class Solution(object):
    ### 贪心+回溯
    def splitIntoFibonacci(self, S):
        sl = len(S)
        li,lj = 1, 1
        while li < sl-1 or lj < sl-1:
            ### 预处理部分：
            ### 前导0的处理
            if S[0]=='0' and lj >= sl-1:
                break
            pre,li = ("0",1) if S[0] == "0" else (S[0:li],li)
            cur = "0" if S[li] == "0" else S[li:li+lj]

            ### 确定刚开始的两个数，并不断回溯
            r = [pre,cur]
            nex = str(int(pre) + int(cur))
            nexi, nexj = li+lj,li+lj+len(nex)
            prefix, three = li+lj, 0

            ### 迭代部分
            while prefix + three < sl and S[nexi:nexj] == str(nex):
                if int(nex) > 2**31 - 1:
                    break
                r.append(nex)
                three += len(str(nex))
                pre,cur = cur,nex
                nex = int(pre) + int(cur)
                nexi,nexj= nexj,nexj+len(str(nex))

            lj += 1

            ### 判断状态是否终止
            if three > 0 and prefix + three == sl:
                return r

            ### 回溯部分
            if lj >= sl:
                li, lj = li + 1, 1

        return []
```

Day 70：最后一块石头的重量

题目

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出两块 最重的 石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：

如果 x == y，那么两块石头都会被完全粉碎；
如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 0。

示例

输入：[2,7,4,1,8,1]
输出：1
解释：
先选出 7 和 8，得到 1，所以数组转换为 [2,4,1,1,1]，
再选出 2 和 4，得到 2，所以数组转换为 [2,1,1,1]，
接着是 2 和 1，得到 1，所以数组转换为 [1,1,1]，
最后选出 1 和 1，得到 0，最终数组转换为 [1]，这就是最后剩下那块石头的重量。

分析

直接根据题目的要求，选出当前最大两块石头，消除一下，显然这个操作与后续状态无关，因此贪心求解即可。

使用堆数据结构，因为Python中默认的`heapq`为小根堆，简单转换一下。时间复杂度为O(n)，空间复杂度为O(n)

代码

```
from heapq import *

class Solution(object):
    def lastStoneWeight(self, stones):
        """
        :type stones: List[int]
        :rtype: int
        """
        stones = [stone*-1 for stone in stones]
        heapify(stones)
        while len(stones) > 1:
            fst,sec = heappop(stones),heappop(stones)
            if fst - sec < 0:
                heappush(stones,fst-sec)
        return -heappop(stones) if stones else 0
```

Day71：k次取反后的最大数组

题目

给定一个整数数组 A，我们只能用以下方法修改该数组：我们选择某个索引 i 并将 A[i] 替换为 -A[i]，然后总共重复这个过程 K 次。（我们可以多次选择同一个索引 i。）

以这种方式修改数组后，返回数组可能的最大和。

示例 1：

输入：A = [4,2,3], K = 1
输出：5
解释：选择索引 (1,) ，然后 A 变为 [4,-2,3]。
示例 2：

输入：A = [3,-1,0,2], K = 3
输出：6
解释：选择索引 (1, 2, 2) ，然后 A 变为 [3,1,0,2]。
示例 3：

输入：A = [2,-3,-1,5,-4], K = 2

输出：13
解释：选择索引 (1, 4) ，然后 A 变为 [2,3,-1,5,4]。

提示：

1 <= A.length <= 10000
1 <= K <= 10000
-100 <= A[i] <= 100

分析

先排序，然后计算A中小于0的元素个数i； 如果K比i小，那么取反的操作都可以对负数完成，为了和最大，就选择最小的K个负数取反即可； 如果K不比i小，那么先把i个负数取反，因为同一个数可以多次取反，那么剩下的取反操作只对此时A中最小的元素进行即可。

代码

```python
class Solution:
    def largestSumAfterKNegations(self, A: List[int], K: int) -> int:
        A.sort()
        n = len(A)
        # 找到A中第一个非负数的索引
        i = 0
        while i < n:
            if A[i] >= 0:
                break
            i += 1
        # 找到后，说明A中有i个负数
        # 如果K比i小，就把前K个数取反就行
        if K < i:
            for j in range(K):
                A[j] = -1 * A[j]
            return sum(A)
        # 如果K不小于i
        else:
            # 先把i个数取反
            for j in range(i):
                A[j] = -1 * A[j]
            # 此时最小的数是A[i]或A[i-1],为了和最大，剩下的取反操作只对最小的数进行
            if (K-i) % 2 == 0:
                return sum(A)
            else:
                if i == n or (i-1 >= 0 and A[i-1] < A[i]):
                    A_min = A[i-1]
                else:
                    A_min = A[i]
                return sum(A) - 2 * A_min
```

解法2

使用最小堆，每次取出最小的元素，并取反后压入堆，并重新构建堆，重复K次后返回的数组即为K次取反后最大的数组，求和：

```python
import heapq
class Solution:
    def largestSumAfterKNegations(self, A: List[int], K: int) -> int:
        heapq.heapify(A)
        for i in range(K):
            heapq.heappush(A,-heapq.heappop(A))
        return sum(A)
```

Day72：判断子序列

题目

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

你可以认为 s 和 t 中仅包含英文小写字母。字符串 t 可能会很长（长度 ~= 500,000），而 s 是个短字符串（长度 <=100）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

示例 1:
s = "abc", t = "ahbgdc"

返回 true.

示例 2:
s = "axc", t = "ahbgdc"

返回 false.

双指针解法

归纳为快慢指针问题：

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        # 双指针
        m, n = len(s), len(t)
        i = j = 0

        while i < m and j < n:
            if s[i] == t[j]:
                i += 1
            j += 1

        return i == m
```

递归解法

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        # 递归
        m, n = len(s), len(t)
        # 递归终止
        if m == 0:
            # t为空
            return True

        if m > n:
            return False

        for i in range(n):
            if t[i] == s[0]:
                return self.isSubsequence(s[1:], t[i+1:])

        return False
```

二分法解法

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        word_set = {}
        for i, j in enumerate(t):
            if j not in word_set:
                word_set[j] = [i]
            else:
                word_set[j].append(i)
        idx = -1
        for word in s:
            if word not in word_set:
                return False
            idxs = word_set[word]
            left, right = 0, len(idxs)
            while left < right:
                mid = (left + right) // 2
                if idxs[mid] > idx:
                    right = mid
                else:
                    left = mid + 1
            if left == len(idxs):
                return False
            idx = idxs[left]

        return True
```

动态规划解法

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        n, m = len(s), len(t)
        dp = [[0] * 26 for _ in range(m)]    #
        dp.append([m] * 26)

        for i in range(m-1,-1,-1):
            for j in range(26):
                dp[i][j] = i if ord(t[i]) == j + ord('a') else dp[i+1][j]

        idx = 0
        for i in range(n):
            if dp[idx][ord(s[i]) - ord('a')] == m:
                return False

            idx = dp[idx][ord(s[i]) - ord('a')] + 1


        return True
```

Day73：分割平衡字符串

题目

在一个「平衡字符串」中，'L' 和 'R' 字符的数量是相同的。

给出一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

返回可以通过分割得到的平衡字符串的最大数量。

示例 1：

输入：s = "RLRRLLRLRL"
输出：4
解释：s 可以分割为 "RL", "RRLL", "RL", "RL", 每个子字符串中都包含相同数量的 'L' 和 'R'。

示例 2：

输入：s = "RLLLLRRRLR"

输出：3
解释：s 可以分割为 "RL", "LLLRRR", "LR", 每个子字符串中都包含相同数量的 'L' 和 'R'。

示例 3：

输入：s = "LLLLRRRR"
输出：1
解释：s 只能保持原样 "LLLLRRRR".

提示：

1 <= s.length <= 1000
s[i] = 'L' 或 'R'
分割得到的每个字符串都必须是平衡字符串。

题目分析

```
字符串 RLLLLRRRLR 可被分解为：
"RL", "LLLRRR", "LR", 每个子字符串中都包含相同数量的 'L' 和 'R'。
```

寻找策略：

1）每次遇上R字符，sumc加1

2. 遇上L字符，sumc减1
3. 当 sumc 等于0时，说明找到一个平衡字符串，平衡字符串size加1

代码

```
class Solution(object):
    def balancedStringSplit(self, s):
        """
        :type s: str
        :rtype: int
        """
        sumc,size = 0,0
        for c in s:
            sumc = sumc+1 if c == 'L' else sumc-1
            if sumc == 0:
                size += 1
        return size
```

代码时间复杂度为 O(n)，空间复杂度为O(1)

Day74：数据流中的第K大元素

题目

设计一个找到数据流中第K大元素的类（class）。注意是排序后的第K大元素，不是第K个不同的元素。

你的 KthLargest 类需要一个同时接收整数 k 和整数数组nums 的构造器，它包含数据流中的初始元素。每次调用 KthLargest.add，返回当前数据流中第K大的元素。

示例

示例:

int k = 3;
int[] arr = [4,5,8,2];
KthLargest kthLargest = new KthLargest(3, arr);
kthLargest.add(3); // returns 4
kthLargest.add(5); // returns 5
kthLargest.add(10); // returns 5
kthLargest.add(9); // returns 8
kthLargest.add(4); // returns 8

分析

使用堆数据结构求解此题，Python中内置的heapq模块为小根堆

首先，对数组完成堆化；

其次，因为堆化后，数组第一个元素为最小值，但是第二个元素不一定为第二小元素，这个大家要注意。

使用heappop方法，弹出的一定是当前堆的最小值，并且弹出后剩余元素重新建立堆，保证堆的性质被完全满足。

所以，基于这一点，对堆剪枝，只剩余k个元素，这样剩余的第一个元素一定是最小值元素，且是当前全局数组的第k大元素。

因此，得到下面代码：

代码

```
import heapq

class KthLargest(object):

    def __init__(self, k, nums):
        """
        :type k: int
        :type nums: List[int]
        """
        self.k = k
        self.nums = nums
        heapq.heapify(self.nums)
        while len(self.nums)>self.k:
            heapq.heappop(self.nums)


    def add(self, val):
        """
        :type val: int
        :rtype: int
        """
        heapq.heappush(self.nums,val)
        while len(self.nums)>self.k:
            heapq.heappop(self.nums)
        return self.nums[0]
```

Day75：前K个高频元素

题目

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]

示例 2:

输入: nums = [1], k = 1

输出: [1]

提示：

你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。

题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
你可以按任意顺序返回答案。

代码

```python
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        val_set = {key:0 for key in nums}
        for num in nums:
            val_set[num] += 1

        topk_val = []
        pq = []
        for key, val in val_set.items():
            if len(pq) < k:
                heapq.heappush(pq,(val,key))
            elif val > pq[0][0]:
                heapq.heapreplace(pq,(val,key))

        while pq:
            topk_val.append(heapq.heappop(pq)[1])

        return topk_val
```

Day76：数组中的第K个最大元素

题目

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

示例 1:

输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
示例 2:

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

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

代码

```python
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heapq.heapify(nums)
        while len(nums) > k:
            heapq.heappop(nums)
        return heapq.heappop(nums)
```

Day77：求网络延迟时间

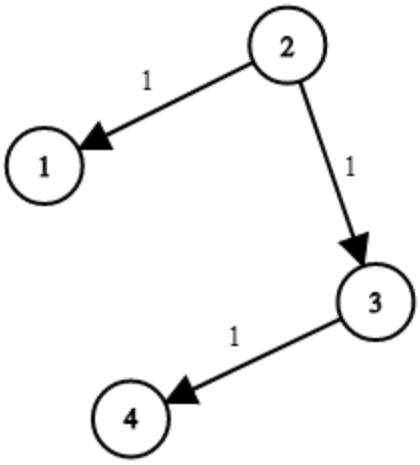
题目

有 N 个网络节点，标记为 1 到 N。

给定一个列表 times，表示信号经过有向边的传递时间。 times[i] = (u, v, w)，其中 u 是源节点，v 是目标节点， w 是一个信号从源节点传递到目标节点的时间。

现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1。

示例：




```
输入：times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2
输出：2
```

注意:

N 的范围在 [1, 100] 之间。
K 的范围在 [1, N] 之间。
times 的长度在 [1, 6000] 之间。
所有的边 times[i] = (u, v, w) 都有 1 <= u, v <= N 且 0 <= w <= 100。

分析

dijkstra算法

用小根堆来维护已到达的点与剩下的点之间的到达时间，每次选时间最小的点（也就是小根堆的堆顶保存的点）加入到已到达的点中。

代码

```python
class Solution:
    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:
        # dijkstra算法
        # 用arrived表示已经到达的点集，初始就是[k]
        # 然后每次在剩下的点里找，arrived中能到达且达到时间最小的点加入到arrived中
        # 可以用一个小顶堆来维护剩下的点及其与arrived中点的最小时间，

        # key为源点u，value存u能到达的点以及所需时间
        graph = collections.defaultdict(list)
        for u, v, w in times:
            graph[u].append((v, w))

        dist = dict()
        pq = [(0, K)]
        while pq:
            # 每次从堆中弹出的点就是距离arrived最小的点
            d, node = heapq.heappop(pq)
            if node in dist:
                continue
            # 将其加入到arrived中
            dist[node] = d
            if len(dist) == N:
                break
            for v, w in graph[node]:
                if v not in dist:
                    heapq.heappush(pq, (d+w, v))

        if len(dist) < N:
            return -1
        return max(dist.values())
```

Day78：创建小根堆

如题，已知数组nums，自己实现heapify函数，返回一个小根堆heap.

代码

```python
#实现小根堆
import numpy as np
class minHeap:
    def __init__(self,nums):
        self.nums = nums
        self.size = len(nums)

    def heapify(self,parentIndex):
        temp = self.nums[parentIndex]
        childIndex = 2 * parentIndex + 1
        while childIndex < self.size:
            if childIndex + 1 < self.size and self.nums[childIndex + 1] < self.nums[childIndex]:
                childIndex += 1
            if temp <= self.nums[childIndex]:
                break

            self.nums[parentIndex] = self.nums[childIndex]
            parentIndex = childIndex
            childIndex = 2 * childIndex + 1

        self.nums[parentIndex] = temp

    def build_heap(self):
        for i in range(self.size // 2,-1,-1):
            self.heapify(i)

if __name__ == "__main__":
    nums = list(range(8))
    np.random.shuffle(nums)
    print("before:",nums)
    heap = minHeap(nums)
    heap.build_heap()
    print("later:",nums)
    print(nums[0])
```

Day79：堆插入一个元素

如题，已知堆heap，插入元素num，依然保持是一个堆。

代码

```
import numpy as np
class minHeap:
    def __init__(self,nums):
        self.nums = nums
        self.size = len(nums)

    def heapify(self,parentIndex):
        temp = self.nums[parentIndex]
        childIndex = 2 * parentIndex + 1
        while childIndex < self.size:
            if childIndex + 1 < self.size and self.nums[childIndex + 1] < self.nums[childIndex]:
                childIndex += 1
            if temp <= self.nums[childIndex]:
                break
            self.nums[parentIndex] = self.nums[childIndex]
            parentIndex = childIndex
            childIndex = 2 * childIndex + 1
        self.nums[parentIndex] = temp

    def build_heap(self):
        for i in range(self.size // 2,-1,-1):
            self.heapify(i)


    def up_adjust(self,low,high):
        child = high
        parent = child // 2
        while parent >= low:
            if self.nums[child] < self.nums[parent]:
                self.nums[child], self.nums[parent] = self.nums[parent],self.nums[child]
            else:
                break

    def heap_push(self,num):
        self.nums.append(num)
        self.up_adjust(0,len(nums)-1)
```

Day80：设计推特

题目

设计一个简化版的推特(Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近十条推文。你的设计需要支持以下的几个功能：

postTweet(userId, tweetId): 创建一条新的推文
getNewsFeed(userId): 检索最近的十条推文。每个推文都必须是由此用户关注的人或者是用户自己发出的。推文必须按照时间顺序由最近的开始排序。

follow(followerId, followeeId): 关注一个用户

unfollow(followerId, followeeId): 取消关注一个用户

示例:

```
Twitter twitter = new Twitter();

// 用户1发送了一条新推文 (用户id = 1, 推文id = 5).
twitter.postTweet(1, 5);

// 用户1的获取推文应当返回一个列表，其中包含一个id为5的推文.
twitter.getNewsFeed(1);

// 用户1关注了用户2.
twitter.follow(1, 2);

// 用户2发送了一个新推文 (推文id = 6).
twitter.postTweet(2, 6);

// 用户1的获取推文应当返回一个列表，其中包含两个推文，id分别为 -> [6, 5].
// 推文id6应当在推文id5之前，因为它是在5之后发送的.
twitter.getNewsFeed(1);

// 用户1取消关注了用户2.
twitter.unfollow(1, 2);

// 用户1的获取推文应当返回一个列表，其中包含一个id为5的推文.
// 因为用户1已经不再关注用户2.
twitter.getNewsFeed(1);
```

代码

```python
from collections import defaultdict
import heapq
class Tweet:
    def __init__(self,tweetId,timestamp):
        self.tweetId =  tweetId
        self.timestamp = timestamp
        self.next = None

    def __lt__(self,other):
        return self.timestamp > other.timestamp

class Twitter:

    def __init__(self):
        self.followings = defaultdict(set)
        self.tweets = defaultdict(lambda:None)
        self.timestamp = 0

    def postTweet(self, userId: int, tweetId: int) -> None:
        self.timestamp += 1
        tweet = Tweet(tweetId,self.timestamp)
        tweet.next = self.tweets[userId]
        self.tweets[userId] = tweet

    def getNewsFeed(self, userId: int) -> List[int]:
        tweets = []
        News_heap = []
        tweet = self.tweets[userId]
        if tweet:
            News_heap.append(tweet)
        for user in self.followings[userId]:
            tweet = self.tweets[user]
            if tweet:
                News_heap.append(tweet)

        heapq.heapify(News_heap)

        while News_heap and len(tweets) < 10:
            head = heapq.heappop(News_heap)
            tweets.append(head.tweetId)

            if head.next:
                heapq.heappush(News_heap,head.next)

        return tweets

    def follow(self, followerId: int, followeeId: int) -> None:
        if followerId == followeeId:
            return
        self.followings[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        if followerId == followeeId:
            return
        self.followings[followerId].discard(followeeId)
```

Day81：超级丑数

题目

编写一段程序来查找第 n 个超级丑数。

超级丑数是指其所有质因数都是长度为 k 的质数列表 primes 中的正整数。

示例:

输入: n = 12, primes = [2,7,13,19]
输出: 32
解释: 给定长度为 4 的质数列表 primes = [2,7,13,19]，前 12 个超级丑数序列为：[1,2,4,7,8,13,14,16,19,26,28,32] 。

说明:

1 是任何给定 primes 的超级丑数。
给定 primes 中的数字以升序排列。
0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000 。
第 n 个超级丑数确保在 32 位有符整数范围内。

代码

堆解法

```python
# 超级丑数,堆解法
import heapq
class Solution:
    def nthSuperUglyNumber(self, n: int, primes: List[int]) -> int:
        hashmap = {1,}
        heap = []
        heapq.heappush(heap,1)
        for _ in range(n):
            cur_ugly = heapq.heappop(heap)
            for p in primes:
                new_ugly = cur_ugly * p
                if new_ugly not in hashmap:
                    hashmap.add(new_ugly)
                    heapq.heappush(heap,new_ugly)

        return cur_ugly
```

动态规划解法

```python
# 超级丑数,动态规划解法
class Solution:
    def nthSuperUglyNumber(self, n: int, primes: List[int]) -> int:
        dp = [0] * n
        pointer = [0] * len(primes)
        dp[0] = 1
        for i in range(1,n):
            dp[i] = min(x * dp[y] for x, y in zip(primes,pointer))
            for j in range(len(primes)):
                if dp[i] == primes[j] * dp[pointer[j]]:
                    pointer[j] += 1
        return dp[-1]
```

Day82：最近请求次数

题目

写一个 RecentCounter 类来计算特定时间范围内最近的请求。

请你实现 RecentCounter 类：

```
RecentCounter() 初始化计数器，请求数为 0 。
int ping(int t) 在时间 t 添加一个新请求，其中 t 表示以毫秒为单位的某个时间，并返回过去 3000 毫秒内发生的
保证 每次对 ping 的调用都使用比之前更大的 t 值。
```

示例：

输入：

```
["RecentCounter", "ping", "ping", "ping", "ping"]
[[], [1], [100], [3001], [3002]]
输出:
[null, 1, 2, 3, 3]
```

解释：

```
RecentCounter recentCounter = new RecentCounter();
recentCounter.ping(1);     // requests = [1]，范围是 [-2999,1]，返回 1
recentCounter.ping(100);   // requests = [1, 100]，范围是 [-2900,100]，返回 2
recentCounter.ping(3001);  // requests = [1, 100, 3001]，范围是 [1,3001]，返回 3
recentCounter.ping(3002);  // requests = [1, 100, 3001, 3002]，范围是 [2,3002]，返回 3
```

提示：

1 <= t <= 109
保证每次对 ping 调用所使用的 t 值都 严格递增
至多调用 ping 方法 104 次

队列求解

```
from collections import deque

class RecentCounter(object):

    def __init__(self):
        self.q = deque([])

    def ping(self, t):
        """
        :type t: int
        :rtype: int
        """
        self.q.append(t)
        while len(self.q) > 0 and self.q[0] + 3000 < t:
            self.q.popleft()
        return len(self.q)
```

Day83：滑动窗口的最大值

题目

给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。

示例:

输入: nums = [1,3,-1,-3,5,3,6,7], 和 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

提示：

你可以假设 k 总是有效的，在输入数组不为空的情况下，1 ≤ k ≤ 输入数组的大小。

单调队列解法

```python
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums:
            return []
        # 单调队列
        # 构建窗口
        q = collections.deque()
        for i in range(k):
            while q and q[-1] < nums[i]:
                q.pop()
            q.append(nums[i])
        res = [q[0]]

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

        return res
```

Day84：设计循环队列

题目

设计你的循环队列实现。 循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

你的实现应该支持如下操作：

MyCircularQueue(k): 构造器，设置队列长度为 k 。
Front: 从队首获取元素。如果队列为空，返回 -1 。
Rear: 获取队尾元素。如果队列为空，返回 -1 。
enQueue(value): 向循环队列插入一个元素。如果成功插入则返回真。
deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。
isEmpty(): 检查循环队列是否为空。
isFull(): 检查循环队列是否已满。

示例：

MyCircularQueue circularQueue = new MyCircularQueue(3); // 设置长度为 3
circularQueue.enQueue(1); // 返回 true
circularQueue.enQueue(2); // 返回 true
circularQueue.enQueue(3); // 返回 true
circularQueue.enQueue(4); // 返回 false，队列已满
circularQueue.Rear(); // 返回 3
circularQueue.isFull(); // 返回 true
circularQueue.deQueue(); // 返回 true
circularQueue.enQueue(4); // 返回 true
circularQueue.Rear(); // 返回 4

提示：

所有的值都在 0 至 1000 的范围内；
操作数将在 1 至 1000 的范围内；
请不要使用内置的队列库。

基于数组解法

```python
class MyCircularQueue:

    def __init__(self, k: int):

        self.headIndex = 0
        self.tailIndex = 0
        self.capacity = k + 1
        self.arr = [0 for _ in range(self.capacity)]

    def enQueue(self, value: int) -> bool:

        if self.isFull():
            return False
        self.arr[self.tailIndex] = value
        self.tailIndex = (self.tailIndex + 1) % self.capacity
        return True


    def deQueue(self) -> bool:

        if self.isEmpty():
            return False
        self.headIndex = (self.headIndex + 1) % self.capacity
        return True

    def Front(self) -> int:

        if self.isEmpty():
            return -1
        return self.arr[self.headIndex]


    def Rear(self) -> int:

        if self.isEmpty():
            return -1
        return self.arr[(self.tailIndex - 1 + self.capacity) % self.capacity]

    def isEmpty(self) -> bool:

        return self.headIndex == self.tailIndex

    def isFull(self) -> bool:

        return (self.tailIndex + 1) % self.capacity == self.headIndex
```

Day85：第K个数

题目

有些数的素因子只有 3，5，7，请设计一个算法找出第 k 个数。注意，不是必须有这些素因子，

而是必须不包含其他的素因子。例如，前几个数按顺序应该是 1，3，5，7，9，15，21。

基于堆解法

```python
import heapq
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        hashmap = {1,}
        heap = []
        heapq.heappush(heap,1)
        for _ in range(k):
            cur_num = heapq.heappop(heap)
            for num in [3,5,7]:
                new_num = cur_num * num
                if new_num not in hashmap:
                    hashmap.add(new_num)
                    heapq.heappush(heap,new_num)

        return cur_num
```

三指针解法

```python
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        magic_nums = [1]
        three = five = seven = 0
        for i in range(k-1):
            t = magic_nums[three] * 3
            f = magic_nums[five] * 5
            s = magic_nums[seven] * 7
            magic_nums.append(min(t, f, s))
            if magic_nums[-1] == t:
                three += 1
            if magic_nums[-1] == f:
                five += 1
            if magic_nums[-1] == s:
                seven += 1

        return magic_nums[-1]
```

Day86：经典汉诺塔问题

题目

在经典汉诺塔问题中，有 3 根柱子及 N 个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自上而下按升序依次套在第一根柱子上(即每一个盘子只能放在更大的盘子上面)。移动圆盘时受到以下限制:
(1) 每次只能移动一个盘子;
(2) 盘子只能从柱子顶端滑出移到下一根柱子;
(3) 盘子只能叠在比它大的盘子上。

请编写程序，用栈将所有盘子从第一根柱子移到最后一根柱子。

你需要原地修改栈。

示例1:

输入：A = [2, 1, 0], B = [], C = []
输出：C = [2, 1, 0]
示例2:

输入：A = [1, 0], B = [], C = []
输出：C = [1, 0]

递归解法

```python
class Solution:
    def hanota(self, A: List[int], B: List[int], C: List[int]) -> None:
        """
        Do not return anything, modify C in-place instead.
        """
        # 用递归的思路将其分解为子问题
        # 把A的上面n-1盘子通过C移动到B
        # 把A的最后一个盘子移动到C
        # 再把B里的n-1盘子通过A移动到C
        # 因为是原地修改，所以定义一个辅助函数

        def move(n, A, B, C):
            if n == 1:
                # 递归终止
                C.append(A.pop())
            else:
                # 把A的上面n-1盘子通过C移动到B
                move(n-1, A, C, B)
                # 把A的最后一个盘子移动到C
                C.append(A.pop())
                # 再把B里的n-1盘子通过A移动到C
                move(n-1, B, A, C)

        move(len(A), A, B, C)
```

Day87：最小栈

题目

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

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

示例:

输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]

],[-2],[0],[-3],[],[],[],[

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); --> 返回 -3.
minStack.pop();
minStack.top(); --> 返回 0.
minStack.getMin(); --> 返回 -2.

提示：

pop、top 和 getMin 操作总是在 非空栈 上调用。

单调栈分析

用到了单调栈的思路；

除了用一个普通的栈来实现压入弹出操作外，还要用一个辅助栈来保存当前栈的最小元素，最小元素就是辅助栈的栈顶元素。

为什么叫把这个辅助栈叫做单调栈呢？根据辅助栈的作用：

入栈时，如果入栈元素大于辅助栈的栈顶元素，那么入栈元素一定不是当前栈的最小元素，且由于栈的先进后出特性，它一定比先入栈的元素先出栈，无论如何都不会是栈的最小值，所以不能加入辅助栈；而不比辅助栈的栈顶元素大的话，那么它就是当前栈的最小元素，要添加到辅助栈中。可以发现，按照这样的入栈思路，辅助栈的元素从栈底到栈顶是非严格单调递减的，所以称之为单调栈。

而出栈的时候，需要将出栈元素x与辅助栈的栈顶元素y比较，显然x>=y；如果x>y，此时栈中的最小元素还是y，辅助栈不用弹出；如果x=y，那么此时也要将y从辅助栈中弹出。

代码

```python
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        # 一个普通的栈，用来实现弹入和弹出
        self.stack = []
        # 一个单调栈，用来输出最小元素
        self.min_stack = []


    def push(self, x: int) -> None:
        self.stack.append(x)
        # 当单调栈为空，或者x不大于栈顶元素的时候才把x添加到单调栈中
        if not self.min_stack or self.min_stack[-1] >= x:
            self.min_stack.append(x)
        # 也就是说从栈底到栈顶，元素是非严格单调递减的


    def pop(self) -> None:
        x = self.stack.pop()
        if self.min_stack[-1] == x:
            self.min_stack.pop()


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

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

Day88：有效的括号

题目

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

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

输入: "()"
输出: true
示例 2:

输入: "()[]{}"
输出: true
示例 3:

输入: "(]"
输出: false
示例 4:

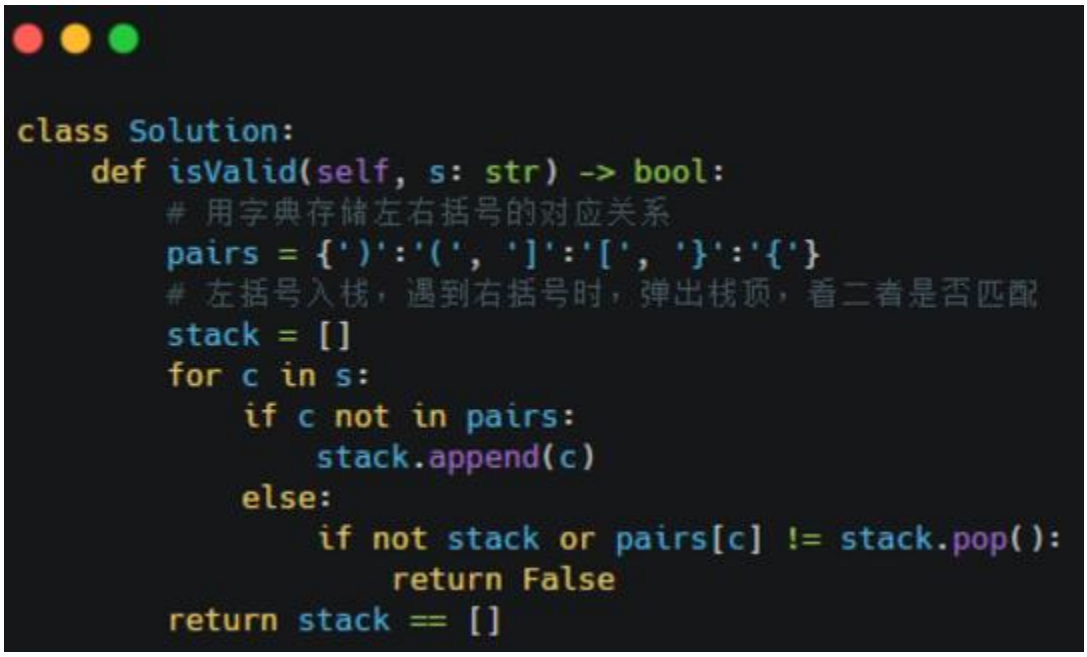
输入: "([)]"
输出: false
示例 5:

输入: "{[]}"
输出: true

使用栈

先用一个字典存储左右括号的对应关系，方便查找； 用栈来模拟配对过程；左括号入栈，右括号则弹出栈顶元素，并判断栈顶是否与右括号匹配

代码



```python
class Solution:
    def isValid(self, s: str) -> bool:
        # 用字典存储左右括号的对应关系
        pairs = {')':'(', ']':'[', '}':'{'}
        # 左括号入栈，遇到右括号时，弹出栈顶，看二者是否匹配
        stack = []
        for c in s:
            if c not in pairs:
                stack.append(c)
            else:
                if not stack or pairs[c] != stack.pop():
                    return False
        return stack == []
```

Day89：

题目

给定两个 没有重复元素 的数组 nums1 和 nums2 ，其中nums1 是 nums2 的子集。找到 nums1 中每个元素在 nums2 中的下一个比其大的值。

nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置的右边的第一个比 x 大的元素。如果不存在，对应位置输出 -1 。

示例 1:

输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释:
对于num1中的数字4，你无法在第二个数组中找到下一个更大的数字，因此输出 -1。
对于num1中的数字1，第二个数组中数字1右边的下一个较大数字是 3。
对于num1中的数字2，第二个数组中没有下一个更大的数字，因此输出 -1。

示例 2:

输入: nums1 = [2,4], nums2 = [1,2,3,4].
输出: [3,-1]
解释:
对于 num1 中的数字 2 ，第二个数组中的下一个较大数字是 3 。
对于 num1 中的数字 4 ，第二个数组中没有下一个更大的数字，因此输出 -1 。

提示：

nums1和nums2中所有元素是唯一的。
nums1和nums2 的数组大小都不超过1000。

基于单调栈分析

单调栈+字典

对nums2遍历，如果当前元素num小于栈顶x，入栈；否则，只要num大于栈顶，将弹出栈顶，并且栈顶元素右边第一个比它大的元素就是num，将这个关系存入字典中。

由入栈条件可知，从栈底到栈顶是单调递减的；假设栈顶为x，x<num，如果在x到num之间存在一个比x大的元素y，那么在遍历到y的时候，x就已经出栈了。

代码

```python
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        # 单调栈
        stack = []
        # seen[x]存储nums2中x右边第一个比x大的元素
        seen = dict()

        # 遍历nums2，将元素存入stack中并保证stack从栈底到栈顶是递减的
        for num in nums2:
            while stack and num > stack[-1]:
                # 如果栈顶元素x小于num
                # 将x弹出，并且可知x右边第一个比x大的元素就是num
                # 假如在x和num之间有元素y比x大，那么在遍历到y的时候，x就出栈了
                x = stack.pop()
                seen[x] = num
            stack.append(num)

        n = len(nums1)
        res = [-1] * n
        for i in range(n):
            if nums1[i] in seen:
                res[i] = seen[nums1[i]]
        return res
```

Day90：棒球比赛的分数

题目

你现在是一场采特殊赛制棒球比赛的记录员。这场比赛由若干回合组成，过去几回合的得分可能会影响以后几回合的得分。

比赛开始时，记录是空白的。你会得到一个记录操作的字符串列表 ops，其中 ops[i] 是你需要记录的第 i 项操作，ops 遵循下述规则：

整数 x - 表示本回合新获得分数 x
"+" - 表示本回合新获得的得分是前两次得分的总和。题目数据保证记录此操作时前面总是存在两个有效的分数。
"D" - 表示本回合新获得的得分是前一次得分的两倍。题目数据保证记录此操作时前面总是存在一个有效的分数。

"C" - 表示前一次得分无效，将其从记录中移除。题目数据保证记录此操作时前面总是存在一个有效的分数。
请你返回记录中所有得分的总和。

示例 1：

输入：ops = ["5","2","C","D","+"]
输出：30
解释：
"5" - 记录加 5 ，记录现在是 [5]
"2" - 记录加 2 ，记录现在是 [5, 2]
"C" - 使前一次得分的记录无效并将其移除，记录现在是 [5].
"D" - 记录加 2 * 5 = 10 ，记录现在是 [5, 10].
"+" - 记录加 5 + 10 = 15 ，记录现在是 [5, 10, 15].
所有得分的总和 5 + 10 + 15 = 30

示例 2：

输入：ops = ["5","-2","4","C","D","9","+","+"]
输出：27
解释：
"5" - 记录加 5 ，记录现在是 [5]
"-2" - 记录加 -2 ，记录现在是 [5, -2]
"4" - 记录加 4 ，记录现在是 [5, -2, 4]
"C" - 使前一次得分的记录无效并将其移除，记录现在是 [5, -2]
"D" - 记录加 2 * -2 = -4 ，记录现在是 [5, -2, -4]
"9" - 记录加 9 ，记录现在是 [5, -2, -4, 9]
"+" - 记录加 -4 + 9 = 5 ，记录现在是 [5, -2, -4, 9, 5]
"+" - 记录加 9 + 5 = 14 ，记录现在是 [5, -2, -4, 9, 5, 14]
所有得分的总和 5 + -2 + -4 + 9 + 5 + 14 = 27

示例 3：

输入：ops = ["1"]
输出：1

提示：

1 <= ops.length <= 1000
ops[i] 为 "C"、"D"、"+"，或者一个表示整数的字符串。整数范围是 [-3 * 104, 3 * 104]
对于 "+" 操作，题目数据保证记录此操作时前面总是存在两个有效的分数
对于 "C" 和 "D" 操作，题目数据保证记录此操作时前面总是存在一个有效的分数

代码

用一个栈来模拟这个过程，这样'C'操作就可以通过出栈来实现

```python
class Solution:
    def calPoints(self, ops: List[str]) -> int:
        stack = []
        res = 0
        for op in ops:
            if op == 'C':
                res -= stack[-1]
                stack.pop()
            else:
                if op == 'D':
                    stack.append(stack[-1] * 2)
                elif op == '+':
                    stack.append(stack[-1] + stack[-2])
                else:
                    stack.append(int(op))
                res += stack[-1]

        return res
```

Day91：整理字符串

题目

给你一个由大小写英文字母组成的字符串 s 。

一个整理好的字符串中，两个相邻字符 s[i] 和 s[i+1]，其中 0<= i <= s.length-2 ，要满足如下条件:

若 s[i] 是小写字符，则 s[i+1] 不可以是相同的大写字符。
若 s[i] 是大写字符，则 s[i+1] 不可以是相同的小写字符。
请你将字符串整理好，每次你都可以从字符串中选出满足上述条件的 两个相邻 字符并删除，直

到字符串整理好为止。

请返回整理好的 字符串 。题目保证在给出的约束条件下，测试样例对应的答案是唯一的。

注意：空字符串也属于整理好的字符串，尽管其中没有任何字符。

示例 1：

输入：s = "leEeetcode"
输出："leetcode"
解释：无论你第一次选的是 i = 1 还是 i = 2，都会使 "leEeetcode" 缩减为 "leetcode" 。
示例 2：

输入：s = "abBAcC"
输出：""
解释：存在多种不同情况，但所有的情况都会导致相同的结果。例如：
"abBAcC" --> "aAcC" --> "cC" --> ""
"abBAcC" --> "abBA" --> "aA" --> ""
示例 3：

输入：s = "s"
输出："s"

提示：

1 <= s.length <= 100
s 只包含小写和大写英文字母

分析+代码

巧用栈结构处理，空栈时先添加一个字符，然后开始与字符串中的字符比较是否互为大小写字母，如果是则将栈顶元素出栈否则将字符串中对应的字符入栈，最终栈中的数据即为结果。

```python
class Solution:
    def makeGood(self, s: str) -> str:
        # 栈来完成
        stack = []
        for c in s:
            if stack and c != stack[-1] and c.upper() == stack[-1].upper():
                stack.pop()
            else:
                stack.append(c)
        return ''.join(stack)
```

Day92：括号的分数

题目

给定一个平衡括号字符串 S，按下述规则计算该字符串的分数：

() 得 1 分。
AB 得 A + B 分，其中 A 和 B 是平衡括号字符串。
(A) 得 2 * A 分，其中 A 是平衡括号字符串。

示例 1：

输入："()"
输出：1

示例 2：

输入："(())"
输出：2

示例 3：

输入："()()"
输出：2
示例 4：

输入："((()()))"
输出：6

提示：

S 是平衡括号字符串，且只含有 (和) 。
2 <= S.length <= 50

代码

```
class Solution:
    def scoreOfParentheses(self, S):
        stack = [0]
        for s in S:
            if s == "(":
                stack.append(0)
            else:
                top = stack.pop()
                stack[-1] += max(2 * top, 1)
        return stack[-1]


if __name__ == "__main__":
    STR = "(()(()))"
    S = Solution()
    print(S.scoreOfParentheses(STR))
```

Day93：每日温度

题目

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。

单调栈解法

还是之前单调栈的思路。 这种在列表中查找第一个大于或小于当前元素的题型，其实都可以考

虑能否采用单调栈的思路。

代码

```python
class Solution:
    def dailyTemperatures(self, T: List[int]) -> List[int]:
        # 单调栈
        n = len(T)
        res = [0] * n

        stack = []
        # 入栈的是气温列表的索引
        for i in range(n):
            while stack and T[stack[-1]] < T[i]:
                # 当栈顶元素对应的气温小于当前对应的气温
                # 当前就是第一次高于栈顶日期温度的日期，更新res数组，并将栈顶弹出
                # 确保栈从底到顶是非严格单调递增的
                res[stack[-1]] = i - stack[-1]
                stack.pop()
            stack.append(i)

        return res
```

Day94：实现 int sqrt(int x) 函数

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

输入: 4
输出: 2
示例 2:

输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,
 由于返回类型是整数，小数部分将被舍去。

链接：https://leetcode-cn.com/problems/sqrtx

这道题看似非常简单，其实非常容易出错，但又经常面试被问到，故今天特别提醒下

这道题的思路大家应该都清楚，二分查找。不过，具体实现起来，容易犯三种错误：

1. 死循环
2. 求出的平方根比正确答案大1
3. 个别数的平方根求错

下面是几种写法，只有一种是正确的，欢迎辨别：

写法1

下面代码正确吗？

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        lo,hi=0,x
        while lo < hi:
            mid = lo + (hi - lo)//2
            if mid**2 <= x:
                lo = mid
            else:
                hi = mid
        return lo
```

写法2

这个呢？

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        lo,hi=0,x
        while lo < hi:
            mid = lo + (hi - lo)//2
            s = mid**2
            if s < x:
                lo = mid + 1
            elif s > x:
                hi = mid - 1
            else:
                return mid

        return lo
```

写法3

那这个呢？

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        lo,hi=0,x
        while lo < hi-1:
            mid = lo + (hi - lo)//2
            if mid**2 <= x:
                lo = mid
            else:
                hi = mid
        return lo
```

写法4

最后一个准确无误吗？

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        lo,hi=0,x
        while lo <= hi:
            mid = lo + (hi - lo)//2
            s = mid**2
            if s < x:
                lo = mid + 1
            elif s > x:
                hi = mid - 1
            else:
                return mid

        return hi
```

总结

写法1：死循环

写法2：返回错误值

写法3：1的平方根返回错误

写法4：100%正确

二分查找的思想并不难，写起来却容易出错，并且不同场景的二分查找代码之间又有一些微妙区别，可以说比较灵活，大家多多琢磨分析。

Day95：两数之和

题目

给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

返回的下标值（index1 和 index2）不是从零开始的。
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。
示例:

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 。

分析+代码

双指针

如果数组不是有序的话，也可以使用哈希表来实现O(n)的复杂度。

```python
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        # 双指针
        l, r = 0, len(numbers) - 1
        while l < r:
            two_sum = numbers[l] + numbers[r]
            if two_sum == target:
                return [l+1, r+1]
            elif two_sum < target:
                l += 1
            else:
                r -= 1
```

Day96：完全平方数

题目

给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 True，否则返回 False。

说明：不要使用任何内置的库函数，如 sqrt。

示例 1：

```
输入：16
输出：True
```

示例 2：

```
输入：14
输出：False
```

转化为二分模板

可以理解为在1到num中寻找平方为num的数。 将1到num当作一个有序数组，等价于在有序数组中查找一个数。 就转化成了二分模板题了

代码

```python
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        # 在1-num之间寻找，是否有一个数的平方等于num
        # 可以将1-num看成一个有序数组，等价于在有序数组中寻找某个数，经典的二分
        l, r = 1, num
        while l <= r:
            mid = (l + r) // 2
            if mid * mid == num:
                #找到了
                return True
            elif mid * mid < num:
                l = mid + 1
            else:
                r = mid - 1
        return False
```

Day97：排列硬币

题目

你总共有 n 枚硬币，你需要将它们摆成一个阶梯形状，第 k 行就必须正好有 k 枚硬币。

给定一个数字 n，找出可形成完整阶梯行的总行数。

n 是一个非负整数，并且在32位有符号整型的范围内。

示例 1:

n = 5

硬币可排列成以下几行:
¤
¤ ¤
¤ ¤

因为第三行不完整，所以返回2.

示例 2:

n = 8

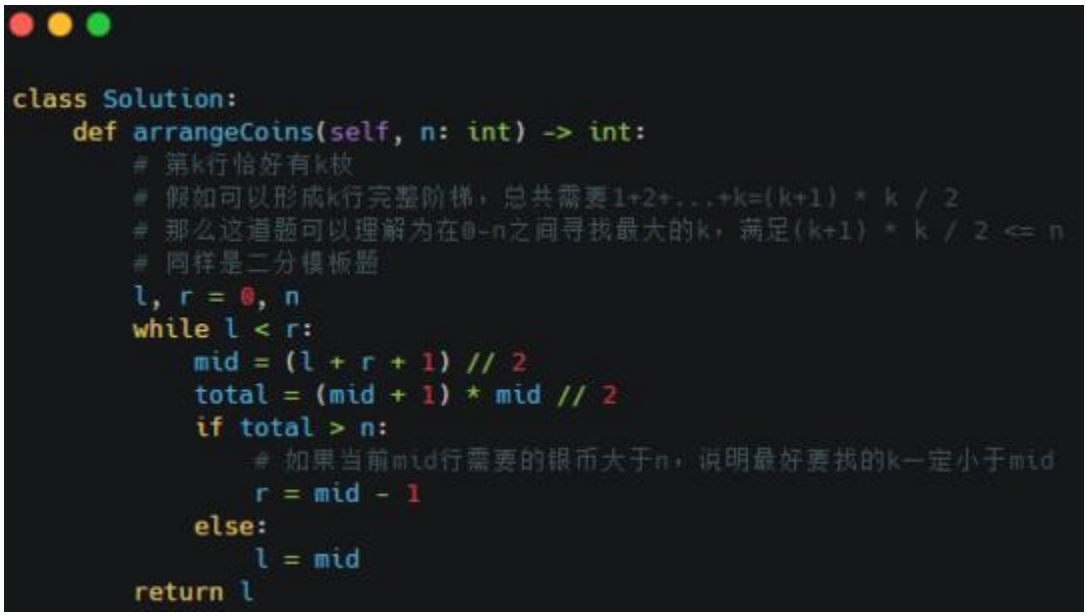
硬币可排列成以下几行:
¤
¤ ¤
¤ ¤ ¤
¤ ¤

因为第四行不完整，所以返回3.

转化为二分模板

假如可以形成k行完整阶梯，总共需要1+2+...+k=(k+1) * k / 2枚硬币； 道题可以理解为在0-n之间寻找最大的k，满足(k+1) * k / 2 <= n； 同样是二分模板题，需要注意区间缩小的条件； 在这道题里，如果计算得到的硬币数大于n，说明mid以及大于mid的情况都可以排除，于是就将r更新为mid-1.

代码



```python
class Solution:
    def arrangeCoins(self, n: int) -> int:
        # 第k行恰好有k枚
        # 假如可以形成k行完整阶梯，总共需要1+2+...+k=(k+1) * k / 2
        # 那么这道题可以理解为在0-n之间寻找最大的k，满足(k+1) * k / 2 <= n
        # 同样是二分模板题
        l, r = 0, n
        while l < r:
            mid = (l + r + 1) // 2
            total = (mid + 1) * mid // 2
            if total > n:
                # 如果当前mid行需要的银币大于n，说明最好要找的k一定小于mid
                r = mid - 1
            else:
                l = mid
        return l
```

Day98：有序矩阵中第K小的元素

题目

给定一个 n x n 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。
请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例：

matrix = [
[1, 5, 9],
[10, 11, 13],
[12, 13, 15]
],
k = 8,

返回 13。

二分查找注意事项

二分查找表面看似很简单，就是对解空间的不断二分，直至逼近解并返回。但是实操起来，却绝非这么简单。

1. 要首先分析出**对谁折半**，有些问题显而易见，如有序数组查某个数的位置，因为索引值越大，元素值就越大，所以对索引范围折半。但是，有些问题对谁折半，就会复杂一些，需要分析分析。**对谁折半**的原则：寻找一个关系 f，如果自变量 x 变大，因变量 y 就变大(小)，是一个线性正或负相关关系，则 x 便可作为折半的对象。
2. 确定好对谁折半后，二分查找的**折半条件** f 一般也就确定下来 。或者步骤1和步骤2是要相结合构思出来的，而不是串行的步骤。
3. 确定好这些后，编写代码仍然需要注意，等号取和不取时，不同的的low,high写法。
4. 再有注意 while 循环条件，low < high，low <= high，这些都会影响二分的写法。
5. 到底是返回 low ，还是 high，这要根据题目、步骤3和4的写法来确定。

写法1：

```
class Solution(object):
    def kthSmallest(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        n = len(matrix)
        low,high=matrix[0][0],matrix[-1][-1]
        while low <= high:
            mid = (low+high)//2
            low_count = 0
            i,j =n-1,0
            while i >=0 and j <=n-1:
                if matrix[i][j] < mid:
                    low_count += i+1
                    j += 1
                else:
                    i -= 1
            if low_count < k:
                low = mid + 1
            else:
                high = mid - 1
        return high
```

写法2：

```
class Solution(object):
    def kthSmallest(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        n = len(matrix)
        low,high=matrix[0][0],matrix[-1][-1]
        while low < high:
            mid = (low+high)//2
            low_count = 0
            i,j =n-1,0
            while i >=0 and j <=n-1:
                if matrix[i][j] <= mid:
                    low_count += i+1
                    j += 1
                else:
                    i -= 1
            if low_count < k:
                low = mid + 1
            else:
                high = mid

        return high
```

提出两个思考问题

借助这道题，我们深入理解二分查找。请仔细体会以上两种写法的差异，对于我们深入理解二分逼近式的锁定解，具有重要意义。

请回答以下两个问题：

1 `matrix[i][j] < mid` 和 `matrix[i][j] <= mid`，这两种求比mid小的元素个数的写法，对外层书写二分条件有什么影响？为什么？

2 二分查找返回的解一定是矩阵中的元素吗？这也是此题最需要被解释的问题之一。

比如，若

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 20]
],
k = 8
```

返回结果14，15，16 都满足是矩阵的第8小元素，但它们都不是矩阵中的元素，以上两种二分查找的解法是**如何保证不返回此类解**。

Day99：回答二分的边界

Q1：matrix[i][j] < mid 和 matrix[i][j] <= mid，这两种求比mid小的元素个数的写法，对外层书写二分条件有什么影响？

A1：影响退出循环的条件是取到等号还是大于号，以及右边界的取值。

Q2：二分查找返回的解一定是矩阵中的元素吗？

A2：以方法一分析，初始时，当矩阵中<=mid的元素个数count一直小于k时，left会不断递增，mid也在不断动态调整，直到突然出现一个mid首次满足<=mid的元素个数count大于等于k，那么此时right = mid。右边界right出现变化导致mid调整进而使得right不断变小，left可能暂时不变，直到又出现<=mid的元素个数count < num。如此最后退出循环时必然有left = right，返回left就是返回right。而退出时又有right = mid，所以mid就是最终答案。

Day100：二分查找的两种实现

二分查找问题描述

给定一个 个元素有序的（升序）整型数组 和一个目标值 ，写一个函数搜索 中的 ，如果目标值存在返回下标，否则返回 -1。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/binary-search

解法1：左闭右开

使用区间：左闭右开，详细思路见文中代码注释：

```python
class Solution(object):
    def search(self, nums, target):
        # 刚开始的取值区间：[0,len(nums) )
        left,right=0,len(nums)
        while right - left > 1:
            mid = (left+right) //2
            # nums[0]<=...<nums[mid] <= target
            if nums[mid] <= target:
                # 遍历后区间改变为：[mid,right)
                left = mid
            else:
                # nums[right-1]...>=nums[mid+1]>=nums[mid] > target
                # 区间调整为：[left,right)
                right = mid
        # 迭代结束后
        # right-left=1，区间为[left, right=left+1)
        # 即nums[left]就是二分后逼近的点
        # 判断nums[left]是否等于target即可
        return left if nums[left] == target else -1
```

解法2：左闭右闭

```
class Solution(object):
    def search(self, nums, target):
        # 左右都是闭合区间的写法
        # 刚开始的取值区间：[0,len(nums)-1]
        left,right=0,len(nums)-1
        while left <= right:
            mid = (left+right) //2
            if nums[mid] == target:
                return mid
            # nums[0]<=...<nums[mid] < target
            elif nums[mid] < target:
                # 遍历后区间改变为：[mid+1,right]
                left = mid + 1
            else:
                # nums[right-1]...>=nums[mid+1]>=nums[mid] > target
                # 区间调整为：[left,mid-1]
                right = mid-1
        # 迭代结束后
        # left - right = 1，此时区间[left,right] = [left,left-1]变为空！
        # 所以返回 -1
        return -1
```

二分查找，是最基本的分支法的一个应用，面试中被问道的频率很高，同时边界取值特别容易出错。

知识星球

星主：zhenguo

算法刷题日记

长按扫码预览社群内容
和星主关系更近一步