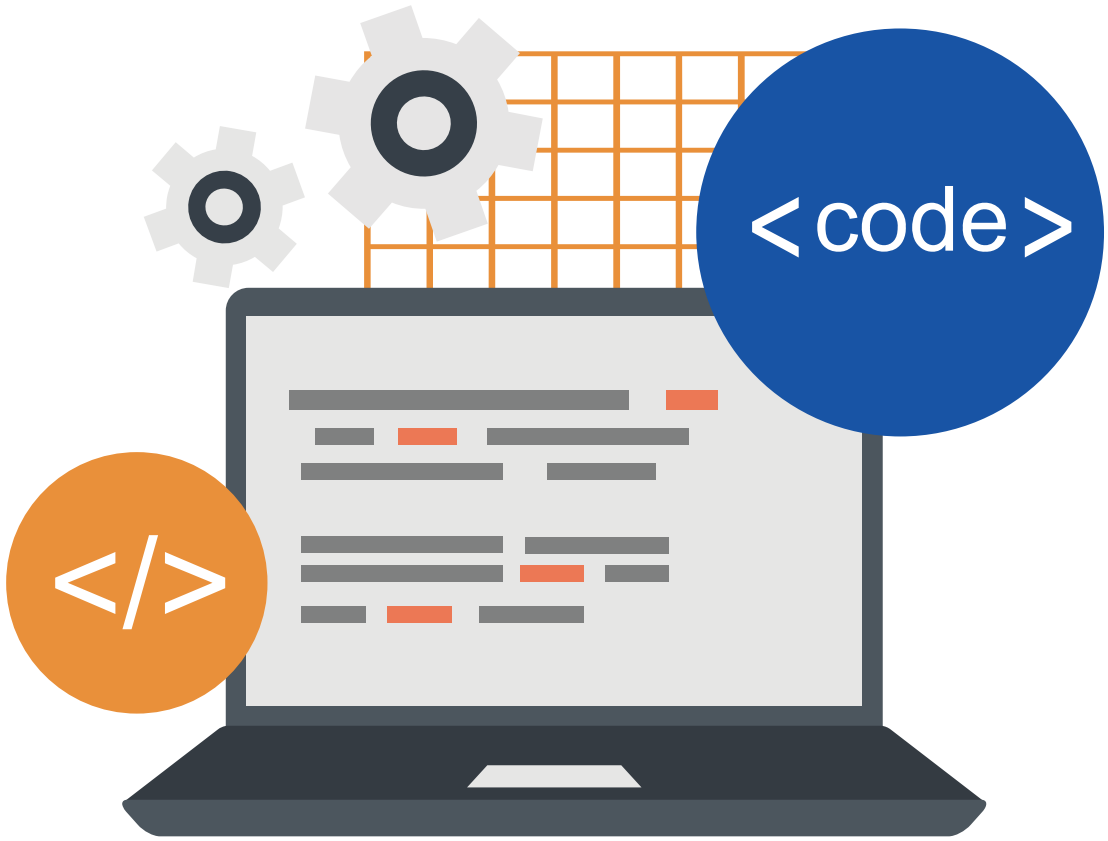
Datawhale

LeetCode组队学习

LeetCode



Datawhale 出品

内容设计

- 黑桃、LeoLRH、王嘉鹏、Axe、十三鸣、Juvien、hero、肖然

学习目标

- 能够熟练的使用LeetCode刷题，提高自身的coding能力

基本信息

- 定位人群：具有一定编程语言基础；
- 时间安排：10天，每天平均花费时间3小时-5小时不等，根据个人学习接受能力强弱有所浮动
- 学习类型：刷题实践
- 难度系数：中

学习内容

Task01：分治

MapReduce（分治算法的应用） 是 Google 大数据处理的三驾马车之一，另外两个是 GFS 和 Bigtable。它在倒排索引、PageRank 计算、网页分析等搜索引擎相关的技术中都有大量的应用。

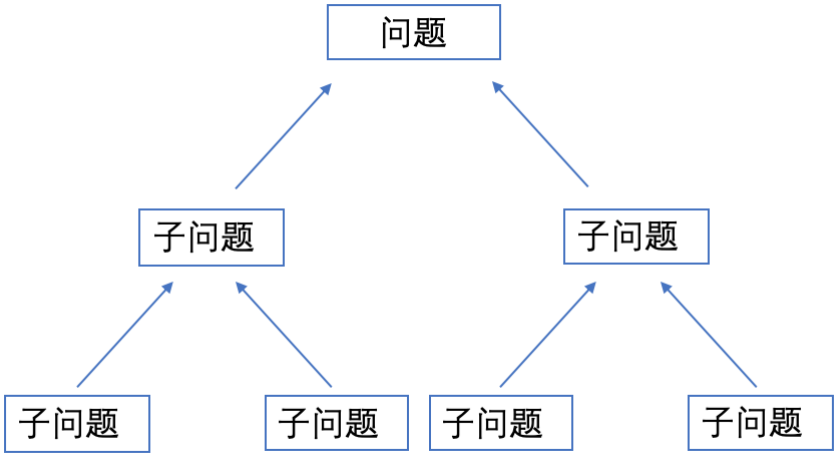
尽管开发一个 MapReduce 看起来很高深，感觉遥不可及。实际上，万变不离其宗，它的本质就是分治算法思想，分治算法。如何理解分治算法？为什么说 MapRedue 的本质就是分治算法呢？

主要思想

分治算法的主要思想是将原问题**递归地分成**若干个子问题，直到子问题**满足边界条件**，停止递归。将子问题逐个击破(一般是同种方法)，将已经解决的子问题合并，最后，算法会**层层合并**得到原问题的答案。

分治算法的步骤

- 分：**递归地**将问题**分解**为各个的子**问题**(性质相同的、相互独立的子问题)；
- 治：将这些规模更小的子问题**逐个击破**；
- 合：将已解决的子问题**逐层合并**，最终得出原问题的解；



分治法适用的情况

- 原问题的**计算复杂度**随着问题的规模的增加而增加。
- 原问题**能够被分解**成更小的子问题。
- 子问题的**结构和性质**与原问题一样，并且**相互独立**，子问题之间**不包含**公共的子子问题。
- 原问题分解出的子问题的解**可以合并**为该问题的解。

伪代码

```
def divide_conquer(problem, param1, param2,...):
    # 不断切分的终止条件
    if problem is None:
        print_result
        return
```

```
    # 准备数据
    data=prepare_data(problem)
    # 将大问题拆分为小问题
    subproblems=split_problem(problem, data)
    # 处理小问题，得到子结果
    subresult1=self.divide_conquer(subproblems[0],p1,...…)
    subresult2=self.divide_conquer(subproblems[1],p1,...)
    subresult3=self.divide_conquer(subproblems[2],p1,...…)
    # 对子结果进行合并 得到最终结果
    result=process_result(subresult1, subresult2, subresult3,...)
```

举个栗子

通过应用举例分析理解分治算法的原理其实并不难，但是要想灵活应用并在编程中体现这种思想中却并不容易。所以，这里这里用分治算法应用在排序的时候的一个栗子，加深对分治算法的理解。

相关概念：

- **有序度**：表示一组数据的有序程度
- **逆序度**：表示一组数据的无序程度

一般通过**计算有序对或者逆序对的个数**，来表示数据的有序度或逆序度。

假设我们有 `n` 个数据，我们期望数据从小到大排列，那完全有序的数据的有序度就是 $n(n-1)/2$，逆序度等于 0；相反，倒序排列的数据的有序度就是 0，逆序度是 $n(n-1)/2$。

Q：如何编程求出一组数据的有序对个数或者逆序对个数呢?

因为有序对个数和逆序对个数的求解方式是类似的，所以这里可以只思考逆序对（常接触的）个数的求解方法。

- 方法1
  - 拿数组里的每个数字跟它后面的数字比较，看有几个比它小的。
  - 把比它小的数字个数记作 `k`，通过这样的方式，把每个数字都考察一遍之后，然后对每个数字对应的 `k` 值求和
  - 最后得到的总和就是逆序对个数。
  - 这样操作的时间复杂度是$O(n^2)$（需要两层循环过滤）。那有没有更加高效的处理方法呢?这里尝试套用分治的思想来求数组 A 的逆序对个数。
- 方法2
  - 首先将数组分成前后两半 A1 和 A2，分别计算 A1 和 A2 的逆序对个数 K1 和 K2
  - 然后再计算 A1 与 A2 之间的逆序对个数 K3。那数组 A 的逆序对个数就等于 K1+K2+K3。
  - 注意使用分治算法其中一个要求是，**子问题合并的代价不能太大**，否则就起不了降低时间复杂度的效果了。
  - **如何快速计算出两个子问题 A1 与 A2 之间的逆序对个数呢？这里就要借助归并排序算法了。（这里先回顾一下归并排序思想）**如何借助归并排序算法来解决呢？归并排序中有一个非常关键的操作，就是将两个有序的小数组，合并成一个有序的数组。实际上，在这个合并的过程中，可以计算这两个小数组的逆序对个数了。每次合并操作，我们都计算逆序对个数，把这些计算出来的逆序对个数求和，就是这个数组的逆序对个数了。

算法应用

[Leetcode 169. 多数元素]

- 题目描述

  给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 [n/2] 的元素。

  你可以假设数组是非空的，并且给定的数组总是存在众数。

示例 1:

```
输入: [3,2,3]
输出: 3
```

示例 2:

```
输入: [2,2,1,1,1,2,2]
输出: 2
```

- 解题思路
  - 确定切分的终止条件

    直到所有的子问题都是长度为 1 的数组，停止切分。
  - 准备数据，将大问题切分为小问题

    递归地将原数组二分为左区间与右区间，直到最终的数组只剩下一个元素，将其返回
  - 处理子问题得到子结果，并合并
    - 长度为 1 的子数组中唯一的数显然是众数，直接返回即可。
    - 如果它们的众数相同，那么显然这一段区间的众数是它们相同的值。
    - 如果他们的众数不同，比较两个众数在整个区间内出现的次数来决定该区间的众数
- 代码

```python
class Solution(object):
    def majorityElement2(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        # 【不断切分的终止条件】
        if not nums:
            return None
        if len(nums) == 1:
            return nums[0]
        # 【准备数据，并将大问题拆分为小问题】
        left = self.majorityElement(nums[:len(nums)//2])
        right = self.majorityElement(nums[len(nums)//2:])
        # 【处理子问题，得到子结果】
        # 【对子结果进行合并 得到最终结果】
        if left == right:
            return left
        if nums.count(left) > nums.count(right):
            return left
        else:
            return right
```

Leetcode 53. 最大子序和

- 题目描述

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

示例:

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

- 解题思路
  - 确定切分的终止条件

    直到所有的子问题都是长度为 1 的数组，停止切分。
  - 准备数据，将大问题切分为小问题

    递归地将原数组二分为左区间与右区间，直到最终的数组只剩下一个元素，将其返回
  - 处理子问题得到子结果，并合并
    - 将数组切分为左右区间
      - 对与左区间：从右到左计算左边的最大子序和
      - 对与右区间：从左到右计算右边的最大子序和
    - 由于左右区间计算累加和的方向不一致，因此，左右区间直接合并相加之后就是整个区间的和
    - 最终返回左区间的元素、右区间的元素、以及整个区间(相对子问题)和的最大值
- 代码

```python
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        # 【确定不断切分的终止条件】
        n = len(nums)
        if n == 1:
            return nums[0]

        # 【准备数据，并将大问题拆分为小的问题】
        left = self.maxSubArray(nums[:len(nums)//2])
        right = self.maxSubArray(nums[len(nums)//2:])

        # 【处理小问题，得到子结果】
        #  从右到左计算左边的最大子序和
        max_l = nums[len(nums)//2 -1] # max_l为该数组的最右边的元素
        tmp = 0 # tmp用来记录连续子数组的和

        for i in range( len(nums)//2-1 , -1 , -1 ):# 从右到左遍历数组的元素
            tmp += nums[i]
            max_l = max(tmp ,max_l)

        # 从左到右计算右边的最大子序和
        max_r = nums[len(nums)//2]
        tmp = 0
        for i in range(len(nums)//2,len(nums)):
            tmp += nums[i]
            max_r = max(tmp,max_r)

        # 【对子结果进行合并 得到最终结果】
        # 返回三个中的最大值
        return max(left,right,max_l+ max_r)
```

Leetcode 50. Pow(x, n)

- 题目描述

  实现 `pow(x, n)`，即计算 `x` 的 `n` 次幂函数。

  示例 1:

  ```
  输入: 2.00000, 10
  输出: 1024.00000
  ```

  示例 2:

  ```
  输入: 2.10000, 3
  输出: 9.26100
  ```

  示例 3:

  ```
  输入: 2.00000, -2
  输出: 0.25000
  解释: 2-2 = 1/22 = 1/4 = 0.25
  ```

  说明:

  `-100.0 < x < 100.0`
  `n`是 32 位有符号整数，其数值范围是$[-2^{31}, 2^{31} - 1]$。

- 解题思路
  - 确定切分的终止条件

    对`n`不断除以2，并更新`n`，直到为0，终止切分
  - 准备数据，将大问题切分为小问题

    对`n`不断除以2，更新
  - 处理子问题得到子结果，并合并
    - `x`与自身相乘更新`x`
    - 如果`n%2 ==1`
      - 将`p`乘以`x`之后赋值给`p`(初始值为1)，返回`p`
  - 最终返回`p`

- 代码

```python
class Solution(object):
    def myPow(self, x, n):
        """
        :type x: float
        :type n: int
        :rtype: float
        """
        # 处理n为负的情况
        if n < 0 :
            x = 1/x
            n = -n
```

```
        # 【确定不断切分的终止条件】
        if n == 0 :
            return 1

        # 【准备数据，并将大问题拆分为小的问题】
        if n%2 ==1:
          # 【处理小问题，得到子结果】
          p = x * self.myPow(x,n-1)# 【对子结果进行合并 得到最终结果】
          return p
        return self.myPow(x*x,n/2)
```

分治作业

- 独立完成leetcode169、53、50

Task02：动态规划

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

主要思想

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

动态规划法仅仅解决每个子问题一次，具有天然剪枝的功能，从而减少计算量，

一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。

动态规划模板步骤：

- 确定动态规划状态
- 写出状态转移方程（画出状态转移表）
- 考虑初始化条件
- 考虑输出状态
- 考虑对时间，空间复杂度的优化（Bonus）

例题详解

接下来，我们对每个步骤进行详细的讲解，并给出不同题目中考虑的不同方式，争取让大家吃透动态规划的套路。
我们以最经典的动态规划题目——Leetcode 300.最长上升子序列 为例子。

题目描述

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

```
可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
你算法的时间复杂度应该为 O(n2) 。
```

解题思路

第一步：确定动态规划状态

- 是否存在状态转移?
- 什么样的状态比较好转移，找到对求解问题最方便的状态转移?

  想清楚到底是直接用需要求的，比如长度作为dp保存的变量还是用某个判断问题的状态比如是否是回文子串来作为方便求解的状态

该题目可以直接用一个一维数组`dp`来存储转移状态，`dp[i]`可以定义为以`nums[i]`这个数结尾的最长递增子序列的长度。举个实际例子，比如在`nums[10,9,2,5,3,7,101,18]`中，`dp[0]`表示数字10的最长递增子序列长度，那就是本身，所以为1，对于`dp[5]`对应的数字7来说的最长递增子序列是`[2,5,7]`（或者`[2,3,7]`）所以`dp[5]=3`。

第二步：写出一个好的状态转移方程

- 使用**数学归纳法**思维，写出准确的状态方程

  比如还是用刚刚那个`nums`数组，我们**思考一下是如何得到`dp[5]=3`的**：既然是递增的子序列，我们只要找到`nums[5]`(也就是7)前面那些结尾比7小的子序列，然后把7接到最后，就可以形成一个新的递增的子序列，也就是这个新的子序列也就是在找到的前面那些数后面加上7，相当长度加1。当然可能会找到很多不同的子序列，比如刚刚在上面列举的，但是只需要找到长度最长的作为`dp[5]`的值就行。总结来说就是比较当前`dp[i]`的长度和`dp[i]`对应产生新的子序列长度，我们用`j`来表示所有比`i`小的组数中的索引，可以用如下代码公式表示

```
for i in range(len(nums)):
    for j in range(i):
        if nums[i]>nums[j]:
            dp[i]=max(dp[i],dp[j]+1)
```

  Tips: 在实际问题中，如果不能很快得出这个递推公式，可以先尝试一步一步把前面几步写出来，如果还是不行很可能就是 dp 数组的定义不够恰当，需要回到第一步重新定义 dp 数组的含义；或者可能是 dp 数组存储的信息还不够，不足以推出下一步的答案，需要把 dp 数组扩大成二维数组甚至三维数组。

第三步：考虑初始条件

这是决定整个程序能否跑通的重要步骤，当我们确定好状态转移方程，我们就需要考虑一下边界值，边界值考虑主要又分为三个地方：

- dp数组整体的初始值
- dp数组(二维)i=0和j=0的地方
- dp存放状态的长度，是整个数组的长度还是数组长度加一，这点需要特别注意。

  对于本问题，子序列最少也是自己，所以长度为1，这样我们就可以方便的把所有的`dp`初始化为1，再考虑长度问题，由于`dp[i]`代表的是`nums[i]`的最长子序列长度，所以并不需要加一。所以用代码表示就是`dp=[1]*len(nums)`

  Tips：还有一点需要注意，找到一个方便的状态转移会使问题变得非常简单。举个例子，对于Leetcode120.三角形最小路径和问题，大多数人刚开始想到的应该是自顶向下的定义状态转移的思路，也就是从最上面的数开始定义状态转移，但是这题优化的解法则是通过定义由下到上的状态转移方程会大大简化问题，同样的对于Leetcode53.最大子序和也是采用从下往上遍历，保证每个子问题都是已经算好的。这个具体我们在题目中会讲到。

  这里额外总结几种Python常用的初始化方法：

  - 对于产生一个全为1，长度为n的数组：

```
1. dp=[1 for _ in range(n)]
2. dp=[1]*n
```

  - 对于产生一个全为0，长度为m，宽度为n的二维矩阵：

```
1. dp=[[0 for _ in range(n)] for _ in range(m)]
2. dp=[[0]*n for _ in range(m)]
```

第四步：考虑输出状态

主要有以下三种形式，对于具体问题，我们一定要想清楚到底dp数组里存储的是哪些值，最后我们需要的是数组中的哪些值：

- 返回dp数组中最后一个值作为输出，一般对应二维dp问题。
- 返回dp数组中最大的那个数字，一般对应记录最大值问题。
- 返回保存的最大值，一般是`Maxval=max(Maxval,dp[i])`这样的形式。

  Tips：这个公式必须是在满足递增的条件下，也就是`nums[i]>nums[j]`的时候才能成立，并不是`nums[i]`前面所有数字都满足这个条件的，理解好这个条件就很容易懂接下来在输出时候应该是`max(dp)`而不是`dp[-1]`，原因就是dp数组由于计算递增的子序列长度，所以dp数组里中间可能有值会是比最后遍历的数值大的情况，每次遍历`nums[j]`所对应的位置都是比`nums[i]`小的那个数。举个例子，比如`nums=[1,3,6,7,9,4,10,5,6]`,而最后`dp=[1,2,3,4,5,3,6,4,5]`。
  总结一下，最后的结果应该返回dp数组中值最大的数。

  最后加上考虑数组是否为空的判断条件，下面是该问题完整的代码：

```
def lengthOfLIS(self, nums: List[int]) -> int:
    if not nums:return 0  #判断边界条件
    dp=[1]*len(nums)      #初始化dp数组状态
    for i in range(len(nums)):
      for j in range(i):
        if nums[i]>nums[j]:   #根据题目所求得到状态转移方程
          dp[i]=max(dp[i],dp[j]+1)
          return max(dp)  #确定输出状态
```

第五步：考虑对时间，空间复杂度的优化（Bonus）

切入点：
我们看到，之前方法遍历dp列表需要$O(N)$，计算每个`dp[i]`需要$O(N)$的时间，所以总复杂度是$O(N^2)$

前面遍历dp列表的时间复杂度肯定无法降低了，但是我们看后面在每轮遍历`[0,i]`的`dp[i]`元素的时间复杂度可以考虑设计状态定义，使得整个dp为一个排序列表，这样我们自然想到了可以利用二分法来把时间复杂度降到了$O(NlogN)$。这里由于篇幅原因，如果大家感兴趣的话详细的解题步骤可以看好心人写的二分方法+动态规划详解

模板总结：

```
for i in range(len(nums)):
    for j in range(i):
       dp[i]=最值(dp[i],dp[j]+...)
```

对于子序列问题，很多也都是用这个模板来进行解题，比如Leetcode53.最大子序和。此外，其他情况的子序列问题可能需要二维的dp数组来记录状态，比如：Leetcode5. 最长回文子串（下面会讲到）、Leetcode1143. 最长公共子序列 (当涉及到两个字符串/数组时)
如果你觉得刚刚那题有点难的话，不如我们从简单一点的题目开始理解一下这类子序列问题。接下来所有题目我们都按照那五个步骤考虑

算法应用

[Leetcode 674.最长连续递增序列]

题目描述

给定一个未经排序的整数数组，找到最长且连续的的递增序列。

```
示例 1:
输入: [1,3,5,4,7]
输出: 3
解释: 最长连续递增序列是 [1,3,5], 长度为3。
尽管 [1,3,5,7] 也是升序的子序列, 但它不是连续的，因为5和7在原数组里被4隔开。
```

解题思路

这道题是不是一眼看过去和上题非常的像，没错了，这个题目最大的不同就是**连续**两个字，这样就让这个问题简单很多了，因为如果要求连续的话，那么就不需要和上题一样遍历两遍数组，只需要比较前后的值是不是符合递增的关系。

- **第一步：确定动态规划状态**
  对于这个问题，我们的状态**dp[i]也是以nums[i]这个数结尾的最长递增子序列的长度**
- **第二步：写出状态转移方程**
  这个问题，我们需要分两种情况考虑，第一种情况是如果遍历到的数`nums[i]`后面一个数不是比他大或者前一个数不是比他小，也就是所谓的不是连续的递增，那么这个数列最长连续递增序列就是他本身，也就是长度为1。
  第二种情况就是如果满足有递增序列，就意味着当前状态只和前一个状态有关，`dp[i]`只需要在前一个状态基础上加一就能得到当前最长连续递增序列的长度。总结起来，状态的转移方程可以写成
  `dp[i]=dp[i-1]+1`
- **第三步：考虑初始化条件**
  和上面最长子序列相似，这个题目的初始化状态就是一个一维的全为1的数组。
- **第四步：考虑输出状态**
  与上题相似，这个问题输出条件也是求dp数组中最大的数。
- **第五步：考虑是否可以优化**
  这个题目只需要一次遍历就能求出连续的序列，所以在时间上已经没有可以优化的余地了，空间上来看的话也是一维数组，并没有优化余地。

综上所述，可以很容易得到最后的代码：

```python
def findLengthOfLCIS(self, nums: List[int]) -> int:
    if not nums:return 0  #判断边界条件
    dp=[1]*len(nums)      #初始化dp数组状态
    #注意需要得到前一个数，所以从1开始遍历，否则会超出范围
    for i in range(1,len(nums)):
      if nums[i]>nums[i-1]:#根据题目所求得到状态转移方程
        dp[i]=dp[i-1]+1
        else:
          dp[i]=1
          return max(dp)  #确定输出状态
```

总结: 通过这个题目和例题的比较，我们需要理清子序列和子数组（连续序列）的差别，前者明显比后者要复杂一点，因为前者是不连续的序列，后者是连续的序列，从复杂度来看也很清楚能看到即使穷举子序列也比穷举子数组要复杂很多。

承接上面的话题，我们接下来继续来看一个子序列问题，这次是另外一种涉及二维状态的题目。

Leetcode5. 最长回文子串

题目描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

```
示例 1：

输入：“babad”
输出：“bab”
注意：“aba” 也是一个有效答案。

```

解题思路

- **第一步：确定动态规划状态**
  与上面两题不同的是，这个题目必须用二维的dp数组来记录状态，主要原因就是子串有回文的限制。用两个指针来记录子串的位置可以很好的实现子串的回文要求，又因为最后结果需要返回的是子串，这里不同于之前题目的用dp保存长度，我们必须找到具体哪个部分符合回文子串的要求。这里插一句，其实也有求回文子串长度的题目Leetcode516. 最长回文子序列,如果有兴趣可以看一下。这里我们定义`dp[i][j]`表示子串s从i到j是否为回文子串。
- **第二步：写出状态转移方程**
  首先我们需要知道符合回文的条件：
  - 字符串首尾两个字符必须相等，否则肯定不是回文。
  - 当字符串首尾两个字符相等时：如果子串是回文，整体就是回文，这里就有了动态规划的思想，出现了子问题；相反，如果子串不是回文，那么整体肯定不是。
    对于字符串`s,s[i,j]`的子串是`s[i+1,j-1]`，如果子串只有本身或者空串，那肯定是回文子串了，所以我们讨论的状态转移方程不是对于`j-1-(i+1)+1<2`的情况(整理得`j-i<3`)，当`s[i]`和`s[j]`相等并且`j-i<3`时，我们可以直接得出`dp[i][j]`是True。

    综上所述，可以得到状态转移方程

    ```
    if s[i]==s[j]:
        if j-i<3:
          dp[i][j]=True
        else:
          dp[i][j]=dp[i+1][j-1]
    ```

- **第三步：考虑初始化条件**
  我们需要建立一个二维的初始状态是False的来保存状态的数组来表示dp，又因为考虑只有一个字符的时候肯定是回文串，所以dp表格的对角线`dp[i][i]`肯定是True。
- **第四步：考虑输出状态**
  这里dp表示的是从`i`到`j`是否是回文子串，这样一来就告诉我们子串的起始位置和结束位置，但是由于我们需要找到最长的子串，所以我们优化一下可以只记录起始位置和当前长度（当然你要是喜欢记录终止位置和当前长度也是没问题的）

```
if dp[i][j]: #只要dp[i][j]成立就表示是回文子串，然后我们记录位置，返回有效答案
    cur_len=j-i+1
    if cur_len>max_len:
        max_len=cur_len
        start=i
```

- **第五步：考虑对时间，空间复杂度的优化**
  对于这个问题，时间和空间都可以进一步优化，对于空间方面的优化：这里采用一种叫中心扩散的方法来进行，而对于时间方面的优化，则是用了Manacher's Algorithm（马拉车算法）来进行优化。具体的实现可以参考动态规划、Manacher 算法

  这里给出比较容易理解的经典方法的代码：

```
 def longestPalindrome(self, s: str) -> str:
        length=len(s)
        if length<2:  #判断边界条件
            return s
        dp=[[False for _ in range(length)]for _ in range(length)] #定义dp状态矩阵
        #定义初试状态，这步其实可以省略
        # for i in range(length):
        #     dp[i][i]=True

        max_len=1
        start=0 #后续记录回文串初试位置
        for j in range(1,length):
            for i in range(j):
                #矩阵中逐个遍历
                if s[i]==s[j]:
                    if j-i<3:
                        dp[i][j]=True
                    else:
                        dp[i][j]=dp[i+1][j-1]
                if dp[i][j]: #记录位置，返回有效答案
                    cur_len=j-i+1
                    if cur_len>max_len:
                        max_len=cur_len
                        start=i
        return s[start:start+max_len]
```

总结：这个是一个二维dp的经典题目，需要注意的就是定义dp数组的状态是什么，这里不用长度作为dp值而用是否是回文子串这个状态来存储也是一个比较巧妙的方法，使得题目变得容易理解。

看了这么多套路相信你也对动态规划有点感觉了，这里再介绍一个求长度的子序列问题。

Leetcode516. 最长回文子序列

题目描述

给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。

```
示例 1:
输入:
"bbbab"
输出:
4
```

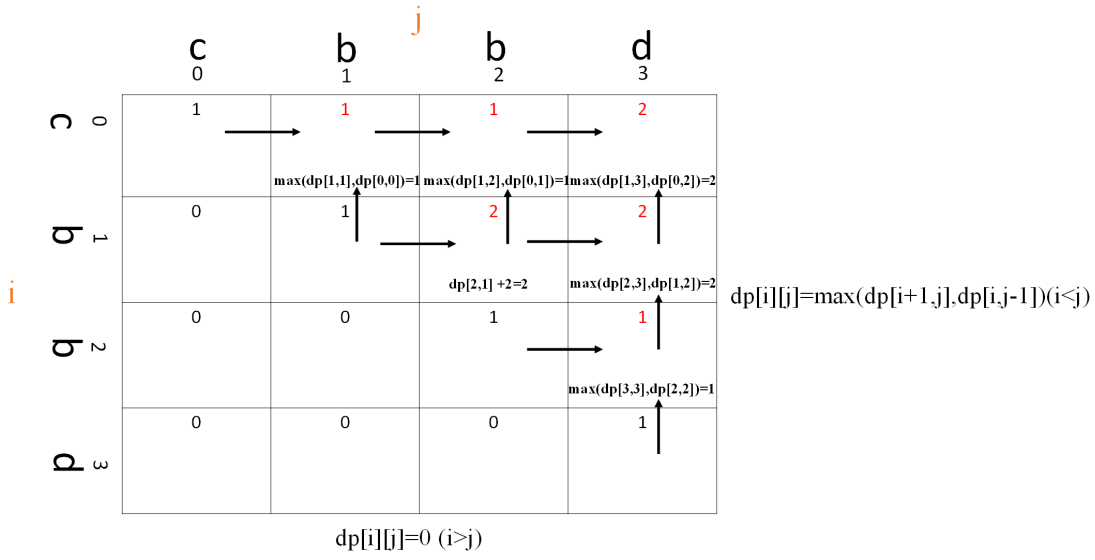
解题思路

这个问题和上面的例题也非常相似，直接套用动态规划套路也可以很快解决出来：

- **第一步：确定动态规划状态**
  这里求的是最长子串的长度，所以我们可以直接定义一个二维的`dp[i][j]`来表示字符串第`i`个字符到第`j`个字符的长度，子问题也就是每个子回文字符串的长度。
- **第二步：写出状态转移方程**
  我们先来具体分析一下整个题目状态转移的规律。对于`d[i][j]`,我们根据上题的分析依然可以看出，
  当`s[i]`和`s[j]`相等时，`s[i+1...j-1]`这个字符串加上2就是最长回文子序列；
  当`s[i]`和`s[j]`不相等时，就说明可能只有其中一个出现在s[i,j]的最长回文子序列中，我们只需要取`s[i-1,j-1]`加上`s[i]`或者`s[j]`的数值中较大的；
  综上所述，状态转移方程也就可以写成：

```python
if s[i]==s[j]:
    dp[i][j]= dp[i+1][j-1]+2
else:
    dp[i][j]=max(dp[i][j-1],dp[i+1][j])
```

  但是问题来了，具体我们应该怎么求每个状态的值呢？这里介绍一种利用状态转移表法写出状态转移方程，我们通过把`dp[i][j]`的状态转移直接画成一张二维表格，我们所要做的也就是往这张表中填充所有的状态，进而得到我们想要的结果。如下图：



  我们用字符串为"**cbbd**"作为输入来举例子，每次遍历就是求出右上角那些红色的值，通过上面的图我们会发现，按照一般的习惯都会先计算第一行的数值，但是当我们计算`dp[0,2]`的时候，我们会需要`dp[1,2]`，按照这个逻辑，我们就可以很容易发现遍历从下往上遍历会很方便计算。

- **第三步：考虑初始化条件**
  很明显看出来的当只有一个字符的时候，最长回文子序列就是1，所以可以得到`dp[i][j]=1(i=j)`
  接下来我们来看看

当`i>j`时，不符合题目要求，不存在子序列，所以直接初始化为0。
当`i<j`时，每次计算表中对应的值就会根据前一个状态的值来计算。

- **第四步：考虑输出状态**

  我们想要求最长子序列的时候，我们可以直接看出来`dp[0][-1]`是最大的值，直接返回这个值就是最后的答案。

- **第五步：考虑对时间，空间复杂度的优化**
  对于这个题目，同样可以考虑空间复杂度的优化，因为我们在计算`dp[i][j]`的时候，只用到左边和下边。如果改为用一维数组存储，那么左边和下边的信息也需要存在数组里，所以我们可以考虑在每次变化前用临时变量`tmp`记录会发生变化的左下边信息。所以状态转移方程就变成了：

```
if s[i] == s[j]:
    tmp, dp[j] = dp[j], tmp + 2
else:
    dp[j] =max(dp[j],dp[j-1])
```

这里给出基本版的实现代码，如果需要优化后的可以看空间压缩优化解法

```
def longestPalindromeSubseq(self, s: str) -> int:
        n=len(s)
        dp=[[0]*n for _ in range(n)]  #定义动态规划状态转移矩阵
        for i in range(n):  #    初始化对角线，单个字符子序列就是1
            dp[i][i]=1
        for i in range(n,-1,-1):  #从右下角开始往上遍历
            for j in range(i+1,n):
                if s[i]==s[j]:   #当两个字符相等时，直接子字符串加2
                    dp[i][j]= dp[i+1][j-1]+2
                else:            #不相等时，取某边最长的字符
                    dp[i][j]=max(dp[i][j-1],dp[i+1][j])
        return dp[0][-1]   #返回右上角位置的状态就是最长
```

总结：对于二维的数组的动态规划，采用了画状态转移表的方法来得到输出的状态，这种方法更加直观能看出状态转移的具体过程，同时也不容易出错。当然具体选择哪种方法则需要根据具体题目来确定，如果状态转移方程比较复杂的利用这种方法就能简化很多。

模板总结：

```
for i in range(len(nums)):
  for j in range(n):
    if s[i]==s[j]:
       dp[i][j]=dp[i][j]+...
    else:
       dp[i][j]=最值(...)
```

当然，动态规划除了解决子序列问题，也可以用来解决其他实际的问题，比如之前提到过的各种AI的经典算法，接下来我们来看一道动态规划的高频面试题，也是实际开发中很常用的。

Leetcode72. 编辑距离

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

题目描述

```
你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符
示例 1:

输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

解题思路

- **第一步：确定动态规划状态**

这个题目涉及到两个字符串，所以我们最先想到就是用两维数组来保存转移状态，定义`dp[i][j]`为字符串word1长度为`i`和字符串word2长度为`j`时，word1转化成word2所执行的最少操作次数的值。

- **第二步：写出状态转移方程**

  关于这个问题的状态转移方程其实很难想到，这里提供的一个方向就是试着举个例子，然后通过例子的变化记录每一步变化得到的最少次数，来找到删除，插入，替换操作的状态转移方程具体应该怎么写。
  我们采用从末尾开始遍历`word1`和`word2`，
  当`word1[i]`等于`word2[j]`时，说明两者完全一样，所以`i`和`j`指针可以任何操作都不做，用状态转移式子表示就是`dp[i][j]=dp[i-1][j-1]`，也就是前一个状态和当前状态是一样的。
  当`word1[i]`和`word2[j]`不相等时，就需要对三个操作进行递归了，这里就需要仔细思考状态转移方程的写法了。
  对于**插入**操作，当我们在word1中插入一个和word2一样的字符，那么word2就被匹配了，所以可以直接表示为`dp[i][j-1]+1`
  对于**删除**操作，直接表示为`dp[i-1][j]+1`
  对于**替换**操作，直接表示为`dp[i-1][j-1]+1`
  所以状态转移方程可以写成`min(dp[i][j-1]+1,dp[i-1][j]+1,dp[i-1][j-1]+1)`

- **第三步：考虑初始化条件**
  我们还是利用dp转移表法来找到状态转移的变化（读者可以自行画一张dp表，具体方法在求最长子序列中已经演示过了），这里我们用空字符串来额外加入到word1和word2中，这样的目的是方便记录每一步操作，例如如果其中一个是空字符串，那么另外一个字符至少的操作数都是1，就从1开始计数操作数，以后每一步都执行插入操作，也就是当`i=0`时，`dp[0][j]=j`,同理可得，如果另外一个是空字符串，则对当前字符串执行删除操作就可以了，也就是`dp[i][0]=i`。

- **第四步：考虑输出状态**
  在转移表中我们可以看到，可以从左上角一直遍历到左下角的值，所以最终的编辑距离就是最后一个状态的值，对应的就是`dp[-1][-1]`。

- **第五步：考虑对时间，空间复杂度的优化**
  和上题一样，这里由于`dp[i][j]`只和dp表中附近的三个状态（左边，右边和左上边）有关，所以同样可以进行压缩状态转移的空间存储，如果觉得有兴趣可以参考@Lyncien的解法,对于时间方面应该并没有可以优化的方法。

总结起来代码如下：

```python
        def minDistance(self, word1, word2):
```

```
        #m,n 表示两个字符串的长度
        m=len(word1)
        n=len(word2)
        #构建二维数组来存储子问题
        dp=[[0 for _ in range(n+1)] for _ in range(m+1)]
        #考虑边界条件，第一行和第一列的条件
        for i in range(n+1):
            dp[0][i]=i  #对于第一行，每次操作都是前一次操作基础上增加一个单位的操作
        for j in range(m+1):
            dp[j][0]=j #对于第一列也一样，所以应该是1,2,3,4,5...
        for i in range(1,m+1):  #对其他情况进行填充
            for j in range(1,n+1):
                if word1[i-1]==word2[j-1]: #当最后一个字符相等的时候，就不会产生任何操作代价，所以与dp[i-1][j-1]一样
                    dp[i][j]=dp[i-1][j-1]
                else:
                    dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1 #分别对应删除，添加和替换操作
        return dp[-1][-1] #返回最终状态就是所求最小的编辑距离
```

如果上面的题目看起来还是有点吃力的话，接下我们来来看轻松一点的题目，下面的题目和斐波那契数列求解类似，既可用迭代也可用动态规划做。

Leetcode198. 打家劫舍

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

题目描述

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

示例 1:

输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

解题思路

这个问题不复杂，其实利用一般的迭代可以直接解出来，但是这里讲动态规划，所以还是按照标准的套路来

- **第一步：确定动态规划状态**
  直接定义题目所求的偷窃的最高金额，所以`dp[i]`表示偷窃第`i`号房子能得到的最高金额。
- **第二步：写出状态转移方程**
  如果我们不考虑限制条件相邻两个房子不能抢，那么问题就很简单。想得到第`i`个房间偷窃到的最高金额的时候，我们会考虑子问题前`i-1`间的最高金额`dp[i-1]`，然后再加上当前房间的金额，所以最后可以表达为`dp[i]=dp[i-1]+nums[i]`。
  需要注意的是，这里限制了相邻两个房子是不能抢的，接下来我们就要考虑两种情况。
  如果抢了第i个房间，那么第`i-1`肯定是不能抢的，这个时候需要再往前一间，用第`i-2`间的金额加上当前房间的金额，得到的状态转移方程是`dp[i]=dp[i-2]+nums[i]`。
  如果没有抢第`i`个房间，那么肯定抢了第`i-1`间的金额，所以直接有`dp[i]=dp[i-1]`。

最后综合一下两种情况，就可以很快得到状态转移方程：`dp[i]=max(dp[i-2]+nums[i],dp[i-1])`

- **第三步：考虑初始化条件**
  初始化条件需要考虑第一个房子和第二个房子，之后的房子都可以按照规律直接求解，当我们只有一个房子的时候，自然只抢那间房子，当有两间房的时候，就抢金额较大的那间。综合起来就是`dp[0]=nums[0]，dp[1]=max(nums[0],nums[1])`。
- **第四步:考虑输出状态**
  直接返回状态转移数组的最后一个值就是所求的最大偷窃金额。
- **第五步：考虑对时间，空间复杂度的优化**
  时间复杂度为$O(N)$不能再优化了，空间复杂度方面如果用动态规划是不能优化，但是如果用迭代的方法只存储临时变量来记录每一步计算结果，这样可以降到$O(1)$。

这里给出动态规划版本的实现代码：

```python
    def rob(self, nums):

        if(not nums):   #特殊情况处理
            return 0
        if len(nums)==1:
            return nums[0]
        n=len(nums)
        dp=[0]*n    #初始化状态转移数组
        dp[0]=nums[0]  #第一个边界值处理
        dp[1]=max(nums[0],nums[1])#第二个边界值处理
        for i in range(2,n):
            dp[i]=max(dp[i-2]+nums[i],dp[i-1]) #状态转移方程
        return dp[-1]
```

Leetcode213. 打家劫舍 II

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

题目描述

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

示例 1:

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

解题思路

- **第一步：确定动态规划状态**
  直接定义题目所求的偷窃的最高金额，所以`dp[i]`表示偷窃第`i`号房子能得到的最高金额。
- **第二步：写出状态转移方程**

和上个题目类似，这个题目不一样的是现在所有房屋都围成一个圈，相比于上个问题又增加了一个限制，这样一来第一个房子和最后一个房子只能选择其中一个偷窃了。所有我们把这个问题拆分成两个问题：

- 偷窃了第一个房子，此时对应的是`nums[1:]`，得到最大的金额value是`v1`。
- 偷窃了最后一个房子，此时对应的是`nums[:n-1]`(其中n是所有房子的数量)，得到的最大金额value是`v2`。
  最后的结果就是取这两种情况的最大值，即`max(v1,v2)`。

每个子问题就和上题是一样的了，所以可以直接得到状态转移方程还是`dp[i]=max(dp[i-2]+nums[i],dp[i-1])`

- **第三步：考虑初始化条件**
  初始化一个房子和两个房子的情况就是`dp[0]=nums[0]，dp[1]=max(nums[0],nums[1])`。
- **第四步：考虑输出状态**
  直接返回状态转移数组的最后一个值就是所求的最大偷窃金额。
- **第五步：考虑对时间，空间复杂度的优化**

时间复杂度为$O(N)$不能再优化了，空间复杂度方面如果用动态规划是不能优化，但是如果用迭代的方法只存储临时变量来记录每一步计算结果，这样可以降到$O(1)$。

最后的代码实现：

```python
def rob(self, nums: List[int]) -> int:
        if not nums:
            return 0
        elif len(nums)<=2:
            return max(nums)
        def helper(nums):
            if len(nums)<=2:
                return max(nums)
            dp=[0]*len(nums)
            dp[0]=nums[0]
            dp[1]=max(nums[0],nums[1])
            for i in range(2,len(nums)):
                dp[i]=max(dp[i-1],dp[i-2]+nums[i])
            return dp[-1]
        return max(helper(nums[1:]),helper(nums[:-1]))
```

写在最后

动态规划是算法中比较难的类型，但是其实主要是掌握一种思维，有了这种思维，其实很难的问题都能一步一步解决好。最后再推荐一些比较优质的动态规划文章。

掌握动态规划，助你成为优秀的算法工程师

推荐MIT的动态规划练习资料，这份资料通过动态规划经典的问题让我们很清晰的了解到这个算法的魅力所在，对于新手入门动态规划是一个很不错的资料。Dynamic Programming Practice Problems

五分钟学算法的动态规划系列:
浅谈什么是动态规划以及相关的「股票」算法题
有了四步解题法模板，再也不害怕动态规划!
（进阶版）有了四步解题法模板，再也不害怕动态规划！

主要参考的Leetcode 优秀题解：
动态规划设计方法&&纸牌游戏讲解二分解法
动态规划、Manacher 算法
编辑距离面试题详解
打家劫舍 II（动态规划，结构化思路，清晰题解）

动态规划作业

- 独立完成leetcode 674、5、516、72、198、213

Task03：查找——查找表

考虑的基本数据结构

第一类： 查找有无--set

元素'a'是否存在，通常用set：集合

set只存储键，而不需要对应其相应的值。

set中的键不允许重复

第二类： 查找对应关系(键值对应)--dict

元素'a'出现了几次：dict-->字典

dict中的键不允许重复

第三类： 改变映射关系--map

通过将原有序列的关系映射统一表示为其他

算法应用

LeetCode 349 Intersection Of Two Arrays 1

题目描述

给定两个数组nums,求两个数组的公共元素。

```
如nums1 = [1,2,2,1],nums2 = [2,2]

结果为[2]
结果中每个元素只能出现一次
出现的顺序可以是任意的
```

分析实现

由于每个元素只出现一次，因此不需要关注每个元素出现的次数，用set的数据结构就可以了。记录元素的有和无。

把nums1记录为set，判断nums2的元素是否在set中，是的话，就放在一个公共的set中，最后公共的set就是我们要的结果。

代码如下：

```python
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        nums1 = set(nums1)
        return set([i for i in nums2 if i in nums1])
```

也可以通过set的内置方法来实现，直接求set的交集：

```python
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        set1 = set(nums1)
        set2 = set(nums2)
        return set2 & set1
```

LeetCode 350 Intersection Of Two Arrays 2

题目描述

给定两个数组nums,求两个数组的交集。

```
如nums1=[1,2,2,1],nums=[2,2

结果为[2,2]

出现的顺序可以是任意的
```

分析实现

元素出现的次数有用，那么对于存储次数就是有意义的，所以选择数据结构时，就应该选择dict的结构，通过字典的比较来判断；

记录每个元素的同时要记录这个元素的频次。

记录num1的字典，遍历nums2，比较nums1的字典的nums的key是否大于零，从而进行判断。

代码如下：

```python
class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
        from collections import Counter
        nums1_dict = Counter(nums1)
        res = []
        for num in nums2:
            if nums1_dict[num] > 0:
                # 说明找到了一个元素即在num1也在nums2
                res.append(num)
                nums1_dict[num] -= 1
        return res
```

LeetCode 242 Intersection Of Two Arrays 2

题目描述

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

```
示例1:

输入: s = "anagram", t = "nagaram"
输出: true

示例 2:

输入: s = "rat", t = "car"
输出: false
```

分析实现

判断异位词即判断变换位置后的字符串和原来是否相同，那么不仅需要存储元素，还需要记录元素的个数。可以选择dict的数据结构，将字符串s和t都用dict存储，而后直接比较两个dict是否相同。

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        from collections import Counter
        s = Counter(s)
        t = Counter(t)
        if s == t:
            return True
        else:
            return False
```

LeetCode 202 Happy number

题目描述

编写一个算法来判断一个数是不是“快乐数”。

一个“快乐数”定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是无限循环但始终变不到 1。如果可以变为 1，那么这个数就是快乐数。

```
示例:
输入: 19
输出: true
解释:
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1
```

分析实现

这道题目思路很明显，当n不等于1时就循环，每次循环时，将其最后一位到第一位的数依次平方求和，比较求和是否为1。

难点在于，什么时候跳出循环？

开始笔者的思路是，循环个100次，还没得出结果就false，但是小学在算无限循环小数时有一个特征，就是当除的数中，和之前历史的得到的数有重合时，这时就是无限循环小数。

那么这里也可以按此判断，因为只需要判断有或无，不需要记录次数，故用set的数据结构。每次对求和的数进行append，当新一次求和的值存在于set中时，就return false.

代码如下：

```
class Solution:
    def isHappy(self, n: int) -> bool:
        already = set()
        while n != 1:
            sum = 0
            while n > 0:
                # 取n的最后一位数
```

```
                tmp = n % 10
                sum += tmp ** 2
                # 将n的最后一位截掉
                n //= 10
            # 如果求的和在过程中出现过
            if sum in already:
                return False
            else:
                already.add(sum)
            n = sum
        return True
```

tips

```
#一般对多位数计算的套路是：
#循环从后向前取位数
while n >0 :
  #取最后一位：
  tmp = n % 10
  #再截掉最后一位：
  n = n // 10
```

LeetCode 290 Word Pattern

题目描述

给出一个模式(pattern)以及一个字符串，判断这个字符串是否符合模式

```
示例1:
输入: pattern = "abba",
str = "dog cat cat dog"
输出: true

示例 2:
输入:pattern = "abba",
str = "dog cat cat fish"
输出: false

示例 3:
输入: pattern = "aaaa", str = "dog cat cat dog"
输出: false

示例 4:
输入: pattern = "abba", str = "dog dog dog dog"
输出: false
```

分析实现

抓住变与不变，笔者开始的思路是选择了dict的数据结构，比较count值和dict对应的keys的个数是否相同，但是这样无法判断顺序的关系，如测试用例：'aba','cat cat dog'。

那么如何能**既考虑顺序**，也考虑**键值对应的关系**呢?

抓住变与不变，变的是键，但是不变的是各个字典中，对应的相同index下的值，如dict1[index] = dict2[index]，那么我们可以创建两个新的字典，遍历index对两个新的字典赋值，并比较value。

还有一个思路比较巧妙，既然不同，那么可以考虑怎么让它们相同，将原来的dict通过map映射为相同的key，再比较相同key的dict是否相同。

代码实现如下：

```
class Solution:
    def wordPattern(self,pattern, str):
        str = str.split()
        return list(map(pattern.index,pattern)) == list(map(str.index,str))
```

tips

1. 因为str是字符串，不是由单个字符组成，所以开始需要根据空格拆成字符list：

```
str = str.split()
```

2. 通过map将字典映射为index的list:

```
map(pattern.index, pattern)
```

3. map是通过hash存储的，不能直接进行比较，需要转换为list比较list

LeetCode 205 Isomorphic Strings

题目描述

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以被替换得到 t ，那么这两个字符串是同构的。

所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。两个字符不能映射到同一个字符上，但字符可以映射自己本身。

```
示例 1:
输入: s = "egg", t = "add"
输出: true

示例 2:
输入: s = "foo", t = "bar"
输出: false

示例 3:
输入: s = "paper", t = "title"
输出: true
```

分析实现

思路与上题一致，可以考虑通过建两个dict，比较怎样不同，也可以将不同转化为相同。

直接用上题的套路代码：

```
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        return list(map(s.index,s)) == list(map(t.index,t))

```

LeetCode 451 Sort Characters By Frequency

题目描述

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

```
示例 1:
输入:
"tree"
输出:
"eert"

示例 2:
输入:
"cccaaa"
输出:
"cccaaa"

示例 3:
输入:
"Aabb"
输出:
"bbAa"
```

分析实现

对于相同频次的字母，顺序任意，需要考虑大小写，返回的是字符串。

使用字典统计频率，对字典的value进行排序，最终根据key的字符串乘上value次数，组合在一起输出。

```python
class Solution:
    def frequencySort(self, s: str) -> str:
        from collections import Counter
        s_dict = Counter(s)
        # sorted返回的是列表元组
        s = sorted(s_dict.items(), key=lambda item:item[1], reverse = True)
        # 因为返回的是字符串
        res = ''
        for key, value in s:
            res += key * value
        return res
```

tips

1. 通过sorted的方法进行value排序，对字典排序后无法直接按照字典进行返回，返回的为列表元组：

```python
# 对value值由大到小排序
s = sorted(s_dict.items(), key=lambda item:item[1], reverse = True)

# 对key由小到大排序
s = sorted(s_dict.items(), key=lambda item:item[0])
```

2. 输出为字符串的情况下，可以由字符串直接进行拼接:

```
# 由key和value相乘进行拼接
's' * 5 + 'd'*2
```

查找表作业

[独立完成leetcode 349、350、242、202、290、205、451]

Task04：查找——对撞指针

算法应用

LeetCode 1 Two Sum

题目描述

```
给出一个整型数组nums，返回这个数组中两个数字的索引值i和j，使得nums[i] + nums[j]等于一个给定的target值，两个索引不能相等。

如：nums= [2,7,11,15],target=9
返回[0,1]
```

解题思路

需要考虑：

1. 开始数组是否有序；
2. 索引从0开始计算还是1开始计算？
3. 没有解该怎么办？
4. 有多个解怎么办？保证有唯一解。

分析实现

暴力法$O(n^2)$

时间复杂度为$O(n^2)$,第一遍遍历数组，第二遍遍历当前遍历值之后的元素，其和等于target则return。

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

排序+指针对撞$(O(n)+O(nlogn)=O(n))$

在数组篇的LeetCode 167题中，也遇到了找到两个数使得它们相加之和等于目标数，但那是对于排序的情况，因此也可以使用上述的思路来完成。

因为问题本身不是有序的，因此需要对原来的数组进行一次排序，排序后就可以用O(n)的指针对撞进行解决。

但是问题是，返回的是数字的索引，如果只是对数组的值进行排序，那么数组原来表示的索引的信息就会丢失，所以在排序前要进行些处理。

错误代码示例--只使用dict来进行保存：

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        record = dict()
        for index in range(len(nums)):
            record[nums[index]] = index
        nums.sort()
        l,r = 0,len(nums)-1
```

```
        while l < r:
            if nums[l] + nums[r] == target:
                return [record[nums[l]],record[nums[r]]]
            elif nums[l] + nums[r] < target:
                l += 1
            else:
                r -= 1
```

当遇到**相同的元素的索引**问题时，会不满足条件：

如：[3,3] 6

在排序前先使用一个额外的数组**拷贝**一份原来的数组，对于两个相同元素的索引问题，使用一个**bool型变量**辅助将两个索引都找到，总的时间复杂度为O(n)+O(nlogn) = O(nlogn)

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        record = dict()
        nums_copy = nums.copy()
        sameFlag = True;
        nums.sort()
        l,r = 0,len(nums)-1
        while l < r:
            if nums[l] + nums[r] == target:
                break
            elif nums[l] + nums[r] < target:
                l += 1
            else:
                r -= 1
        res = []
        for i in range(len(nums)):
            if nums_copy[i] == nums[l] and sameFlag:
                res.append(i)
                sameFlag = False
            elif nums_copy[i] == nums[r]:
                res.append(i)
        return res

```

小套路:如果只是对数组的值进行排序，那么数组原来表示的索引的信息就会丢失的情况，可以在排序前：

更加pythonic的实现

通过list(enumerate(nums))开始实现下标和值的绑定，不用专门的再copy加bool判断。

```
nums = list(enumerate(nums))
nums.sort(key = lambda x:x[1])
i,j = 0, len(nums)-1
while i < j:
    if nums[i][1] + nums[j][1] > target:
        j -= 1
    elif nums[i][1] + nums[j][1] < target:
        i += 1
    else:
        if nums[j][0] < nums[i][0]:
            nums[j],nums[i] = nums[i],nums[j]
        return num[i][0],nums[j][0]
```

拷贝数组 + bool型变量辅助

查找表--O(n)

遍历数组过程中，当遍历到元素v时，可以只看v前面的元素，是否含有target-v的元素存在。

1. 如果查找成功，就返回解；
2. 如果没有查找成功，就把v放在查找表中，继续查找下一个解。

即使v放在了之前的查找表中覆盖了v，也不影响当前v元素的查找。因为只需要找到两个元素，只需要找target-v的另一个元素即可。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        record = dict()
        for i in range(len(nums)):
            complement = target - nums[i]
            # 已经在之前的字典中找到这个值
            if record.get(complement) is not None:
                res = [i,record[complement]]
                return res
            record[nums[i]] = i
```

只进行一次循环，故时间复杂度O(n),空间复杂度为O(n)

补充思路：

通过enumerate来把索引和值进行绑定，进而对value进行sort，前后对撞指针进行返回。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        nums = list(enumerate(nums))
        # 根据value来排序
        nums.sort(key = lambda x:x[1])
        l,r = 0, len(nums)-1
        while l < r:
            if nums[l][1] + nums[r][1] == target:
                return nums[l][0],nums[r][0]
            elif nums[l][1] + nums[r][1] < target:
                l += 1
            else:
                r -= 1

```

LeetCode 15 3Sum

题目描述

给出一个整型数组，寻找其中的所有不同的三元组(a,b,c)，使得a+b+c=0

注意：答案中不可以包含重复的三元组。

如：nums = [-1, 0, 1, 2, -1, -4]，

结果为：[[-1, 0, 1],[-1, -1, 2]]

解题思路

1. 数组不是有序的；
2. 返回结果为全部解，多个解的顺序是否需要考虑？--不需要考虑顺序
3. 什么叫不同的三元组？索引不同即不同，还是值不同？--题目定义的是，值不同才为不同的三元组
4. 没有解时怎么返回？--空列表

分析实现

因为上篇中已经实现了Two Sum的问题，因此对于3Sum，首先想到的思路就是，开始固定一个k，然后在其后都当成two sum问题来进行解决，但是这样就ok了吗？

没有考虑重复元素导致错误

直接使用Two Sum问题中的查找表的解法，根据第一层遍历的i，将i之后的数组作为two sum问题进行解决。

```
class Solution:
    def threeSum(self, nums: [int]) -> [[int]]:
        res = []
        for i in range(len(nums)):
            num = 0 - nums[i]
            record = dict()
            for j in range(i + 1, len(nums)):
                complement = num - nums[j]
                # 已经在之前的字典中找到这个值
                if record.get(complement) is not None:
                    res_lis = [nums[i], nums[j], complement]
                    res.append(res_lis)
                record[nums[j]] = i
        return res

```

但是这样会导致一个错误，错误用例如下:

```
输入：
[-1,0,1,2,-1,-4]
输出：
[[-1,1,0],[-1,-1,2],[0,-1,1]]
预期结果：
[[-1,-1,2],[-1,0,1]]

```

代码在实现的过程中没有把第一次遍历的i的索引指向相同元素的情况排除掉，于是出现了当i指针后面位置的元素有和之前访问过的相同的值，于是重复遍历。

那么可以考虑，开始时对nums数组进行排序，排序后，当第一次遍历的指针k遇到下一个和前一个指向的值重复时，就将其跳过。为了方便计算，在第二层循环中，可以使用**对撞指针**的套路：

```
# 对撞指针套路
l,r = 0, len(nums)-1
while l < r:
    if nums[l] + nums[r] == target:
        return nums[l],nums[r]
    elif nums[l] + nums[r] < target:
        l += 1
    else:
        r -= 1

```

其中需要注意的是，在里层循环中，也要考虑重复值的情况，因此当值相等时，再次移动指针时，需要保证其指向的值和前一次指向的值不重复，因此可以：

```
# 对撞指针套路
l,r = 0, len(nums)-1
while l < r:
    sum = nums[i] + nums[l] + nums[r]
    if sum == target:
        res.append([nums[i],nums[l],nums[r])
        l += 1
        r -= 1
        while l < r and nums[l] == nums[l-1]: l += 1
        while l < r and nums[r] == nums[r+1]: r -= 1
    elif sum < target:
        l += 1
    else:
        r -= 1

```

再调整下遍历的范围，因为设了3个索引：i，l，r。边界情况下，r索引指向len-1, l指向len-2，索引i遍历的边界为len-3，故for循环是从0到len-2。

代码实现

```
class Solution:
    def threeSum(self, nums: [int]) -> [[int]]:
        nums.sort()
        res = []
        for i in range(len(nums)-2):
            # 因为是排序好的数组，如果最小的都大于0可以直接排除
            if nums[i] > 0: break
            # 排除i的重复值
            if i > 0 and nums[i] == nums[i-1]: continue
            l,r = i+1, len(nums)-1
            while l < r:
                sum = nums[i] + nums[l] + nums[r]
                if sum == 0:
                    res.append([nums[i],nums[l],nums[r]])
                    l += 1
                    r -= 1

```

```
                    while l < r and nums[l] == nums[l-1]: l += 1
                    while l < r and nums[r] == nums[r+1]: r -= 1
                elif sum < 0:
                    l += 1
                else:
                    r -= 1
        return res

```

小套路

1. 采用**for + while**的形式来处理三索引；
2. 当数组不是有序时需要注意，有序的特点在哪里，有序就可以用哪些方法解决？无序的话不便在哪里？
3. 对撞指针套路：

```
# 对撞指针套路
l,r = 0, len(nums)-1
while l < r:
    if nums[l] + nums[r] == target:
        return nums[l],nums[r]
    elif nums[l] + nums[r] < target:
        l += 1
    else:
        r -= 1

```

4. 处理重复值的套路：先转换为有序数组，再循环判断其与上一次值是否重复：

```
# 1.
for i in range(len(nums)):
    if i > 0 and nums[i] == nums[i-1]: continue
# 2.
while l < r:
    while l < r and nums[l] == nums[l-1]: l += 1

```

LeetCode 18 4Sum

题目描述

给出一个整形数组，寻找其中的所有不同的四元组(a,b,c,d)，使得a+b+c+d等于一个给定的数字target。

```
如：
nums = [1, 0, -1, 0, -2, 2]，target = 0

结果为：
[[-1,  0, 0, 1],[-2, -1, 1, 2],[-2,  0, 0, 2]]

```

题目分析

4Sum可以当作是3Sum问题的扩展，注意事项仍是一样的，同样是不能返回重复值得解。首先排序。接着从[0,len-1]遍历i，跳过i的重复元素，再在[i+1,len-1]中遍历j，得到i，j后，再选择首尾的l和r，通过对撞指针的思路，四数和大的话r--，小的话l++,相等的话纳入结果list，最后返回。

套用3Sum得代码，在其前加一层循环，对边界情况进行改动即可:

1. 原来3个是到len-2,现在外层循环是到len-3;
2. 在中间层得迭代中，当第二个遍历得值在第一个遍历得值之后且后项大于前项时，认定为重复;
3. 加些边界条件判断：当len小于4时，直接返回；当只有4个值且长度等于target时，直接返回本身即可。

代码实现如下:

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        res = []
        if len(nums) < 4: return res
        if len(nums) == 4 and sum(nums) == target:
            res.append(nums)
            return res
        for i in range(len(nums)-3):
            if i > 0 and nums[i] == nums[i-1]: continue
            for j in range(i+1,len(nums)-2):
                if j > i+1 and nums[j] == nums[j-1]: continue
                l,r = j+1, len(nums)-1
                while l < r:
                    sum_value = nums[i] + nums[j] + nums[l] + nums[r]
                    if sum_value == target:
                        res.append([nums[i],nums[j],nums[l],nums[r]])
                        l += 1
                        r -= 1
                        while l < r and nums[l] == nums[l-1]: l += 1
                        while l < r and nums[r] == nums[r+1]: r -= 1
                    elif sum_value < target:
                        l += 1
                    else:
                        r -= 1
        return res

```

还可以使用combinations(nums, 4)来对原数组中得4个元素全排列，在开始sort后，对排列得到得元素进行set去重。但单纯利用combinations实现会超时。

超出时间限制

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        from itertools import combinations
        res = []
        for i in combinations(nums, 4):
            if sum(i) == target:
                res.append(i)
        res = set(res)
        return res


```

LeetCode 16 3Sum Closest

题目描述

```
给出一个整形数组，寻找其中的三个元素a,b,c，使得a+b+c的值最接近另外一个给定的数字target。

如：给定数组 nums = [-1，2，1，-4]，和 target = 1.

与 target 最接近的三个数的和为 2. (-1 + 2 + 1 = 2).
```

分析实现

这道题也是2sum,3sum等题组中的，只不过变形的地方在于不是找相等的target，而是找最近的。

那么开始时可以随机设定一个三个数的和为结果值，在每次比较中，先判断三个数的和是否和target相等，如果相等直接返回和。如果不相等，则判断三个数的和与target的差是否小于这个结果值时，如果小于则进行则进行替换，并保存和的结果值。

伪代码

```
# 先排序
nums.sort()
# 随机选择一个和作为结果值
res = nums[0] + nums[1] + nums[2]
# 记录这个差值
diff = abs(nums[0]+nums[1]+nums[2]-target)
# 第一遍遍历
for i in range(len(nums)):
    # 标记好剩余元素的l和r
    l,r = i+1, len(nums-1)
    while l < r:
        if 后续的值等于target:
            return 三个数值得和
        else:
            if 差值小于diff:
                更新diff值
                更新res值
            if 和小于target:
                将l移动
            else:(开始已经排除了等于得情况，要判断和大于target)
                将r移动

```

3Sum问题两层遍历得套路代码

```
nums.sort()
res = []
for i in range(len(nums)-2):
    l,r = i+1, len(nums)-1
    while l < r:
        sum = nums[i] + nums[l] + nums[r]
        if sum == 0:
            res.append([nums[i],nums[l],nums[r]])
        elif sum < 0:
            l += 1
        else:
            r -= 1

```

代码实现

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        diff = abs(nums[0]+nums[1]+nums[2]-target)
        res = nums[0] + nums[1] + nums[2]
        for i in range(len(nums)):
            l,r = i+1,len(nums)-1
            t = target - nums[i]
            while l < r:
                if nums[l] + nums[r] == t:
                    return nums[i] + t
                else:
                    if abs(nums[l]+nums[r]-t) < diff:
                        diff = abs(nums[l]+nums[r]-t)
                        res = nums[i]+nums[l]+nums[r]
                    if nums[l]+nums[r] < t:
                        l += 1
                    else:
                        r -= 1
        return res

```

时间复杂度为O(n^2)，空间复杂度为O(1);

LeetCode 454 4SumII

题目描述

```
给出四个整形数组A,B,C,D,寻找有多少i,j,k,l的组合,使得A[i]+B[j]+C[k]+D[l]=0。其中,A,B,C,D中均含有相同的元素个数N，且0<=N<=500；

输入:

A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

输出:2
```

分析实现

这个问题同样是Sum类问题得变种，其将同一个数组的条件，变为了四个数组中，依然可以用查找表的思想来实现。

首先可以考虑把D数组中的元素都放入查找表，然后遍历前三个数组，判断target减去每个元素后的值是否在查找表中存在，存在的话，把结果值加1。那么查找表的数据结构选择用set还是dict？考虑到数组中可能存在重复的元素，而重复的元素属于不同的情况，因此用dict存储，最后的结果值加上dict相应key的value，代码如下：

$O(n^3)$代码

```
from collections import Counter
record = Counter()
# 先建立数组D的查找表
```

```
for i in range(len(D)):
    record[D[i]] += 1
res = 0
for i in range(len(A)):
    for j in range(len(B)):
        for k in range(len(C)):
            num_find = 0-A[i]-B[j]-C[k]
            if record.get(num_find) != None:
                res += record(num_find)
return res

```

但是对于题目中给出的数据规模：N<=500，如果N为500时，n^3的算法依然消耗很大，能否再进行优化呢?

根据之前的思路继续往前走，如果只遍历两个数组，那么就可以得到O(n^2)级别的算法，但是遍历两个数组，那么还剩下C和D两个数组，上面的值怎么放?

对于查找表问题而言，**很多时候到底要查找什么**，是解决的关键。对于C和D的数组，可以通过dict来记录其中和的个数，之后遍历结果在和中进行查找。代码如下：

$O(n^2)$级代码

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: List[int], D: List[int]) -> int:
        from collections import Counter
        record = Counter()
        for i in range(len(A)):
            for j in range(len(B)):
                record[A[i]+B[j]] += 1
        res = 0
        for i in range(len(C)):
            for j in range(len(D)):
                find_num = 0 - C[i] - D[j]
                if record.get(find_num) != None:
                    res += record[find_num]
        return res

```

再使用Pythonic的列表生成式和sum函数进行优化，如下：

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: List[int], D: List[int]) -> int:
        record = collections.Counter(a + b for a in A for b in B)
        return sum(record.get(- c - d, 0) for c in C for d in D)

```

LeetCode 49 Group Anagrams

题目描述

给出一个字符串数组，将其中所有可以通过颠倒字符顺序产生相同结果的单词进行分组。

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

说明:
所有输入均为小写字母。
不考虑答案输出的顺序。

```

分析实现

在之前LeetCode 242的问题中，对字符串t和s来判断，判断t是否是s的字母异位词。当时的方法是通过构建t和s的字典，比较字典是否相同来判断是否为异位词。

在刚开始解决这个问题时，我也局限于了这个思路，以为是通过移动指针，来依次比较两个字符串是否对应的字典相等，进而确定异位词列表，再把异位词列表添加到结果集res中。于是有：

错误思路

```python
nums = ["eat", "tea", "tan", "ate", "nat", "bat"]

from collections import Counter
cum = []
for i in range(len(nums)):
    l,r = i+1,len(nums)-1
    i_dict = Counter(nums[i])
    res = []
    if nums[i] not in cum:
        res.append(nums[i])
    while l < r:
        l_dict = Counter(nums[l])
        r_dict = Counter(nums[r])
        if i_dict == l_dict and l_dict == r_dict:
            res.append(nums[l],nums[r])
            l += 1
            r -= 1
        elif i_dict == l_dict:
            res.append(nums[l])
            l += 1
        elif i_dict == r_dict:
            res.append(nums[r])
            r -= 1
        else:
            l += 1
    print(res)
    cum.append(res)
.....................................

```

这时发现长长绵绵考虑不完，而且还要注意指针的条件，怎样遍历才能遍历所有的情况且判断列表是否相互间包含。。。

于是立即开始反思是否哪块考虑错了?回顾第一开始的选择数据结构，在dict和list中，自己错误的选择了list来当作数据结构，进而用指针移动来判断元素的情况。而**没有利用题目中不变的条件。**

题目的意思，对异位词的进行分组，同异位词的分为一组，那么考虑对这一组内什么是相同的，且这个相同的也能作为不同组的判断条件。

不同组的判断条件，就可以用数据结构dict中的key来代表，那么什么相同的适合当作key呢?

这时回顾下下LeetCode 242，当时是因为异位字符串中包含的**字符串的字母个数**都是相同的，故把字母当作key来进行判断是否为异位词。

但是对于本题，把每个字符串的字母dict，再当作字符串数组的dict的key，显然不太合适，那么对于异位词，还有什么是相同的?

显然，如果将字符串统一排序，**异位词排序后的字符串**，显然都是相同的。那么就可以把其当作key，把遍历的数组中的异位词当作value，对字典进行赋值，进而遍历字典的value，得到结果list。

需要注意的细节是，**字符串和list之间的转换**:

1. 默认构造字典需为list的字典;
2. 排序使用sorted()函数，而不用list.sort()方法，因为其不返回值;
3. 通过''.join(list)，将list转换为字符串;
4. 通过str.split(',')将字符串整个转换为list中的一项;

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        from collections import defaultdict
        strs_dict = defaultdict(list)
        res = []
        for str in strs:
            key = ''.join(sorted(list(str)))
            strs_dict[key] += str.split(',')
        for v in strs_dict.values():
            res.append(v)
        return res

```

再将能用列表生成式替换的地方替换掉,代码实现如下:

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        from collections import defaultdict
        strs_dict = defaultdict(list)
        for str in strs:
            key = ''.join(sorted(list(str)))
            strs_dict[key] += str.split(',')
        return [v for v in strs_dict.values()]

```

LeetCode 447 Number of Boomerangs

题目描述

给出一个平面上的n个点，寻找存在多少个由这些点构成的三元组(i,j,k)，**使得i,j两点的距离等于i,k两点的距离。**

其中n最多为500,且所有的点坐标的范围在[-10000,10000]之间。

```
输入：
[[0,0],[1,0],[2,0]]

输出：
2
解释：
两个结果为： [[1,0],[0,0],[2,0]] 和 [[1,0],[2,0],[0,0]]

```

分析实现

题目的要求是：使得i,j两点的距离等于i,k两点的距离，那么相当于是比较三个点之间距离的，那么开始的思路就是三层遍历，i从0到len，j从i+1到len，k从j+1到len，然后比较三个点的距离，相等则结果数加一。

显然这样的时间复杂度为O(n^3)，对于这道题目，能否用查找表的思路进行解决优化?

查找表

之前的查找表问题，大多是通过**构建一个查找表**，而避免了在查找中再内层嵌套循环，从而降低了时间复杂度。那么可以考虑在这道题中，可以通过查找表进行代替哪两层循环。

当i,j两点距离等于i,k时，用查找表的思路，等价于：对距离key(i,j或i,k的距离)，其值value(个数)为2。

那么就可以做一个查找表，用来查找相同距离key的个数value是多少。遍历每一个节点i，扫描得到其他点到节点i的距离，在查找表中，对应的键就是距离的值，对应的值就是距离值得个数。

在拿到对于元素i的距离查找表后，接下来就是排列选择问题了：

1. 如果当距离为x的值有2个时，那么选择j,k的可能情况有：第一次选择有2种，第二次选择有1种，为2*1；
2. 如果当距离为x的值有3个时，那么选择j,k的可能的情况有：第一次选择有3种，第二次选择有2种，为3*2;
3. 那么当距离为x的值有n个时，选择j,k的可能情况有：第一次选择有n种，第二次选择有n-1种。

距离

对于距离值的求算，按照欧式距离的方法进行求算的话，容易产生浮点数，可以将根号去掉，用差的平方和来进行比较距离。

实现代码如下：

```python
class Solution:
    def numberOfBoomerangs(self, points: List[List[int]]) -> int:
        res = 0
        from collections import Counter
        for i in points:
            record = Counter()
            for j in points:
                if i != j:
                    record[self.dis(i,j)] += 1
            for k,v in record.items():
                res += v*(v-1)
        return res
    def dis(self,point1,point2):
        return (point1[0]-point2[0]) ** 2 + (point1[1]-point2[1]) ** 2

```

优化

对实现的代码进行优化：

1. 将for循环遍历改为列表生成式;
2. 对sum+=的操作，考虑使用sum函数。
3. 对不同的函数使用闭包的方式内嵌;

```
class Solution:
    def numberOfBoomerangs(self, points: List[List[int]]) -> int:
        from collections import Counter
        def f(x1, y1):
            # 对一个i下j,k的距离值求和
            d = Counter((x2 - x1) ** 2 + (y2 - y1) ** 2 for x2, y2 in
points)
            return sum(t * (t-1) for t in d.values())
        # 对每个i的距离进行求和
        return sum(f(x1, y1) for x1, y1 in points)

```

LeetCode 149 Max Points on a Line

题目描述

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

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

示例 2:
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4

```

分析实现

本道题目的要求是：看有多少个点在同一条直线上，那么判断点是否在一条直线上，其实就等价于判断i,j两点的斜率是否等于i,k两点的斜率。

回顾上道447题目中的要求：使得i,j两点的距离等于i,k两点的距离，那么在这里，直接考虑使用查找表实现，即**查找相同斜率key的个数value是多少。**

在上个问题中，i和j，j和i算是两种不同的情况，但是这道题目中，这是属于相同的两个点，
因此在对遍历每个i,查找与i相同斜率的点时，不能再对结果数res++，而应该取查找表中的最大值。如果有两个斜率相同时，返回的应该是3个点，故返回的是结果数+1。

查找表实现套路如下：

```
class Solution:
    def maxPoints(self,points):
        res = 0
        from collections import defaultdict
        for i in range(len(points)):
            record = defaultdict(int)
            for j in range(len(points)):
```

```
                if i != j:
                    record[self.get_Slope(points,i,j)] += 1
            for v in record.values():
                res = max(res, v)
        return res + 1
    def get_Slope(self,points,i,j):
        return (points[i][0] - points[j][0]) / (points[i][1] - points[j]
[1])

```

但是这样会出现一个问题，即斜率的求算中，有时会出现直线为垂直的情况，故需要对返回的结果进行判断，如果分母为0，则返回inf，如下：

```
def get_Slope(self,points,i,j):
    if points[i][1] - points[j][1] == 0:
        return float('Inf')
    else:
        return (points[i][0] - points[j][0]) / (points[i][1] - points[j][1])

```

再次提交，发现对于空列表的测试用例会判断错误，于是对边界情况进行判断，如果初始长度小于等于1,则直接返回len：

```
if len(points) <= 1:
    return len(points)

```

再次提交，对于相同元素的测试用例会出现错误，回想刚才的过程，当有相同元素时，题目的要求是算作两个不同的点，但是在程序运行时，会将其考虑为相同的点，return回了inf。但在实际运行时，需要对相同元素的情况单独考虑。

于是可以设定samepoint值，遍历时判断，如果相同时，same值++,最后取v+same的值作为结果数。

考虑到如果全是相同值，那么这时dict中的record为空，也要将same值当作结果数返回，代码实现如下：

```
class Solution:
    def maxPoints(self,points):
        if len(points) <= 1:
            return len(points)
        res = 0
        from collections import defaultdict
        for i in range(len(points)):
            record = defaultdict(int)
            samepoint = 0
            for j in range(len(points)):
                if points[i][0] == points[j][0] and points[i][1] ==
points[j][1]:
                    samepoint += 1
                else:
                    record[self.get_Slope(points,i,j)] += 1
            for v in record.values():
                res = max(res, v+samepoint)
            res = max(res, samepoint)
        return res
    def get_Slope(self,points,i,j):
```

```
        if points[i][1] - points[j][1] == 0:
            return float('Inf')
        else:
            return (points[i][0] - points[j][0]) / (points[i][1] - points[j][1])

```

时间复杂度为O(n^2)，空间复杂度为O(n)

总结

遍历时多用索引，而不要直接用值进行遍历；

对撞指针作业

独立完成leetcode 1、15、18、16、454、49、447、149

Task05：查找——滑动数组&二分查找

滑动数组算法应用

LeetCode 219 Contains Dupliccate II

题目描述

给出一个整形数组nums和一个整数k，是否存在索引i和j，使得nums[i]==nums[j]，且i和j之间的差不超过k。

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

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

```

分析实现

翻译下这个题目：在这个数组中，如果有两个元素索引i和j，它们对应的元素是相等的，且索引j-i是小于等于k，那么就返回True，否则返回False。

因为对于这道题目可以用暴力解法双层循环，即：

```
for i in range(len(nums)):
    for j in range(i+1,len(nums)):
        if i == j:
            return True
return False

```

故这道题目可以考虑使用滑动数组来解决：

固定滑动数组的长度为K+1，当这个滑动数组内如果能找到两个元素的值相等，就可以保证两个元素的索引的差是小于等于k的。如果当前的滑动数组中没有元素相同，就右移滑动数组的右边界r,同时将左边界l右移。查看r++的元素是否在l右移过后的数组里，如果不在就将其添加数组，在的话返回true表示两元素相等。

因为滑动数组中的元素是不同的，考虑用set作为数据结构：

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        record = set()
        for i in range(len(nums)):
            if nums[i] in record:
                return True
            record.add(nums[i])
            if len(record) == k+1:
                record.remove(nums[i-k])
        return False

```

时间复杂度为O(n)，空间复杂度为O(n)

LeetCode 220 Contains Duplicate III

题目描述

给定一个整数数组，判断数组中是否有两个不同的索引 i 和 j，使得nums [i] 和nums [j]的差的绝对值最大为 t，并且 i 和 j 之间的差的绝对值最大为 ḳ。

```
示例 1:
输入: nums = [1,2,3,1], k = 3, t = 0
输出: true

示例 2:
输入: nums = [1,0,1,1], k = 1, t = 2
输出: true

示例 3:
输入: nums = [1,5,9,1,5,9], k = 2, t = 3
输出: false
```

分析实现

相比较上一个问题，这个问题多了一个限定条件，条件不仅索引差限定k，数值差也限定为了t。

将索引的差值固定，于是问题和上道一样，同样转化为了固定长度K+1的滑动窗口内，是否存在两个值的差距不超过 t，考虑使用**滑动窗口**的思想来解决。

在遍历的过程中，目的是要在“已经出现、但还未滑出滑动窗口”的所有数中查找，是否有一个数与滑动数组中的数的**差的绝对值**最大为 t。对于差的绝对值最大为t，实际上等价于所要找的这个元素v的范围是在v-t到v+t之间，即查找“滑动数组”中的元素有没有范围内的数存在。

因为只需证明是否存在即可，这时判断的逻辑是：如果在滑动数组**查找比v-t大的最小的元素**,如果这个元素小于等于v+t,即可以证明存在[v-t,v+t]。

那么实现过程其实和上题是一致的，只是上题中的判断条件是**在查找表中找到和nums[i]相同的元素**，而这题中的判断条件是**查找比v-t大的最小的元素，判断其小于等于v+t**，下面是实现的框架：

```python
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        record = set()
        for i in range(len(nums)):
            if 查找的比v-t大的最小的元素 <= v+t:
                return True
            record.add(nums[i])
            if len(record) == k+1:
                record.remove(nums[i-k])
        return False

```

接下来考虑，如何查找比v-t大的最小的元素呢?

【注：C++中有lower_bound(v-t)的实现，py需要自己写函数】

当然首先考虑可以通过O(n)的解法来完成，如下：

```
def lower_bound(self,array,v):
    array = list(array)
    for i in range(len(array)):
        if array[i] >= v:
            return i
    return -1

```

但是滑动数组作为set，是有序的数组。对于有序的数组，应该第一反应就是**二分查找**，于是考虑二分查找实现，查找比v-t大的最小的元素：

```
def lower_bound(self, nums, target):
    low, high = 0, len(nums)-1
    while low<high:
        mid = int((low+high)/2)
        if nums[mid] < target:
            low = mid+1
        else:
            high = mid
    return low if nums[low] >= target else -1

```

整体代码实现如下，时间复杂度为O(nlogn),空间复杂度为O(n):

```
class Solution:
    def containsNearbyAlmostDuplicate(self, nums, k, t) -> bool:
        record = set()
        for i in range(len(nums)):
            if len(record) != 0:
                rec = list(record)
                find_index = self.lower_bound(rec,nums[i]-t)
                if find_index != -1 and rec[find_index] <= nums[i] + t:
                    return True
            record.add(nums[i])
            if len(record) == k + 1:
                record.remove(nums[i - k])
        return False
    def lower_bound(self, nums, target):
        low, high = 0, len(nums)-1
        while low<high:
            mid = int((low+high)/2)
            if nums[mid] < target:
                low = mid+1
            else:
                high = mid
        return low if nums[low] >= target else -1

```

当然。。。在和小伙伴一起刷的时候，这样写的$O(n^2)$的结果会比上面要高，讨论的原因应该是上面的步骤存在着大量set和list的转换导致，对于py，仍旧是考虑算法思想实现为主，下面是$O(n^2)$的代码：

```python
class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:
        if t == 0 and len(nums) == len(set(nums)):
            return False
        for i in range(len(nums)):
            for j in range(1,k+1):
                if i+j >= len(nums): break
                if abs(nums[i+j]-nums[i]) <= t: return True
        return False

```

小套路

二分查找实现，查找比v-t大的最小的元素：

```python
def lower_bound(self, nums, target):
    low, high = 0, len(nums)-1
    while low<high:
        mid = int((low+high)/2)
        if nums[mid] < target:
            low = mid+1
        else:
            high = mid
    return low if nums[low] >= target else -1

```

二分查找实现，查找比v-t大的最小的元素：

```python
def upper_bound(nums, target):
    low, high = 0, len(nums)-1
    while low<high:
        mid=(low+high)/2
        if nums[mid]<=target:
            low = mid+1
        else:#>
            high = mid
            pos = high
    if nums[low]>target:
        pos = low
    return -1

```

二分查找代码模板

总体来说二分查找是比较简单的算法，网上看到的写法也很多，掌握一种就可以了。
以下是我的写法，参考C++标准库里的写法。这种写法比较好的点在于：

- 1.即使区间为空、答案不存在、有重复元素、搜索开/闭区间的上/下界也同样适用
- 2.+-1 的位置调整只出现了一次，而且最后返回lo还是hi都是对的，无需纠结

```python
class Solution:
    def firstBadVersion(self, arr):
        # 第一点
        lo, hi = 0, len(arr)-1
        while lo < hi:
            # 第二点
            mid = (lo+hi) // 2
            # 第三点
            if f(x):
                lo = mid + 1
            else:
                hi = mid
        return lo
```

解释:

- 第一点：lo和hi分别对应搜索的上界和下界，但不一定为0和arr最后一个元素的下标。
- 第二点：因为Python没有溢出，int型不够了会自动改成long int型，所以无需担心。如果再苛求一点，可以把这一行改成

```python
mid = lo + (hi-lo) // 2
# 之所以 //2 这部分不用位运算 >> 1 是因为会自动优化，效率不会提升
```

- 第三点：
  比较重要的就是这个f(x)，在带入模板的情况下，写对函数就完了。

那么我们一步一步地揭开二分查找的神秘面纱，首先来一道简单的题。

二分查找算法应用

LeetCode 35. Search Insert Position

给定排序数组和目标值，如果找到目标，则返回索引。如果不是，则返回按顺序插入索引的位置的索引。 您可以假设数组中没有重复项。

Example

```
Example 1:
Input: [1,3,5,6], 5
Output: 2

Example 2:
Input: [1,3,5,6], 2
Output: 1

Example 3:
Input: [1,3,5,6], 7
Output: 4

Example 4:
Input: [1,3,5,6], 0
Output: 0
```

分析： 这里要注意的点是 high 要设置为 len(nums) 的原因是像第三个例子会超出数组的最大值，所以要让 lo 能到 这个下标。

```python
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        lo, hi = 0, len(nums)
        while lo < hi:
            mid = (lo + hi) // 2
            if nums[mid] < target:
                lo = mid + 1
            else:
                hi = mid
        return lo
```

LeetCode540. Single Element in a Sorted Array

您将获得一个仅由整数组成的排序数组，其中每个元素精确出现两次，但一个元素仅出现一次。 找到只出现一次的单个元素。

Example

```
Example 1:


0.: [1,1,2,3,3,4,4,8,8]
Output: 2

Example 2:

Input: [3,3,7,7,10,11,11]
Output: 10
```

分析： 异或的巧妙应用！如果mid是偶数，那么和1异或的话，那么得到的是mid+1，如果mid是奇数，得到的是mid-1。如果相等的话，那么唯一的元素还在这之后，往后找就可以了。

```python
class Solution:
    def singleNonDuplicate(self, nums):
        lo, hi = 0, len(nums) - 1
        while lo < hi:
            mid = (lo + hi) // 2
            if nums[mid] == nums[mid ^ 1]:
                lo = mid + 1
            else:
                hi = mid
        return nums[lo]
```

是不是还挺简单哈哈，那我们来道HARD难度的题!

LeetCode 410. Split Array Largest Sum

给定一个由非负整数和整数m组成的数组，您可以将该数组拆分为m个非空连续子数组。编写算法以最小化这m个子数组中的最大和。

Example

```
Input:
nums = [7,2,5,10,8]
m = 2

Output:
18

Explanation:
There are four ways to split nums into two subarrays.
The best way is to split it into [7,2,5] and [10,8],
where the largest sum among the two subarrays is only 18.
```

分析：

- 这其实就是二分查找里的按值二分了，可以看出这里的元素就无序了。但是我们的目标是找到一个合适的最小和，换个角度理解我们要找的值在最小值max(nums)和sum(nums)内，而这两个值中间是连续的。是不是有点难理解，那么看代码吧
- 辅助函数的作用是判断当前的“最小和”的情况下，区间数是多少，来和m判断
- 这里的下界是数组的最大值是因为如果比最大值小那么一个区间就装不下，数组的上界是数组和因为区间最少是一个，没必要扩大搜索的范围

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

        def helper(mid):
            res = tmp = 0
            for num in nums:
                if tmp + num <= mid:
                    tmp += num
                else:
                    res += 1
                    tmp = num
            return res + 1

        lo, hi = max(nums), sum(nums)
        while lo < hi:
            mid = (lo + hi) // 2
            if helper(mid) > m:
                lo = mid + 1
            else:
                hi = mid
        return lo
```

滑动数组&二分查找作业

- 独立完成35、540、410
- 独立完成219、220