



- 开始刷题前
  - 1 星球使用方法
    - 1.1 知识星球有 web 版和 app 版
    - 1.2 基本功能
    - 1.3 点赞和评论
    - 1.4 精华主题和文件主题
    - 1.5 置顶主题
  - 2 程序员还需要学算法吗？
  - 3 从零学算法大纲
- Day1 冒泡排序
  - 1.1 学习算法需要什么基础
  - 1.2 算法入门参考资料
  - 1.3 算法进阶参考资料
  - 1.4 算法入门最基本思路
  - 1.5 今日打卡题
  - 1.6 参考思路
- Day2 选择排序
  - 2.1 精选回答
  - 2.2 作业点评
- Day3 算法基本要素
  - 3.1 精选回答
- Day4 Hailstone
  - 4.1 精选回答
  - 4.2 Collatz 猜想
- Day5 数组中插入元素
  - 5.1 精彩回答
- Day6 求中心索引
  - 6.1 精彩回答
  - 6.2 不高效解
  - 6.3 算法分析总结
- Day7 如何培养算法思维
  - 7.1 经历描述
  - 7.2 经验总结
  - 7.3 追求目标
  - 7.4 精选回答

- Day8 两数之和
  - 8.1 精选回答
  - 8.2 分析总结
  - 8.3 不高效解 1
  - 8.4 不高效解 2
- Day9 什么是哈希表
  - 9.1 精彩回答
- Day10 哈希表设计艺术
  - 10.1 什么是哈希表？
  - 10.2 哈希表原理
  - 10.3 哈希集
  - 10.4 哈希映射
  - 10.5 设计键
  - 10.6 精选答案
    - 第一种设计键的方法
    - 第二种设计键的方法
    - 第三种设计键的方法
- Day11 宝石和石头
  - 方法 1 暴力破解：
  - 方法 2 哈希表
- Day12 删除链表节点
  - 12.1 链表基础
  - 12.2 作业分析
  - 12.3 删除链表的节点
- Day13 访问链表第 $i$ 个节点
  - 13.1 分析总结
- Day14 反转单链表
  - 14.1 反转单链表
    - 1 迭代法
    - 2 递归法
    - 3 尾递归
- Day15 程序员为什么学算法
  - 为什么要学算法
- Day17 算法好坏度量大O记号
  - 1 数学定义

- 2 大 O 记号
- 3 基本原则
- Day18 二分查找
  - 迭代二分查找
  - 递归二分查找
  - 更多演示动画
- Day19 合并两个有序数
  - 分析过程
- Day20 归并排序算法
  - 分析过程
- Day21 21天刷题总结
  - 刷题总结
- Day22 递归相反顺序打印字符串
  - 递归方法一
- Day23 递归两两交换链表节点
  - 分析过程
- Day24 递归生成杨辉三角
  - 分析过程
- Day25 递归求斐波那契数列前 N 项
  - 分析过程
  - 精选回答
- Day26 递归的时间复杂度分析
  - 分析过程
- Day28 0-1 背包问题
  - 分析过程
  - 求解
- Day29 复习 0-1 背包问题
- Day30 理解递归的特例：尾递归
  - 分析过程
- Day31 递归快速幂算法： $\text{Pow}(x,n)$ 
  - 分析过程
- Day32 递推专题总结
- Day33 合并两个有序链表
- Day34 寻找有序数组元素插入位置
- Day 35 寻找有序数组元素插入位置，若重复靠后插入

- Day36 合并两个数组
- Day37 移动零
  - 题目
  - 分析
  - 求解代码
- Day38 最大连续1的个数
- Day39 找到重复数
  - 1 题目
  - 2 分析
- 2.1 二分查找

# 开始刷题前

本pdf来自《算法刷题日记》知识星球，经过振哥精心整理，版权完全归《算法刷题日记》星球和振哥所有，严禁将此pdf分享到其他地方，仅用作星球里的成员学习使用。



# 1 星球使用方法

## 1.1 知识星球有 web 版和 app 版

## 1.2 基本功能

接下来以 web 版使用举例，点击发表主题，完成后，可以选择一个标签，一对 #，比如 #算法刷题#，提交后，发表的主题最下面会有一个“算法刷题”的标签。



zhenguo

前天 18:10

...

算法和数据结构 学习资料推荐

zheng

推荐三本不错的算法与数据结构书籍，适合入门和复习

1 数据结构与算法图解

2 算法图解

3 我的第一本算法书

星友们太积极了，等我晚上到家时，你们已经把这三本书的电子版上传上来了啊，很感动，谢谢你们，彼此帮助的感觉真棒。

#算法刷题#

收起



算法刷题

## 1.3 点赞和评论

为其他星友的主题点赞和评论



Leven

2020/5/22

#冒泡排序#

优化点：

- 1、添加有序标记 (flag)，当没有元素交换时跳出循环
- 2、记录有序/无序边界，已有序的元素不需要再被进行比较，因此每轮需比较的数据长度会减少

```
knowledgePlanet > BubbleSort.py > ...
1 def bubble_sort(our_list):
2     n = len(our_list)
3     lastExchangeIndex = 0           #记录最后一次交换元素的位置
4     sortBorder = n-1               #无序数列边界
5     for i in range(n):
6         flag = True                #有序标记，每轮开始初始值均为True
7         for j in range(0,sortBorder):
8             if our_list[j] > our_list[j+1]:
9                 our_list[j],our_list[j+1] = our_list[j+1],our_list[j]
10                flag = False           #有元素交换，则将标记置为False
11                lastExchangeIndex = j
12            sortBorder = lastExchangeIndex
13            if flag:
14                break
15    return our_list
```

| 查看作业题目 >



等30人赞过

## 1.4 精华主题和文件主题

精华主题：

全部主题 ▾

全部主题

精华主题

文件主题

问答主题

只看星主

等你回答

| 从Day1~Day现在，作业帖和总结帖：【星球如何使用】新手必看...

| 什么是哈希表？今日作业题详见下方链接，在这篇文章里首先总结...

| LeetCode 经典题：两数之和 今日作业题详见下方链接，在这篇文章...



| 是哈希表？）打卡：

| 表是根据键值（key value）而直接进行访问的数据结构。它通过  
| 将键映射到一个位置来访问记录，以加快查找的速度。具体映射过程是：

文件主题（都是我和星友上传的资料）：

文件主题 ▾



洪二

前天 23:49

...

分享一本书，《我的第一本算法书》



我的第一本算法书.pdf



查看详情 >

bruce lee、Leven、zhenguo、洪二 觉得很赞

zhenguo：谢谢分享

前天 23:55

莱布妮子：不错

前天 23:59

bruce lee：感谢分享

昨天 15:02

## 1.5 置顶主题

置顶主题一般是星球的作业贴，精华帖，汇总贴等，注意查看。

全部主题 ▾

**置顶** 【作业汇总】从Day1~Day现在，作业帖和总结帖：【星球如何使用】新手必看...

**置顶** Day9 作业帖 什么是哈希表？今日作业题详见下方链接，在这篇文章里首先总结...

**置顶** Day8 作业帖 LeetCode 经典题：两数之和 今日作业题详见下方链接，在这篇文章...

## 2 程序员还需要学算法吗？

近来经常有朋友问我，程序员还需要学习算法吗？我的数学又不太好，我不想碰算法。

其实你说出了很多程序员的心声，因为很多程序员都持有此想法。

在这里，作为一个工作 6 年多的程序员，告诉大家一个秘密，程序员必须要懂些算法，尤其是基本的算法思维必须得有。

为啥呢？

只有具备算法思维的程序员才能写出赏心悦目的代码，注意程序可不是越短越好哦，而是执行效率高，占用存储空间少；

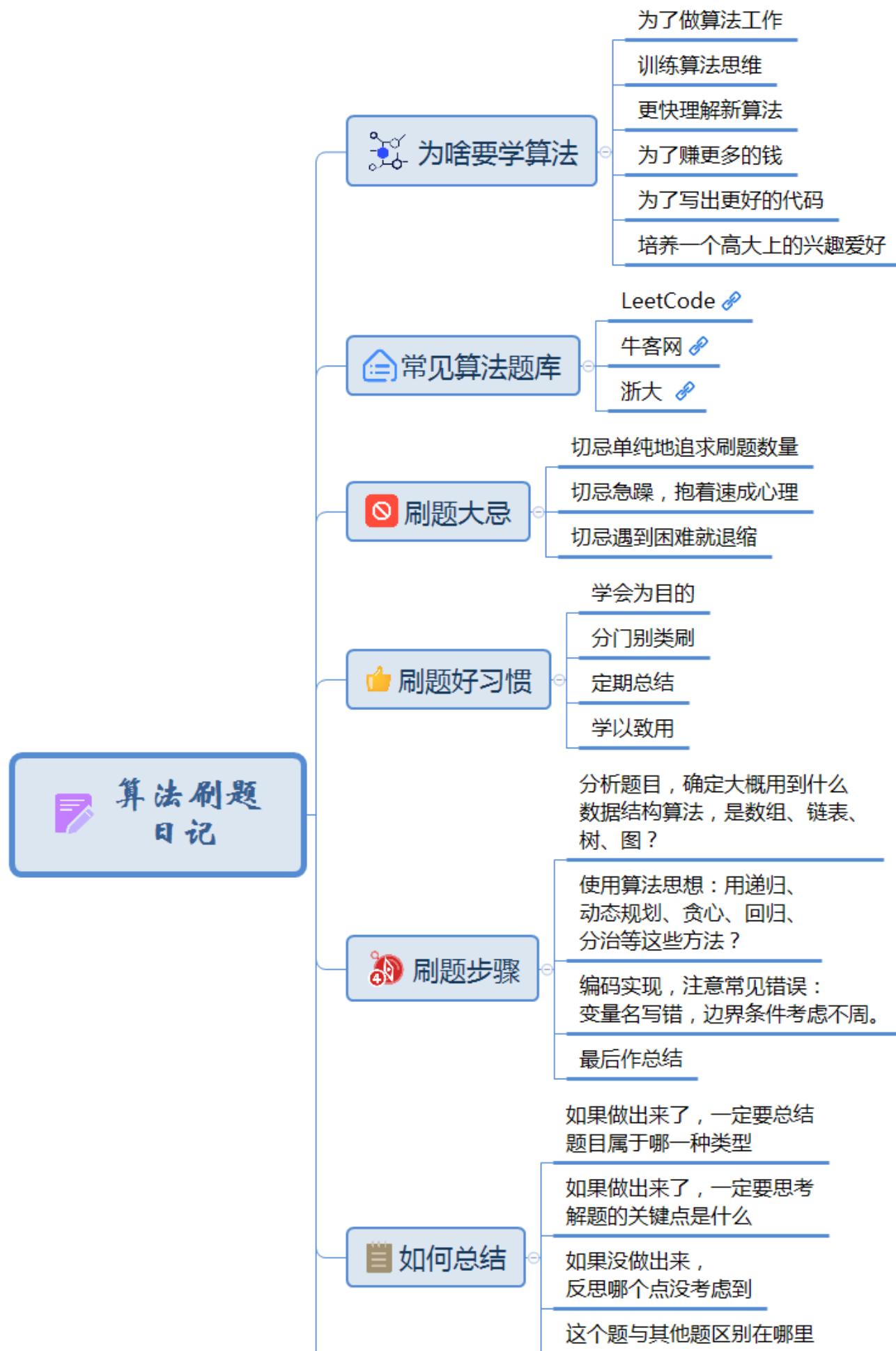
同时，具备算法思维的程序员其实已经甩开没有这方面思维的程序员，一大截！并且，随着码龄变长，优势愈加明显。

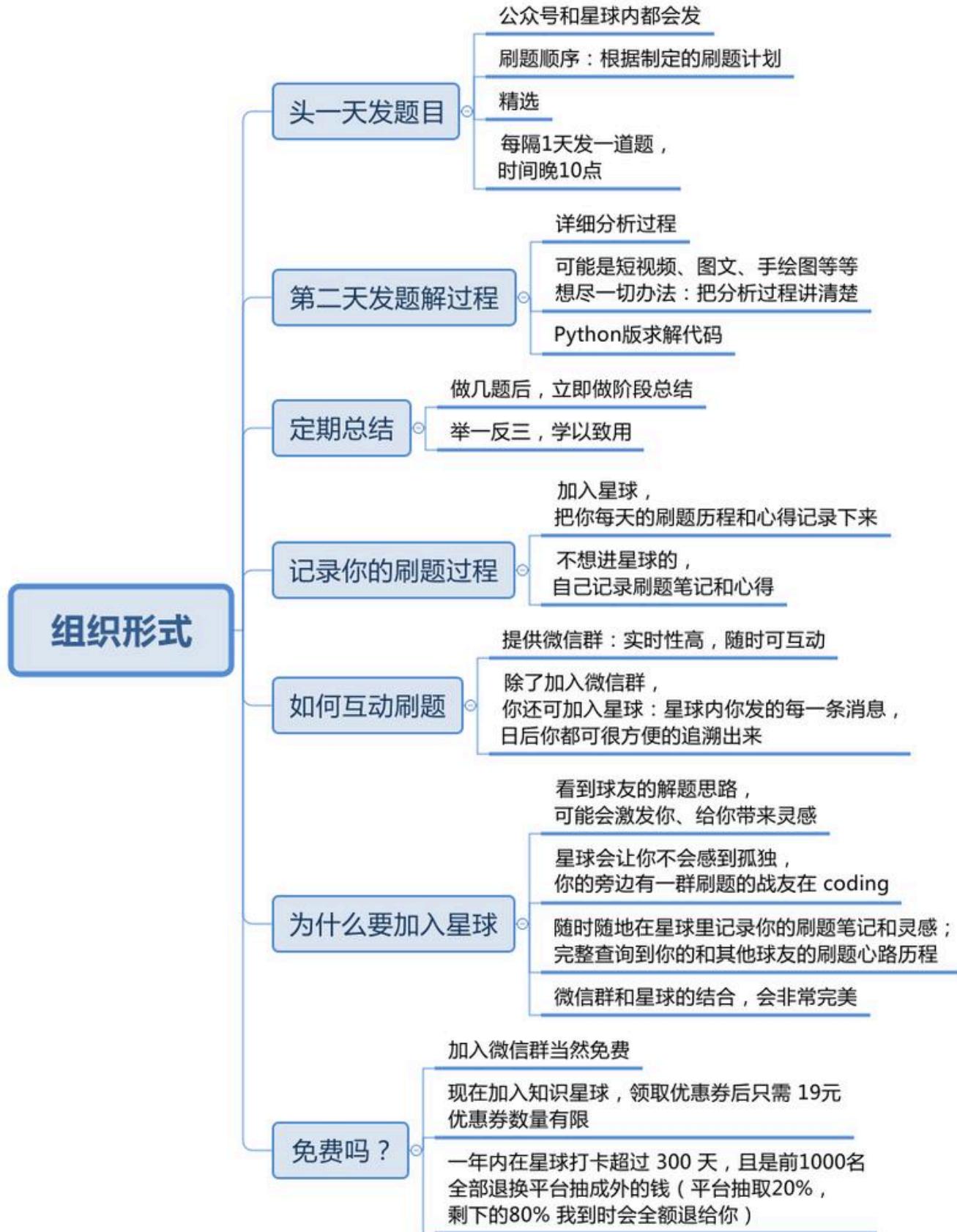
这些都是实在话，信不信由你。

### 3 从零学算法大纲

下面主要有我制定的学习大纲，一个蓝图。







# Day1 冒泡排序

作为算法刷题起航篇，我们有必要先做一个背景介绍，照顾一下算法入门的朋友。

## 1.1 学习算法需要什么基础

至少熟悉一门编程语言 c, c++, python, java等， 推荐 Python，入门简单

不需要任何算法基础

需要强大的毅力：做到不折不挠

养成喜欢总结的习惯

## 1.2 算法入门参考资料

书籍：算法图解

书籍：大话数据结构

书籍：数据结构和算法分析 - C 语言描述

书籍：妙趣横生的算法

免费的，算法可视化动画演示：（强烈推荐）

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

## 1.3 算法进阶参考资料

书籍：算法导论

书籍：编程珠玑

两本极好的外文PDF：免费开源，已上传到知识星球里

## 1.4 算法入门最基本思路

学习数据结构：先了解典型的几种数据结构，数组、链表必备，建立意识：算法与数据结构紧密联系

算法，先从基础的算法开始，通常是排序算法，学会算法的评价指标

入门学习参考视频：油管上已经播放100万次+：[https://www.youtube.com/watch?v=bum\\_19loj9A&t=50s](https://www.youtube.com/watch?v=bum_19loj9A&t=50s)



## 1.5 今日打卡题

学会冒泡排序算法

文章参考：

<https://stackabuse.com/bubble-sort-in-python/>

作业：写出冒泡的代码，并上传到知识星球里，详见下面介绍。

#### 算法刷题 1：精通冒泡排序

对于大多数人来说，冒泡排序可能是他们在计算机科学课程中听说的第一种排序算法。

它高度直观且易于“转换”为代码，这对于新软件开发人员而言非常重要，因此他们可以轻松地将自己转变为可以在计算机上执行的形式。

但是，Bubble Sort 是在每种情况下性能最差的排序算法之一。但是，排序算法也不是一无是处，检查数组是否已排序，它通常优于快速排序等更有效的排序算法。

Bubble Sort 背后的想法非常简单，我们查看数组中相邻的成对元素，一次查看一对。

如果第一个元素大于第二个元素，则交换它们的位置，否则将它们继续移动。

想办法补全如下代码：

```
def bubble_sort(our_list):
    # 写出你的代码
    # 补充完整
    return our_sorted_list
```

## 1.6 参考思路

优化点：

- 1、添加有序标记（flag），当没有元素交换时跳出循环
- 2、记录有序/无序边界，已有序的元素不需要再被进行比较，因此每轮需比较的数列长度会减少

```

knowledgePlanet > BubbleSort.py > ...
1  def bubble_sort(our_list):
2      n = len(our_list)
3      lastExchangeIndex = 0          #记录最后一次交换元素的位置
4      sortBorder = n-1              #无序数列边界
5      for i in range(n):
6          flag = True               #有序标记，每轮开始初始值均为True
7          for j in range(0,sortBorder):
8              if our_list[j] > our_list[j+1]:
9                  our_list[j],our_list[j+1] = our_list[j+1],our_list[j]
10             flag = False            #有元素交换，则将标记置为False
11             lastExchangeIndex = j
12             sortBorder = lastExchangeIndex
13             if flag:
14                 break
15     return our_list

```

学习别人写的代码，也能从中发现一些问题。比如球友小六首先在群里提出来，有的代码把冒泡排序写成选择排序了，并且好多都出现这个问题。所以在此统一吆喝一声，大家看看有没有类似的错误。

区别：冒泡排序比较的一定是紧紧相邻的两个元素；而选择排序却不是每次比较紧邻的两个元素，而下面的代码就不是每次比较紧邻的两个元素，正是选择排序的基本实现。

In [4]:

```

1  def bubble_sort(our_list):
2      our_sorted_list=[]
3      n=len(our_list)
4      for i in range(n):
5          for j in range(i+1, n):
6              if our_list[i]>our_list[j]:
7                  mid_element=our_list[i]
8                  our_list[i]=our_list[j]
9                  our_list[j]=mid_element
10     our_sorted_list=our_list
11     return our_sorted_list
12 our_list=[1,0,3,5,8,7,9,4,2,6]
13 bubble_sort(our_list)

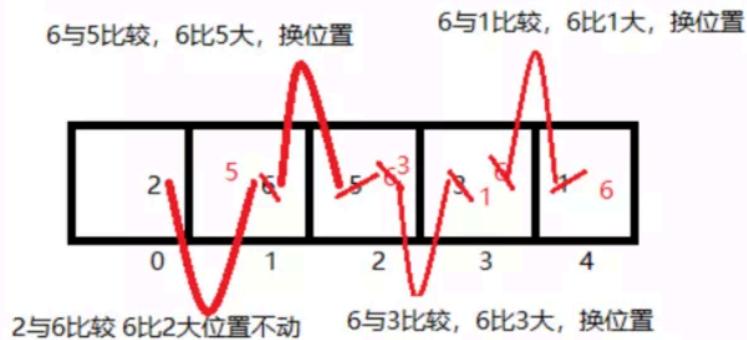
```

Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

下面是冒泡排序的例子：

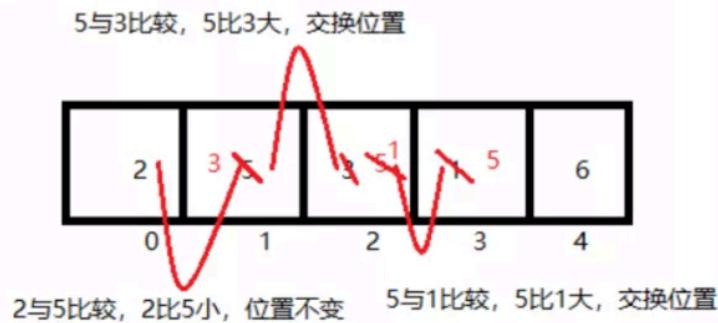
我们从下面这个例子中去学习下冒泡排序；

例如：有一个int [] a={2,6,5,3,1};



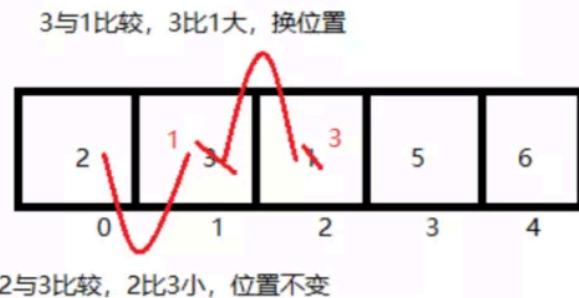
这个就是用冒泡排序的思路进行的第一轮排序：从图中，不难看出第一轮比较。比较了4次；

第二轮排序开始时的数组已经变成了{2, 5, 3, 1, 6}；



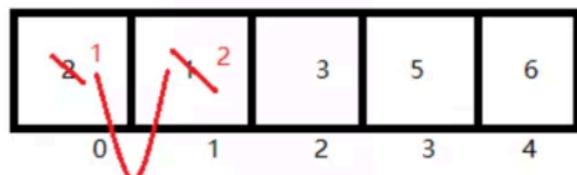
因为第一轮已经确定6的位置，所以，第二轮比较是不再需要再去与这个6比较的，从图可以看出，第二轮比较，比较了3次，确定了5的位置；

第三轮排序开始时的数组已经变成了{2, 3, 1, 5, 6}；



同理，第三轮是不需要去与5进行比较的，从图可以看出，第三轮比较了2次，确定了3的位置。

第四轮排序开始时的数组已经变成了{2, 1, 3, 5, 6}；

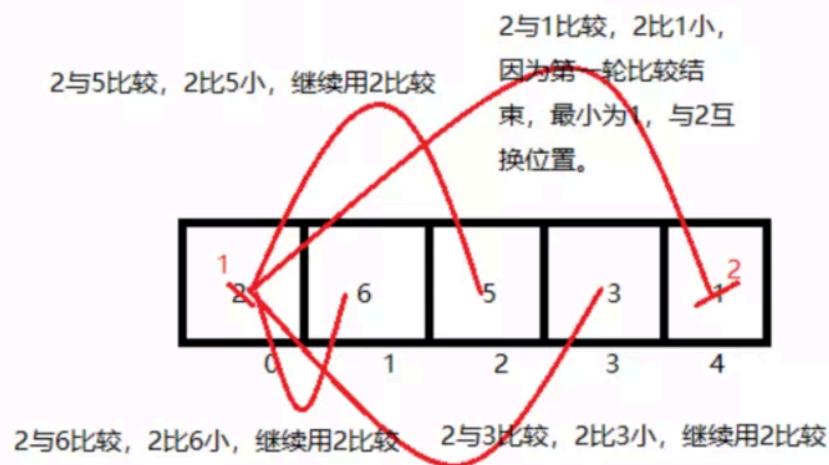


2与1比较，2比1大，换位置

下面是选择排序的例子：

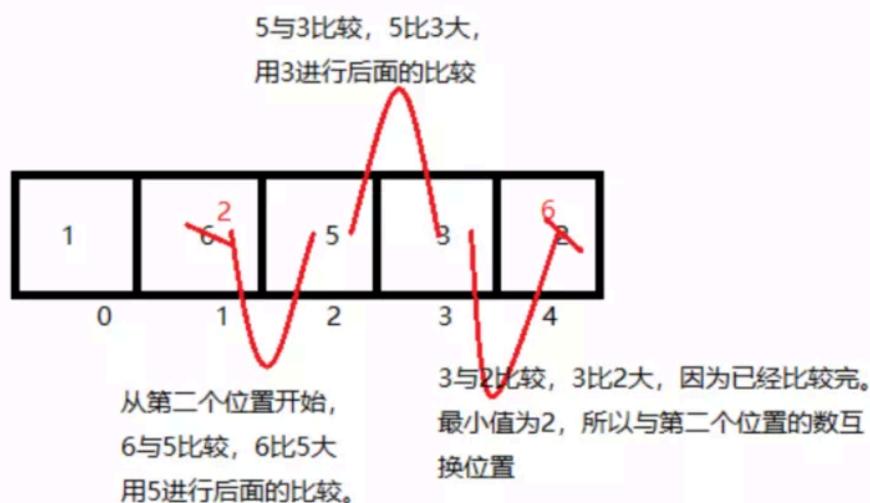
到这里呢，冒泡排序就结束了；下面是选择排序，总结一句话就是(划重点)：从第一个位置开始比较，找出最小的，和第一个位置互换，开始下一轮。

我们同样，以上面的例子为例 int [] a= {2,6,5,3,1};



从图可以看出，第一轮比较，比较了4轮，找出了最小数1，与第一个位置的数字进行了换位；

第二轮排序开始时的数组已经变成了{1, 6, 5, 3, 2}；



从图可以看出，第二轮比较，比较了3次，确定剩余数中的最小数为2，与第二个位置的数交换。

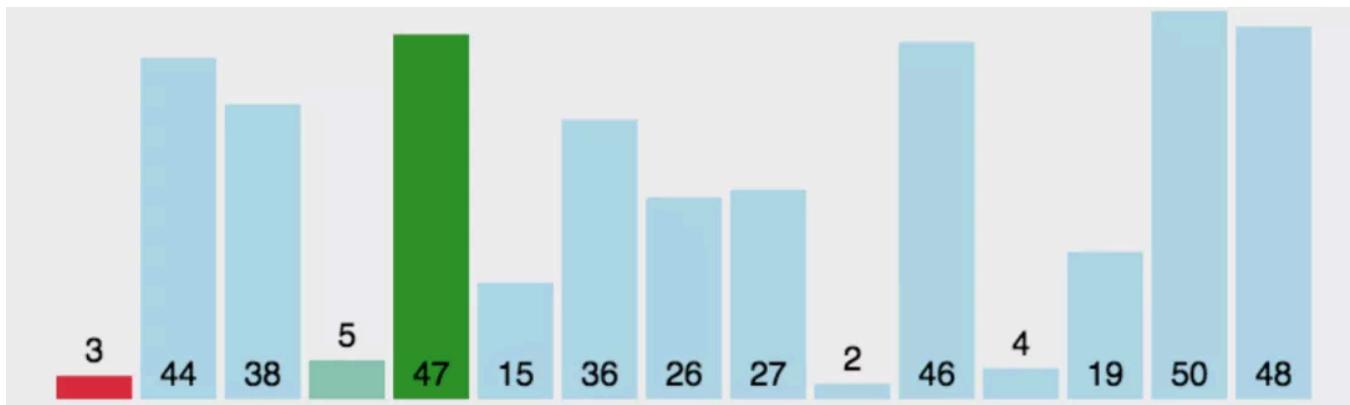
第三轮排序开始时的数组已经变成了{1, 2, 5, 3, 6}；



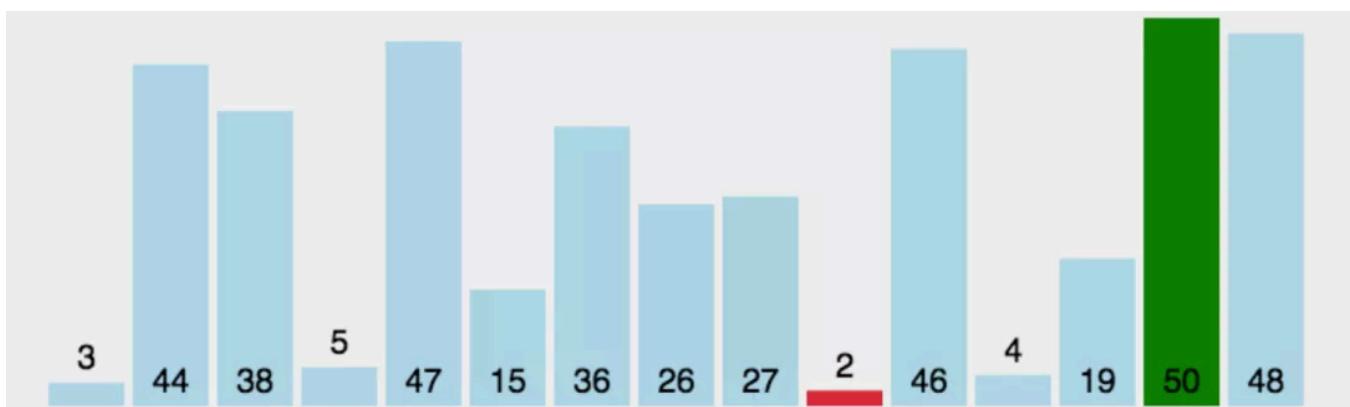
你看一个不起眼的冒泡排序算法，如果细细品味起来也是很有意思的，虽然它的性能注定不好，但是我们的目的是为了训练算法思维，提升算法的分析和应用能力。从这个角度而言，我们的目的达到了。我相信坚持这样分析下去，一定可以让大家的算法思维能力变得更好。

## Day2 选择排序

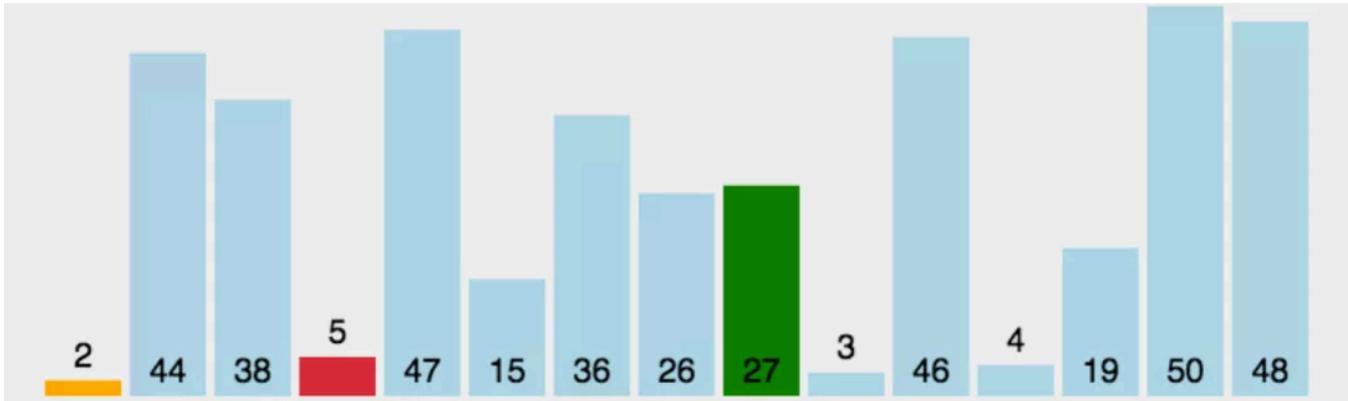
参考下面的几幅图，红色表示当前找到的未排序序列中的最小值，绿色表示当前被比较的元素：



又找到一个更小的值2，重新标记它为红色：



一轮比较后，找到最小值2并标记为黄色，表示就位，继续在未排序序列中寻找最小值：



补全下面代码：

```
def selection_sort(our_list):
    # 补全代码
    #
    #
    return our_sorted_list
```

## 2.1 精选回答

图一：简单选择排序

```
knowledgePlanet > SelectSort.py > ...
1 def selectsort(arr):
2     n = len(arr)
3     for i in range(n-1):
4         mark = i
5         for j in range(i+1,n):
6             if arr[j] < arr[mark]:          #如果有比当前arr[mark]还小的值，则更新mark值，让新的最小值与后续元素继续比较
7                 mark = j
8         if mark != i:                  #找到最小元素的下标值，进行交换。如果arr[i]已经是最小值，#即mark值在内循环中没改变过，则不需要交换元素
9             arr[i],arr[mark] = arr[mark],arr[i]
10    return arr
```

图二：同时找出最小值与最大值放在数列两侧，两边逐渐逼近，循环次数会减少一些

```
knowledgePlanet > SelectSort.py > ...
14  def selectsort(arr):
15      n = len(arr)
16      for i in range(n//2):
17          min_mark = i
18          max_mark = n-i-1
19          for j in range(i+1,n-i):
20              if arr[j] < arr[min_mark]:
21                  min_mark = j
22          if min_mark != i:
23              arr[i],arr[min_mark] = arr[min_mark],arr[i]
24
25          for j in range(n-i-2,i,-1):
26              if arr[j] > arr[max_mark]:
27                  max_mark = j
28          if max_mark != (n-i-1):
29              arr[n-i-1],arr[max_mark] = arr[max_mark],arr[n-i-1]
30
31      return arr
```

选择排序是一种不稳定的排序，虽然代码有优化，但平均时间复杂度始终为  $O(n^2)$ ，有兴趣的可以了解下堆排序（需要有树的基础），时间复杂度可以优化至  $O(n \log_2 n)$

## 2.2 作业点评

### 算法步骤

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。

再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。

重复第二步，直到所有元素均排序完毕。

这种性能差

有一部分球友提交了下面这个版本，此代码中虽然也没有太大问题，但是循环中只要找到小的就交换一次，增加了内存占用。

实际应该为在单次循环时不进行位置交换，而是单次循环后再进行位置交换。

```

def select_fun(list1):
    n = len(list1)
    for i in range(n-1): # 开启循环, 只需要循环n-1次即可,
        # 假设第i个值为最小值, min_value = list1[i]
        for j in range(i+1, n): # j 输入i后面的数字
            if list1[j] < list1[i]: # 如果发现后面存在比最小值还小的数
                list1[i], list1[j] = list1[j], list1[i] # 交换顺序
    return list1

if __name__ == '__main__':
    list2 = [3, 5, 6, 1, 2]
    print(select_fun(list2))
    """
    # 输出结果
    [1, 2, 3, 5, 6]
    """

```

## 性能良

大部分球友提交的代码实现版本：

```

def select_fun(list1):
    n = len(list1)
    count = 0 # 统计更改次数
    for i in range(n-1): # 开启循环, 只需要循环n-1次即可,
        min_index = i
        for j in range(i+1, n): # j 输入i后面的数字
            if list1[j] < list1[min_index]: # 如果发现后面存在比最小值还小的数
                min_index = j # 这里暂时不更改位置, 等循环结束后再更改
        if min_index != i: # 判断当前的最小值是不是i所在位置
            list1[i], list1[min_index] = list1[min_index], list1[i]
            count += 1
    print('交换次数', count)
    return list1

if __name__ == '__main__':
    list2 = [3, 5, 6, 1, 2]
    print(select_fun(list2))
    """
    # 输出结果
    交换次数 4
    [1, 2, 3, 5, 6]
    """

```

性能更好的实现

```
knowledgePlanet > SelectSort.py > ...
14  def selectsort(arr):
15      n = len(arr)
16      for i in range(n//2):
17          min_mark = i
18          max_mark = n-i-1
19          for j in range(i+1,n-i):
20              if arr[j] < arr[min_mark]:
21                  min_mark = j
22          if min_mark != i:
23              arr[i],arr[min_mark] = arr[min_mark],arr[i]
24
25          for j in range(n-i-2,i,-1):
26              if arr[j] > arr[max_mark]:
27                  max_mark = j
28          if max_mark != (n-i-1):
29              arr[n-i-1],arr[max_mark] = arr[max_mark],arr[n-i-1]
30
31      return arr
```

## Day3 算法基本要素

什么是一个算法？程序就等于算法吗？

理解以上参考清华大学邓俊辉老师的视频：

<https://next.xuetangx.com/learn/THU08091000384/THU08091000384/1516243/video/1387095>

❖ 所谓算法，即特定计算模型下，旨在解决特定问题的指令序列

输入 待处理的信息（问题）

输出 经处理的信息（答案）

正确性 的确可以解决指定的问题

确定性 任一算法都可以描述为一个由基本操作组成的序列

可行性 每一基本操作都可实现，且在常数时间内完成

有穷性 对于任何输入，经有穷次基本操作，都可以得到输出

... ...

## 3.1 精选回答

精选回答1

## 算法(Algorithm)

---

- An algorithm is a sequence of unambiguous instructions for solving a problem, Satisfying:
  - (1) **Input**: 0 or more valid input values
  - (2) **Output**: at least one value is produced
  - (3) **Definite**: each instruction / each step is clearly precisely and unambiguously specified
  - (4) **Finiteness**: finite instructions, finite execution times for each instruction, finite running time for each instruction

## 程序(Program)

---

- an implemented coding of a solution to a problem based on the algorithm
- could be infinitely executed
- E.g., Operating system: not an algorithm, but a program running in infinite circles
- each task could be viewed as subprogram according to specific algorithm

精选回答2



21小时前

## 第三天打卡

借用一句话和两个公式大概可以回答什么是算法以及程序和算法的区别

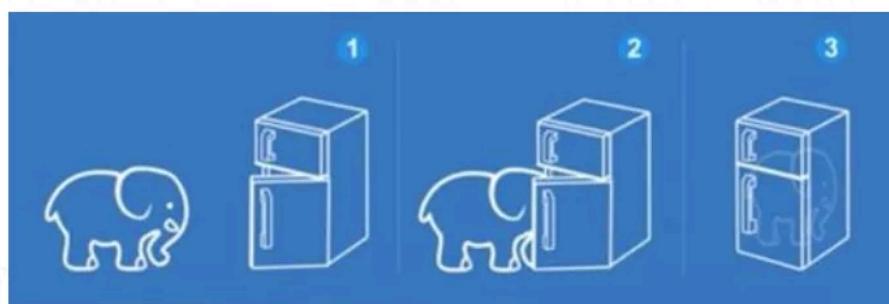
Computer science should be called computing science, for the same reason why surgery is not called knife science.

我理解的算法本质是计算模型

Algorithms+Data Structures=Programs

(Algorithms+Data Structures) \*Efficiency=Computation

算法是程序的一部分，而效率是算法的内容也是算法的目标



| 查看作业题目 >

## 精选回答3



20小时前

算法是指解决问题的一种方法或一个过程。程序是算法用某种程序设计语言的具体实现。程序不等于算法，因为程序不一定满足有穷性

| 查看作业题目 >

根据清华大学邓俊辉老师的解释，算法的特质包括但不限于这些：

## ❖ 所谓算法，即特定计算模型下，旨在解决特定问题的指令序列

<b>输入</b>	待处理的信息（问题）
<b>输出</b>	经处理的信息（答案）
<b>正确性</b>	的确可以解决指定的问题
<b>确定性</b>	任一算法都可以描述为一个由基本操作组成的序列
<b>可行性</b>	每一基本操作都可实现，且在常数时间内完成
<b>有穷性</b>	对于任何输入，经有穷次基本操作，都可以得到输出
...	...

大家注意，算法特征邓工在最后是使用省略号，说明算法的特点不仅仅包括以上几条。

下载地址：

<https://t.zsxq.com/JEuVbYB>

## Day4 Hailstone

算法的这些主要特点有时并不像我们想象的那样好去证明。比如，我们会觉得算法有穷性是最容易判断的，其实不然，邓工在课程中介绍了一个例子，在此我引用一下：

```
def hailstone(n):
    length = 1
    while(1 < n):
        if n % 2 == 0:
            n = n/2
        else:
            n = n*3 + 1
        length += 1
    return length
```

以上这个 `hailstone` 函数满足有穷性吗？意思是对于任意的 `n`, `while` 循环都会执行有限次而退出吗？

根据邓工的课程，我们得知：

至今学术界都无法证明对于任意的  $n$ ，一定满足： $\text{hailstone}(n) < \infty$

请列举几个不同的  $n$  值(如  $n=120, 7, 27$ )，分别求出 `hailstone` 的返回值，体会证明算法有穷性的困难。

## 4.1 精选回答

精选回答1

### 希尔顿序列 (Hailstone Sequence)

希尔顿序列（即考拉兹猜想，又称奇偶归一猜想）是一个著名的数学问题，至今没有证明其正确性，也没证明其是错误的。即任何一个正整数，如果是偶数的话就除以2，如果是奇数的话就乘以3再加上1，最后这个数都会变为1。其表达式如下：

$$\text{Hailstone}(n) = \begin{cases} 1 & n = 1 \\ n/2 & n \% 2 = 0 \\ 3n + 1 & n \% 2 \neq 0 \end{cases}$$

问题的特殊之处在于：尽管很容易将这个问题讲清楚，但直到今天仍不能保证这个问题的算法对所有可能的输入都有效——即至今没有人证明对所有的正整数该过程都终止。

```
import random
def hailstone(n):
    hailstone_sequence = []
    length = 1
    while(1 < n):
        if n % 2 == 0:
            n = n//2
        else:
            n = n*3 + 1
        hailstone_sequence.append(n)
        length += 1

    # return length
    return hailstone_sequence

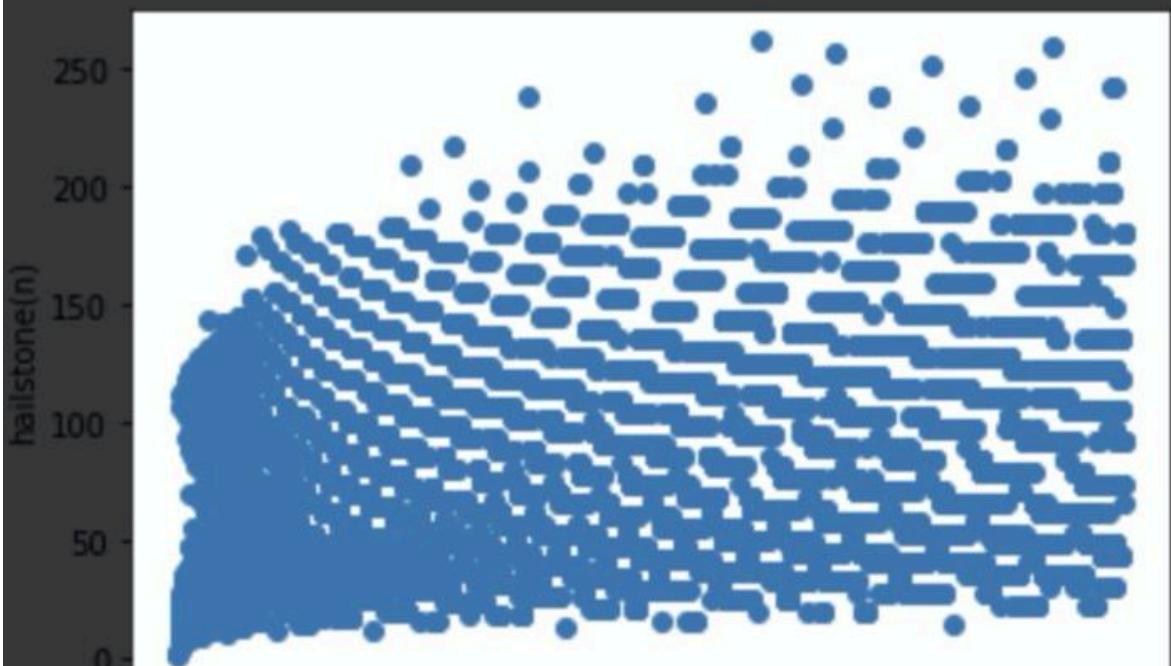
if __name__ == "__main__":
    n = random.randint(1,1000)
    print("%d的Hailstone的序列是" %n,hailstone(n))
```

精选回答 2

下面我们先绘制 1 万以内的正整数其 Hailstone 序列的长度，代码参考我的知识星球中星友

@Kevin :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def hailstone(n):
5     length = 1
6     while(l<n):
7         if n%2 == 0:
8             n = n/2
9         else:
10            n = n*3 + 1
11        length += 1
12    return length
13 x = [ n for n in range(10000) ]
14 y = [hailstone(h) for h in x]
15
16 plt.scatter(x,y)
17 plt.xticks(np.arange(min(x),max(x)+1 , 1000))
18 plt.xlabel('n')
19 plt.ylabel('hailstone(n)')
20 plt.show()
```



注意，代码有一个瑕疵，应该是  $n = n//2$  这样才会返回一个整数

因为图形貌似冰雹，所以被称为 Hailstone 序列。

然后，星友@guolong 提出一个很好的问题，是否存在一些数，使while 循环一直执行下去？



guolong

8 小时前

day4 希尔顿序列

是否存在一些数，使下面的while循环一直执行下去？

个问题困扰着世人，至今都没有确定的答案。

## 4.2 Collatz 猜想

有一个很有意思的问题，我们这样实验：

```
def hailstone(n):
    length = 1
    seq = []
    while(1 < n):
        if n % 2 == 0:
            n = n//2
            seq.append(n)
        else:
            n = n * 3 + 1
            seq.append(n)
    length += 1
    return length, seq

print(hailstone(7))
print(hailstone(17))
print(hailstone(103))
```

打印结果如下，数字7,17,103的 Hailstone 序列长度分别为 17， 13， 88，最惊奇的是：每个这种序列的最后三个数一定是 4,2,1

```
(17, [22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1])
(13, [52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1])
(88, [310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 13
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911,
866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 8
```

感兴趣的可一直向右滑动，查看序列的结尾是不是4,2,1这个周期！

并且已经得出：无论使用什么起始值，每个hailstone 序列最终都将停留在4、2、1个周期上。

计算机甚至已经检查了所有起始值（最大到  $5 \times 2_{60}$ ）（一个 19 位数字），并发现最终出现4、2、1个周期。

但是很遗憾：依然没有科学家能够证明所有序列都是这种情况。

这个开放问题在数学家 Lothar Collatz 于 1937 年首次提出该问题之后被称为 Collatz 猜想。

令人惊讶的是，即使是最好的数学家都无法回答，如此简单的形成序列的公式也会引发一个问题。

## Day5 数组中插入元素

那么接下来，我们趁热打铁，先学习最最基础的数据结构：array(数组)和vector(向量)，数据结构和算法是相辅相成的，二者结合彰显算法之美，所以对于常见数据结构的掌握是很必要的。

参考教材如下：

<http://www.xuetangx.com/courses/course-v1:TsinghuaX+30240184+sp/pdfbook/0/>

Day5 打卡题：

```
def insert(lst, i, e):
    """
    lst: 一个数组或向量, Python 就用 list 表达吧
    i: 待插入元素的位置
    e: 待插入元素
    """
    #
    #补全代码
    #
```

## 5.1 精彩回答

1、下标处理：正/负数绝对值大于数组长度则添加在数组末尾和首位，负数绝对值小于数组长度则用`index+len(list)`来处理

2、`None`值判断及优化：若插入位置刚好是`None`值，则直接替换

(PS：元素的插入需要考虑扩容问题，如果原本数组已满，则需要扩容。实际上许多API底层也是用最原始的方式去实现的，只是封装起来了就成了方便调用的API，建议可以的话尝试自己的方式去实现；顺便想下，如果把它封装起来需要优化哪些地方。)

```

class Array(object):
    def __init__(self, lst):
        self.size = len(lst)
        self.lst = lst

    def insert(self, index, e):
        count = 0           #用于判断数组中的是否存在None值
        if index < 0:
            if (-index) > self.size: #下标处理
                index = 0
            else:
                index = index+ self.size
        if index > self.size:
            index = self.size

        #None值判断及优化
        if self.lst[index] is None:
            self.lst[index] = e
            return self.lst
        else:
            for i in range(self.size-1, index, -1):
                if self.lst[i] is None:
                    count += 1

        if count == 0:       #没有None值，扩容数组
            self.resize()

        #插入数据
        for i in range(self.size-2, index-1, -1):
            if (self.lst[i] is not None) and (self.lst[i+1] is None):
                self.lst[i+1] = self.lst[i]
                self.lst[i] = None

        self.lst[index] = e

        return self.lst

    def resize(self):
        self.size = self.size * 2
        lst2 = [None for i in range(self.size)]
        for i in range(len(self.lst)):
            lst2[i] = self.lst[i]
        self.lst = lst2.copy()

if __name__ == '__main__':
    my_list = [1, 2, 3, 4, 8, None]
    my_array = Array(my_list)
    my_array.insert(1, 7)
    print(my_array.lst)

```

# Day6 求中心索引

给定一个整数类型的数组 `nums`，请编写一个能够返回数组“中心索引”的方法。

我们是这样定义数组中心索引的：数组中心索引的左侧所有元素相加的和等于右侧所有元素相加的和。

如果数组不存在中心索引，那么我们应该返回 `-1`。如果数组有多个中心索引，那么我们应该返回最靠近左边的那一个。

示例 1：

输入: `nums = [1, 7, 3, 6, 5, 6]` 输出: 3 解释: 索引 3 (`nums[3] = 6`) 的左侧数之和( $1 + 7 + 3 = 11$ )，与右侧数之和( $5 + 6 = 11$ )相等。同时, 3 也是第一个符合要求的中心索引。

示例 2:

输入: `nums = [1, 2, 3]` 输出: -1 解释: 数组中不存在满足此条件的中心索引。

请补充完成下面的代码：

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

## 6.1 精彩回答

参考星友 infrared62\* 的分析和代码：

```
1  class Solution:  
2      def pivotIndex(self, nums: List[int]) -> int:  
3          total = sum(nums)  
4          left = 0  
5          for i in range(len(nums)):  
6              if left == total - left - nums[i]:  
7                  return i  
8              left += nums[i]  
9  
10     return -1
```

这种解法是很高效的。

infrared62\* 的总结：Java and Python，前缀和的应用，前缀和可以简单看作数列前n项的和，在DP和树路径求和也有应用，同理还有后缀和，前缀积，后缀积。

## 6.2 不高效解

不过，我也看到有些星友是这么求解的：

```
def center_index(nums):
    for i in range(len(nums)):
        if sum(nums[:i]) == sum(nums[i + 1:]):
            return i
    return -1
```

相比第一种解法，这种求解方法不高效。

因为每迭代一次，都要 `sum` 求和两次，而 `sum` 求和本质也是一个循环，所以相当于嵌套 `for` 循环。

我们需要思考，是否有必要每次都要求和，显然是不必要的。

如果存在中心索引，则一定满足：中心索引左侧和 \* 2 + `nums[i]` == `sum(nums)`

而 `sum(nums)` 一定是个定值，中心索引的左侧求和可放在循环中逐渐累加得到，所以只用一层 `for` 即可。

## 6.3 算法分析总结

往往我们第一下想到的解法未必是最高效的，还要多加分析，培养算法思维才行。而如果想要培养算法思维和敏锐度，就要多加练习，通常来回训练 LeetCode 题是不错的方法。

训练后的结果，就像上面的星友 infrared62\* 一样，看到这道题，马上想到一连串的类似思想：前缀和可以简单看作数列前n项的和，在DP和树路径求和也有应用，同理还有后缀和，前缀积，后缀积。

加油，只要我们坚持下去。

# Day7 如何培养算法思维

之前查到过一个比较好的算法学习方法总结，来自清华大学算法训练营 longyue0521 ，提出的“做中学”方法，个人也是比较认同的，大家不妨看看下面的详细介绍。

事半功倍：Learning by doing 做中学

## 7.1 经历描述

在我初学编程时，因没有掌握计算机相关专业的学习方法，走了不少弯路。

我总是想先“打好基础”，再走下一步，但这需要时间、毅力与坚持。

我花了很多气力学习，但都事倍功半！

我想找到效率更高的学习方法，于是我开始浏览美国计算机四大名校的课程网站。

经过一番研究，自学几门课程后，我发现了他们的教学套路：

- 教授理论知识（一），小作业，用于巩固理论知识（一）
- 教授理论知识（二），小作业，用于巩固理论知识（二）
- 大作业，编程实践，需要用到理论知识（一）与（二）
- 教授理论知识（三），小作业，用于巩固理论知识（三）
- 教授理论知识（四），小作业，用于巩固理论知识（四）
- 大作业，编程实践，需要用到理论知识（三）与（四）
- 项目作业，编程实践，多人协作，需要用到理论知识（一）～（四）+ hits
- 重复上述过程，一般重复4～6次，中间穿插期中考试，最后期末考试

由此“套路”总结出另一种学习方法——“迭代学习”法：

- 理解待解决的问题
- 学习部分理论知识
- 动手实践尝试解决，无法解决，回到1或2
- 成功解决抓紧总结
- 即使现在回头看，我也不能说第一种学习方法有错，“迭代”学习法更好！但这两种学习方法都是以同一个核心为基础的——动手做，做中学！

你可以都尝试一下，然后选取自己喜欢、又高效的学习方法！

当然也欢迎分享你的学习方法！

## 7.2 经验总结

越早适应“迭代”学习法对你越有利。

大多数时候你没有足够的时间来“学完再做”。

若你在“迭代”学习过程产生的焦虑、沮丧、挫败感，请及时排解  
排解后记得回来，坚持才能胜利！

#### 个人建议

学习《算法设计》在借鉴学习《数据结构》的经验的同时，

需要做适当调整——在每次大迭代中应用“迭代”学习法：

第一次迭代，熟悉常用的算法设计策略，掌握策略的使用方法及适用的场景

- 其实学习《数据结构》时你已经学了不少经典算法
- 带着学到的算法设计策略回头总结、归纳经典算法
- 可以在纸上画画设计策略与经典算法的关系图，是一对一，一对多，还是多对多
- 这个阶段的重点，在脑中建立常用算法设计策略与经典算法的对应关系
- 如果个人能力不错，可考虑与《数据结构》第三次迭代同时进行

第二次迭代，灵活运用算法设计策略，解决实际问题

- 大量的解决问题，在此过程中总结出你个人解决问题的流程
- 可以针对某项设计策略进行专项训练，但要考虑实际需求——工作、面试、竞赛
- 此阶段的重点就是解决《数据结构》第三次迭代中的隐藏关卡，同时培养解决问题的感觉、自觉
- 别忘了“迭代”学习方法
- 大量训练、多与他人探讨、扩展自己的思路并及时总结

第三次迭代，对给定问题能运用数学证明你的算法设计策略是正确的、可行的、高效的

- 这个阶段要做的事本应该融入到前两次迭代中的，甚至更早比如在离散数学课上
- 之所以单抽出来是因为有太多的人因这个“拦路虎”而徘徊在“算法设计”的大门前迟迟不敢踏入半步，更有甚者转身离开了就再也没有回来.....
- 如果你不擅长数学，或不打算从事科研及对数学要求较高的工作，可以跳过
- 对于打算从事科研及对数学要求较高的工作的人来说，这也可以说迂回策略
- 可以先从教材对经典算法的证明学习，然后重走第二次迭代实践（这才是看CLRS的时机）
- 这个阶段的重点，有意识地运用数学来决定设计策略的选取

## 7.3 追求目标

你在学习《算法设计》的初期没能养成良好的推理、证明习惯，后期改正要费些功夫信心/底气不足，有强大的数学理论作为支撑你敲代码、测试或和别人辩论也底气十足尽管不完美但比起那些徘徊在门口、转身离开的人，你已进入“算法设计”的大门！

这难道不值得高兴吗？

## 7.4 精选回答

打卡第七天

我本人对编程最深的追求，大概是想在解决现实问题的过程中写出最漂亮的代码。曾经入手python时，以为的漂亮就是越简洁越好越短越好。什么时候认识到算法和数据结构的问题呢，大概是有些代码的写法的内在逻辑让我头秃的时候。我不能忍受自己死记硬背一些明明很有逻辑的东西，那个时候就开始回头一点点交补课费了。感谢振哥以及这个平台。最开始我都没想过打卡，就是花点钱看看算法的大佬都是什么样子。可振哥的作业设计有介绍有链接有大家的分享，一下子对我宽容了许多。虽然这并不能替代我自己去深耕这些知识背后的逻辑，但的确给了我些许的勇气告诉自己这也可以慢慢学。事实上，仔细去学习这些根本上的东西对自己理解代码书写代码思考问题的解决方式真的有很明显的改变。写这些感觉写早了，毕竟300天才走了7天，但也的确想写出来，接下来的路还要坚持走，给自己加油！

## Day8 两数之和

首先看下题目描述：

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那两个整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例：

给定 `nums = [2, 7, 11, 15]`, `target = 9`

因为 `nums[0] + nums[1] = 2 + 7 = 9`

所以返回 `[0, 1]`

题目参考链接：<https://leetcode-cn.com/problems/two-sum>

考察知识点：数组，哈希表(字典)

请补充完整如下代码：

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

这道题目大家仔细思考下，想办法写出尽可能高效的代码。提示：牺牲空间换取时间

## 8.1 精选回答

思路：结合题目及提示这里使用字典。

用target减去列表中的某个值，并将该值作为key，其下标作为value存放到字典中，接着target依次减去剩下的列表项中的值并判断结果是否存在于字典，如果存在即表示列表中有两个值相加等于target，此时即可直接返回答案。并且字典存储可以起到去重的作用。

```
1 #两数之和  
2 class Solution:  
3     def twoSum(self,nums,target):  
4         dict = {}  
5         for i,n in enumerate(nums):  
6             if target-n in dict:  
7                 return [dict[target-n],i]  
8             else:  
9                 dict[n] = i  
10  
11 if __name__ == "__main__":  
12     so = Solution()  
13     result = so.twoSum([1,8,9,6,2],10)  
14     print(result)
```

## 8.2 分析总结

大家注意审题，确定输入是什么，输出又是什么，假定又是什么。

输入：待寻找的列表 `nums`, 两数之和 `target`

输出：有且仅有满足要求的一对整数的下标

假定：一定存在，且仅有一个答案

题目分析：两个数之和等于 `target`, 首先标记所有遍历过的数，如果 `target` 减去当前被遍历到的值 `e` 后，即 `target-e` 被标记过，则找到答案。

判断值是否在某个容器中，做到  $O(1)$  时间复杂度的便是最常用的散列表，对应 Python 中的字典。就本题而言，键为标记元素值，字典值为数组下标，所以更加确定使用字典这个数据结构。

代码：

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        d = {}
        for i,e in enumerate(nums):
            if target - e in d:
                return [i,d.get(target-e)]
            d[e] = i
```

以上是比较高效的解法之一。

从星球中星友提交的代码看，有一些星友的代码就是上面的实现思路。

但是，也有一些星友的代码是这样的，解并没有达到时间复杂度为  $O(n)$ ，大家不妨参考并回头检查下自己写的。

## 8.3 不高效解 1

`index` 复杂度为  $O(n)$ , 所以实际时间复杂度为  $O(n^2)$ ，尽管表面上看只有一个 `for` 循环。



9 小时前

#算法刷题# Day 8

```
def twoSum1(nums, target):  
    for index, num in enumerate(nums):  
        another_num = target - num  
        nums[index] = None  
        if another_num in nums:  
            return [index, nums.index(another_num)]  
    return None
```

## 8.4 不高效解 2

下面代码两层 for，空间复杂度虽然为 O(1)，但是时间复杂度为 O( $n^2$ )。所以需要找到牺牲空间换取时间的方法。



```
#打卡#
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        n = len(nums)
        for i in range(n):
            for j in range(i+1,n):
                if nums[j] == target - nums[i]:
                    return i,j
                break
            else:
                continue
```

以上使用散列表牺牲空间，但是换取时间，实际中能找到节省时间的解往往更有价值。

## Day9 什么是哈希表

我们来学习最重要的数据结构之一：散列表或哈希表。

那么什么是哈希表呢？哈希表怎么做到  $O(1)$  时间复杂度找到某个元素的呢？

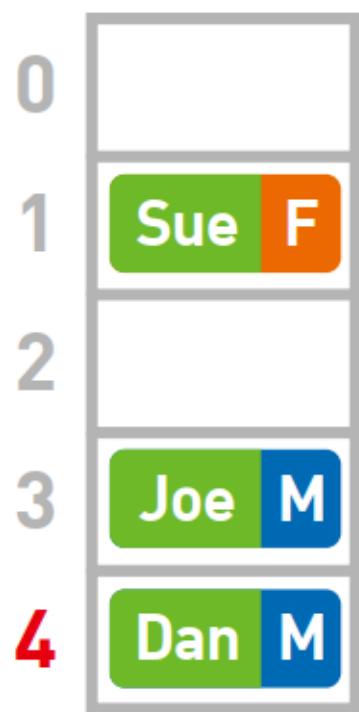
提供参考资料《我的第一本算法书》

大家可参考如下哈希表的基本介绍：

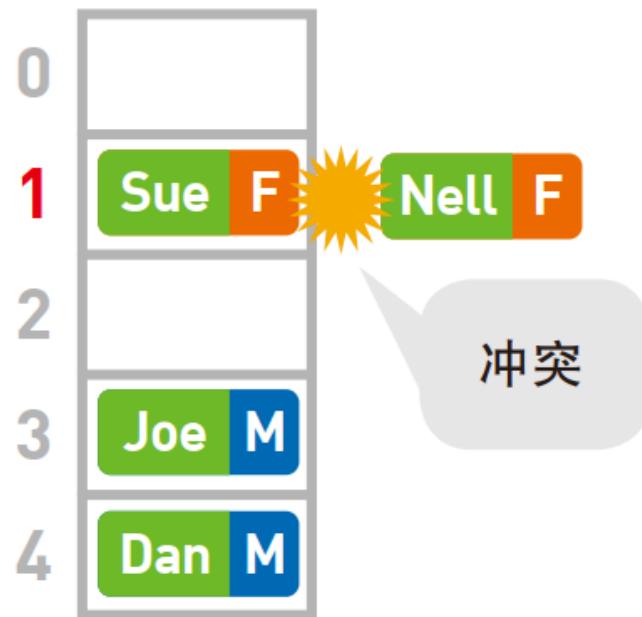
Key	Value
Joe	M
Sue	F
Dan	M
Nell	F
Ally	F
Bob	M

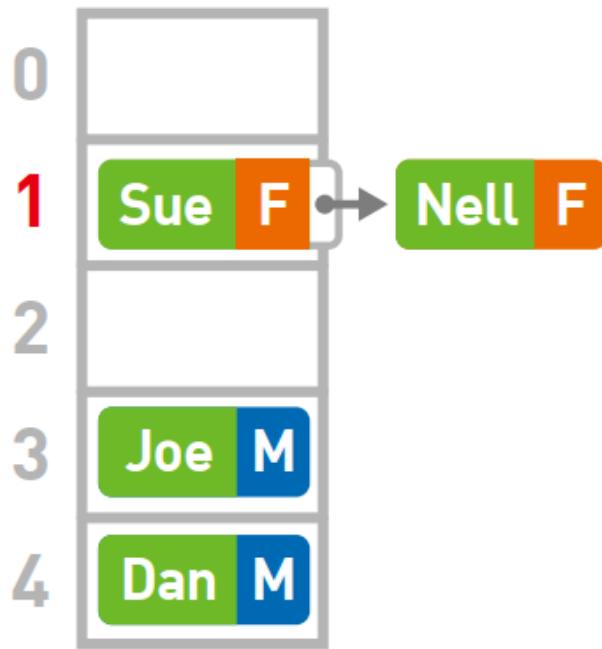
M 表示性别为男，F 表示性别为女。

哈希表存储的是由键（key）和值（value）组成的数据。例如，我们将每个人的性别作为数据进行存储，键为人名，值为对应的性别。



Nell → Hash →  $6276 \bmod 5 = 1$





遇到这种情况，可使用链表在已有数据的后面继续存储新的数据。关于链表的详细说明请见1-2节。

## 9.1 精彩回答

概念：哈希表是根据键值（key value）而直接进行访问的数据结构。它通过把键值映射到一个位置来访问记录，以加快查找的速度。

具体映射过程是：把 Key 通过一个映射函数转换成一个整型数字，然后将该整型数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该取余数字为下标的数组空间里。

这个映射函数称为 **哈希函数**，映射过程称为哈希化，存放记录的数组叫做 **哈希表**。

**查询：**数组的特点是寻址容易，插入和删除困难；而链表的特点是寻址困难，插入和删除容易；哈希表则实现了寻址容易，插入删除也容易。

当使用哈希表进行查询的时候，就是再次使用哈希函数将key转换为对应的数组下标，并定位到该空间获取value。哈希表是一个在时间和空间上做出权衡的数据结构。

如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么可以使用无序数组并进行顺序查找，这样只需要很少的内存。

**冲突：**哈希映射对不同的键，可能得到同一个散列地址，即同一个数组下标，这种现象称为冲突。

避免哈希冲突：

1.**拉链法**：通过哈希函数，我们可以将键转换为数组的索引(0~M-1)，但是对于两个或者多个键具有相同索引值的情况，我们需要有一种方法来处理这种冲突。一种比较直接的办法就是，将大小为M的数组的每一个元素指向一个链表，链表中的每一个节点都存储散列值为该索引的键值对，这就是拉链法。

2.**线性探测法**：基本原理为，使用大小为M的数组来保存N个键值对，其中M>N，我们需要使用数组中的空位解决碰撞冲突。当碰撞发生时即一个键的散列值被另外一个键占用时，直接检查散列表中的下一个位置即将索引值加1，这样的线性探测会出现三种结果：1) 命中，该位置的键和被查找的键相同；2) 未命中，键为空；3) 继续查找，该位置和键被查找的键不同。

# Day10 哈希表设计艺术

## 10.1 什么是哈希表？

哈希表是一种使用哈希函数组织数据，以支持快速插入和搜索的数据结构。

有两种不同类型的哈希表：**哈希集合**和**哈希映射**。

哈希集合是集合数据结构的实现之一，用于存储非重复值。

哈希映射是映射数据结构的实现之一，用于存储(key, value)键值对。

## 10.2 哈希表原理

哈希表的关键思想是使用哈希函数将键映射到存储桶。

当插入一个新键时，哈希函数决定该键应该分配到哪个桶中，并将该键存储在相应的桶中；

当搜索一个键时，哈希表使用相同的哈希函数来查找对应的桶，并只在特定的桶中进行搜索。

## 10.3 哈希集

哈希集是集合的实现之一，对应 Python 中的 **set** 类型

## 10.4 哈希映射

哈希映射是用于存储 (key, value) 键值对的一种实现，对应 Python 中的 **dict** 类型

## 10.5 设计键

设计哈希表时，选择合适的键是一门艺术。

这次 **Day 10：字母异位词分组** 打卡题来训练一下，如何为哈希表设计合适的键。

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

示例：

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

输出:

```
[  
  ["ate", "eat", "tea"],  
  ["nat", "tan"],  
  ["bat"]  
]
```

说明：

所有输入均为小写字母。

不考虑答案输出的顺序。

```
class Solution:  
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
```

## 10.6 精选答案

今天很多星友都提交了很不错的答案，切实感受到设计哈希表的键是一门艺术，同时也真正体会到众人拾柴火焰高，越来越喜欢咱们的星友们和这个平台，我们一起加油走下去，3个月后遇见更好的自己。

参考大家的打卡，精选几种解法：

### 第一种设计键的方法

对原数组中的每个字符串进行排序，如果是字母异位词，那排序后的字符串肯定就是相等的。因此可以以排序后的字符串为Key，Value则为字符串列表。

```
class Solution(object):  
    def groupAnagrams(self, strs):  
        dict = {}  
        for s in strs:  
            key = tuple(sorted(s))  
            dict[key] = dict.get(key, []) + [s]  
        return list(dict.values())
```

### 第二种设计键的方法

以元组统计字符串中每个字符出现的次数，并做为哈希表的Key，Value为字符串列表

```

from collections import defaultdict
class Solution:
    def groupAnagrams(self, strs):
        dict = collections.defaultdict(list)
        for s in strs:
            count = [0] * 26
            for i in s:
                count[ord(i) - ord('a')] += 1
            dict[tuple(count)].append(s)
        return list(dict.values())

```

## 第三种设计键的方法

质数方式表示，**26**个小写字母，以**26**个质数表示，然后把字符串各个字母相乘即对应的各个质数相乘，得出的结果作为Key，Value为字符串列表。

**@Leven** 星友非常贴心的提示：注意这里如果字符串长度很长，质数相乘的结果可能会造成int溢出，不过Python3只有长整型，以int表示，基本不会溢出。

```

class Solution(object):
    def groupAnagrams(self, strs):
        dict = {}
        z = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101]
        for i in range(len(strs)):
            key = 1
            for j in range(len(strs[i])):
                key *= z[ord(strs[i][j]) - ord('a')]
            dict[key] = dict.get(key, []) + [strs[i]]

        return list(dict.values())

```

怎么样，相比大家也切切实实感受到键设计的艺术了吧。

## Day11 宝石和石头

给定字符串J代表石头中宝石的类型，和字符串S代表你拥有的石头。S中每个字符代表了一种你拥有的石头的类型，你想知道你拥有的石头中有多少是宝石。

J中的字母不重复，J和S中的所有字符都是字母。字母区分大小写，因此"a"和"A"是不同类型的石头。

示例 1：

输入: J = "aA", S = "aAAbbbb" 输出: 3

示例 2:

输入: J = "z", S = "ZZ" 输出: 0

注意:

S 和 J 最多含有50个字母。

J 中的字符不重复。

```
class Solution:  
    def numJewelsInStones(self, J: str, S: str) -> int:
```

石头与宝石，错过的星友查看作业题目：<https://t.zsxq.com/F6eMfIE>

我看星友们提交的答案都很不错，主要有两种：

## 方法 1 暴力破解：

```
#石头与宝石——暴力破解  
class Solution(object):  
    def numJewelsInStone(self,J:str,S:str)->int:  
        count = 0  
        for s in S:  
            if s in J:  
                count += 1  
        return count
```

Pythonic一点：

```
class Solution(object):  
    def numJewelsInStone(self,J:str,S:str)->int:  
        return sum(s in J for s in S)
```

时间复杂度是  $\text{len}(S) * \text{len}(J)$

@Leven 还给出一种更加 Pythonic 的写法，利用 **sum**生成器的方法，好处节省内存，大家不妨留意下。

## 方法 2 哈希表

此解法的时间复杂度是  $\text{len}(S) + \text{len}(J)$

```
#宝石与石头，哈希表
class Solution(object):
    def numJewelsInStone(self, J:str, S:str) -> int:
        dict = {}
        sum = 0
        for s in S:
            dict[s] = dict.get(s, 0) + 1
        for j in J:
            sum += dict.get(j, 0)
        return sum
```

根据字符哈希为出现次数，然后挑选出出现在宝石串的字符个数。

相信大家这10天的训练，对算法形成一个大概的时间复杂度概念、

等讲完链表后会单独有一天讲时间复杂度分析。

# Day12 删 除 链 表 节 点

## 12.1 链表基础

链表这个数据结构是绝对的重中之重，它是线性一维结构的代表，插入和删除具有  $O(1)$  复杂度，也是后面讲二叉树等的基础。

因此，链表务必要掌握。

链表又很容易出错，所以只能掌握它的基本定义后，多做练习巩固它。

链表的基本结构：

### 指针



Red 是最后 1 个数据，所以 Red 的指针不指向任何位置。

这就是链表的概念图。Blue、Yellow、Red 这 3 个字符串作为数据被存储于链表中。每个数据都有 1 个“指针”，它指向下一个数据的内存地址。

链表的顺序访问方法：

### 顺序访问



因为数据都是分散存储的，所以如果想要访问数据，只能从第 1 个数据开始，顺着指针的指向一一往下访问（这便是顺序访问）。比如，想要找到 Red 这一数据，就得从 Blue 开始访问。

## 12.2 作业分析

链表中如何删除一个节点？

比如，插入节点**Green**

05



Green

如果想要添加数据，只需要改变添加位置前后的指针指向就可以，非常简单。比如，在Blue和Yellow之间添加Green。

请根据上面的提示，补全下面代码：

```
# 单向链表的定义
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
```

```
class Solution:  
    def deleteNode(self, node):  
        # 补全代码  
        #  
        #
```

这是leetcode 237 题，写完后建议去验证一下，链接：

<https://leetcode-cn.com/problems/delete-node-in-a-linked-list/>

## 12.3 删 除链表的节点

删除链表的某个节点 `target`

下面我们看下星友金金的精彩回答，从链表的建立到删除指定节点，比较全面。

首先建立 `1->7->3->6->5->6` 的链表，注意查看链表的迭代过程，对于习惯了 `i+=1` 迭代的朋友，可能对链表的迭代逐渐熟悉起来： `tmp=newNode`

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

class Solution:
    def deleteNode(self, head, target):
        cur = head
        while cur:
            if cur.val == target:
                cur.val = cur.next.val
                cur.next = cur.next.next
                return head
            else:
                cur = cur.next
        return head

if __name__ == "__main__":
    original_list = [1, 7, 3, 6, 5, 6]
    target = 5

    # create singly-linked list
    head = ListNode(None)
    tmp = head
    for i in original_list:
        newNode = ListNode(i)
        tmp.next = newNode
        tmp = newNode
    head = head.next

    solution = Solution()
    res = solution.deleteNode(head, target=target)
    # 打印结果
    while res:
        print(res.val)
        res = res.next
```

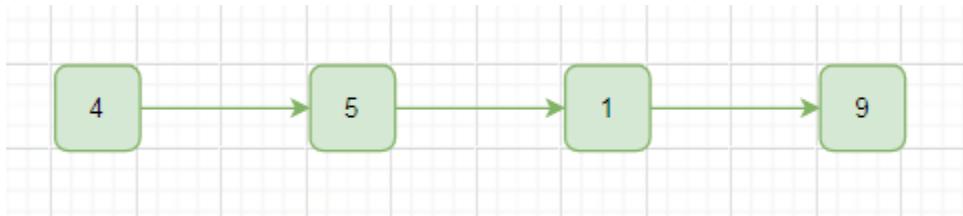
上面的删除是根据 `val` 判断删除的节点，这与 Day12 题目删除节点 `node` 略有区别（题目中

target直接就是一个节点指针)

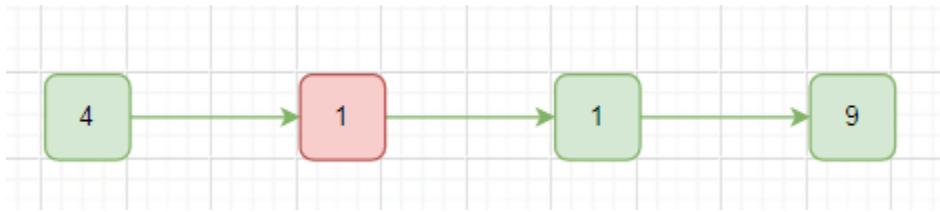
回答这个题目，注意leetcode中237题，是删除非尾部节点。

```
class Solution:  
    def deleteNode(self, node):  
        node.val = node.next.val  
        node.next = node.next.next
```

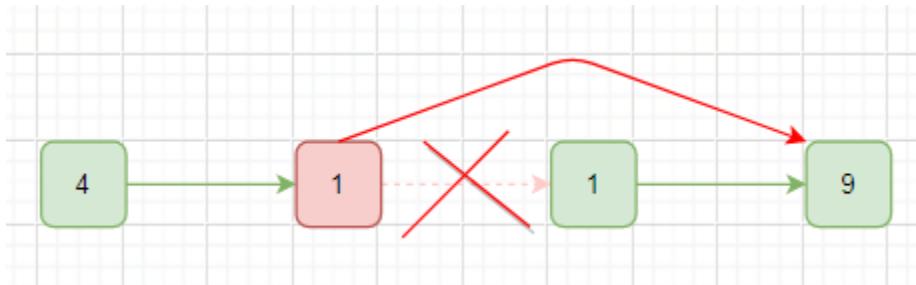
举例，删除节点 5：



第一步：



第二步：



但是，星友们想过没有，为什么要题目要敲掉删除的是非尾部节点。

这是因为，如果删除尾节点，`node.next`就不适应上面的操作步骤，必须要找到倒数第二个节点才可，但是对于单向链表只知道当前删除节点 `node`，是无法定位到的。所以才需要这个限制条件。

对于初次接触链表的朋友，可能觉得使用起来比较别扭，可能需要勤加练习才行和多多理解。

# Day13 访问链表第*i*个节点

依然考虑到新接触链表的朋友不习惯使用它，本次作业还是强化下链表的基本操作。

```
# 定义单链表
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
```

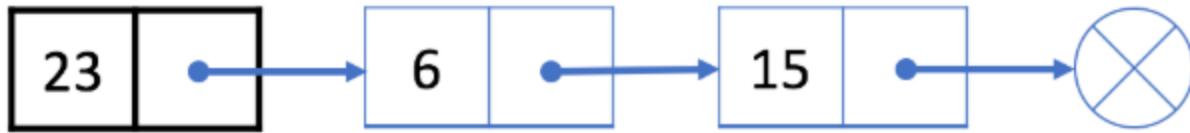
```
def getiNode(head, n, i):
    """
    head: 链表头结点
    n: 链表长度
    i: 返回的第 i 个节点 nodei
    """
    #
    #补全代码
    #
```

## 13.1 分析总结

获得长度为 `n` 头节点为 `head` 链表的第 `i` 个节点，代码如下：

```
def getiNode( head, n, i):
    if i < 0 or i > n: # 检查边界条件
        raise IndexError
    cur = head
    for _ in range(i):
        cur = cur.next
    return cur
```

对于如下链表，值为23的节点为表头，它的指针域取值是下一个节点的指针，依次类推，最后串成一条线：



这是链表这个数据结构的特点。

这两天训练链表时，也有一些星友实际上完成了列表转化为链表的任务，代码如下所示：

传入 `lst` 转化链表返回 `cur_Node`：

```
#访问链表第i个节点
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

# 单节点类
class SingleLinkedList:
    def __init__(self):
        self.head = ListNode(None)
    def createLinkedList(self, lst):
        cur_Node = self.head
        tmp = cur_Node
        for data in lst:
            link_list = ListNode(data)
            tmp.next = link_list
            tmp = link_list
        cur_Node = cur_Node.next
    return cur_Node
```

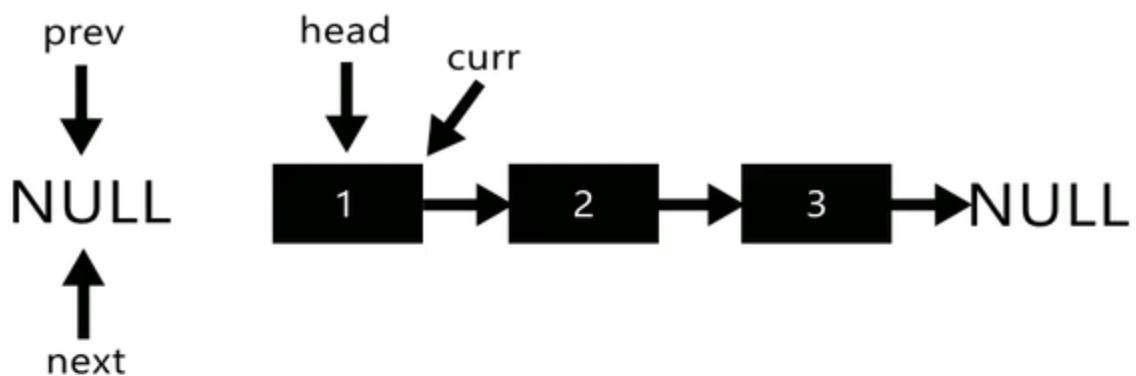
以上代码实现有一个巧妙之处：`self.head=ListNode(None)`，设置一个空的哨兵表头，并使 `tmp` 和 `cur_Node` 分别指向这个空表头，`for` 中依次创建一个节点，`tmp` 完成链表串联任务。

遍历完成后，`cur_Node.next`便是真正的链表表头。

初次接触链表的星友，不妨多理解一下，慢慢就会习惯链表这种数据结构。

## Day14 反转单链表

反转单链表检验我们是否能够真正灵活使用它，也是面试频频被问道的一个题目。



```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

例如反转上面单链表的方法之一：

1. 首先，我们将黑色结点的下一个结点（即结点 6）移动到列表的头部：



2. 然后，我们将黑色结点的下一个结点（即结点 15）移动到列表的头部：



黑色结点的下一个结点现在是空。因此，我们停止这一过程并返回新的头结点 15。

根据以上提示，请补全下面代码：

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

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        #
        #补全代码
        #
```

写完后去这里验证：

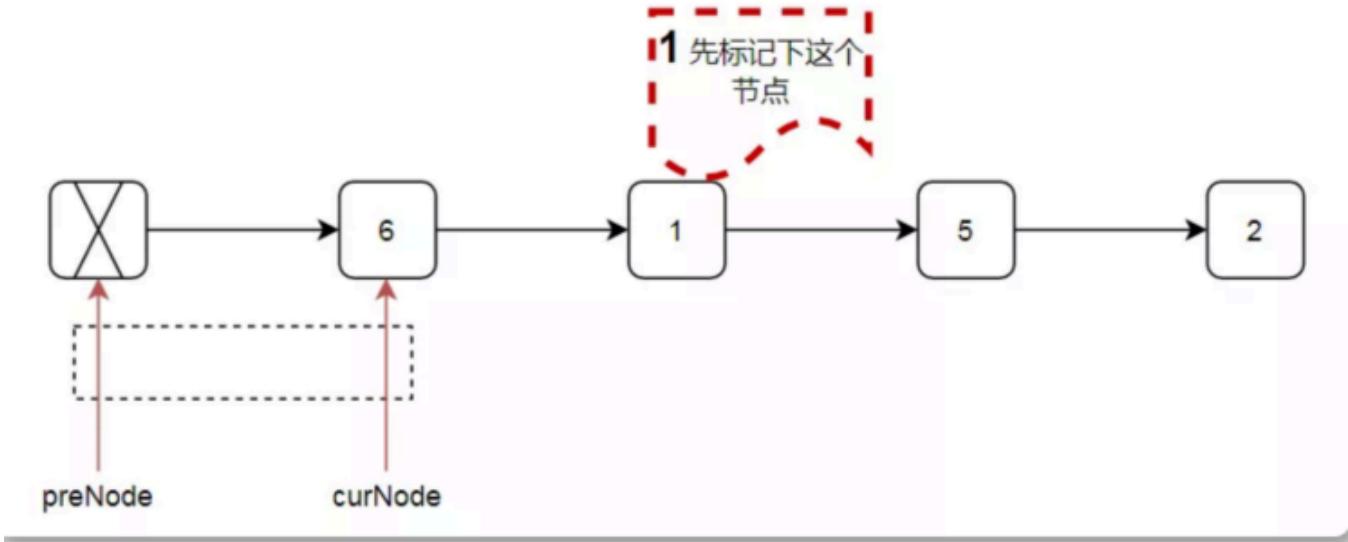
<https://leetcode-cn.com/explore/learn/card/linked-list/195/classic-problems/750/>

## 14.1 反转单链表

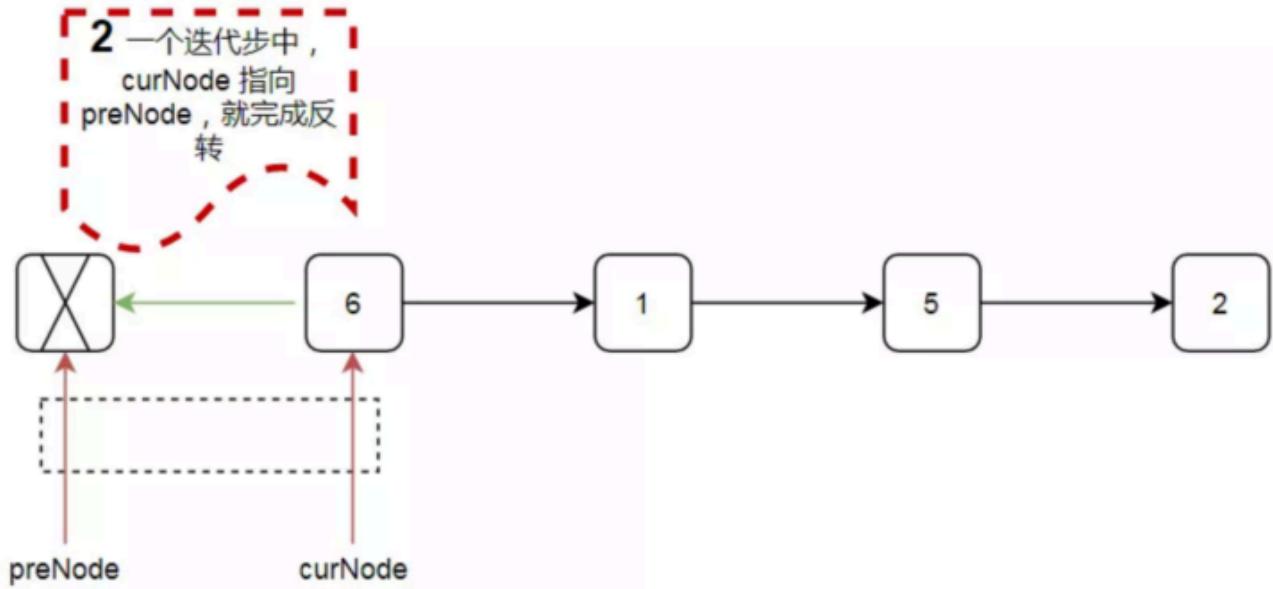
反转链表的方法：迭代和递归。

### 1 迭代法

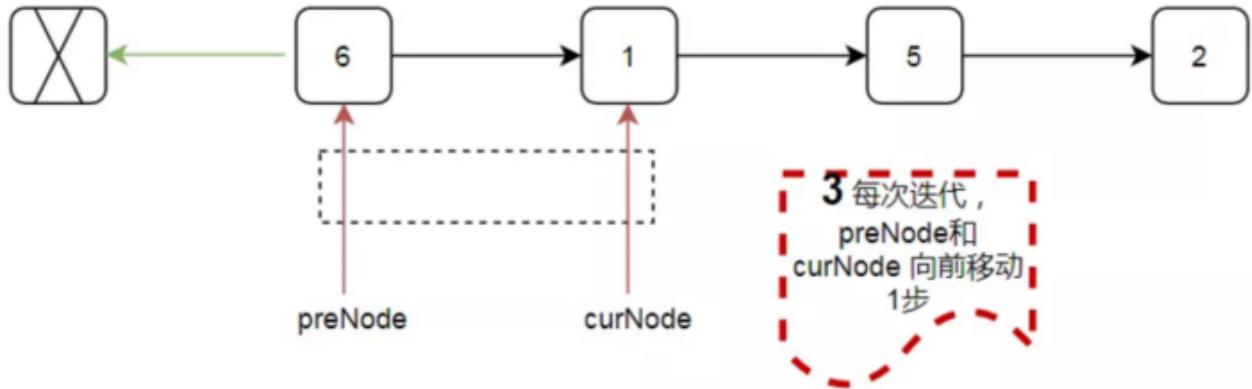
Step 1 先标记下这个节点



Step 2 一个迭代步中，curNode 指向 preNode，便完成当前迭代步的反转



Step 3 每次迭代，preNode 和 curNode 向前移动 1 步



以上步骤归结为如下代码（代码来自我的室友 Leven）：

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        preNode = None
        curNode = head
        while curNode:
            next = curNode.next
            curNode.next = preNode
            preNode = curNode
            curNode = next
        return preNode
```

## 2 递归法

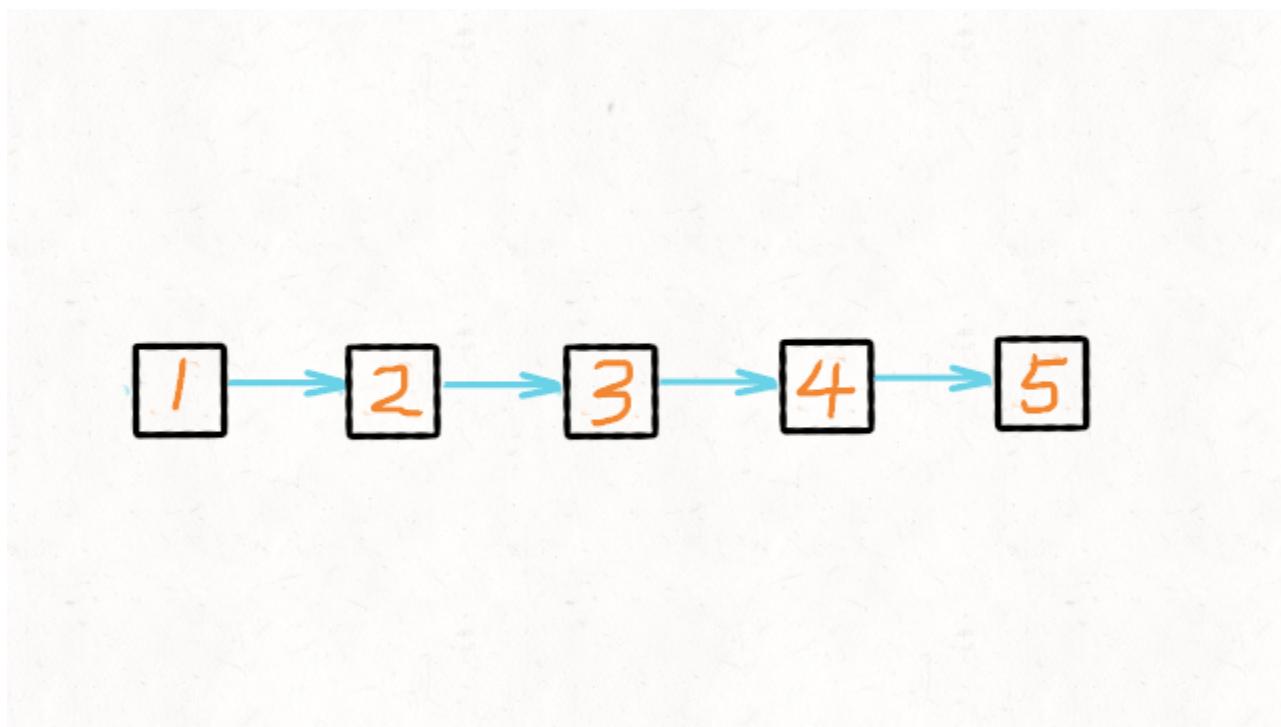
首先反转自 `head.next` 后的链表，`node` 此时指向翻转后链表的头部；最后将 `head.next` 节点链接到 `head` 节点，最后的最后将 `head` 节点的 `next` 域置为 `None`，因为 `head` 是终点了。

代码如下：

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
            return head
        next=head.next
        reversehead=self.reverseList(next)
        next.next=head
        head.next=None
        return reversehead
```

### 3 尾递归

还有一位星友诚Slime提供了第二种递归思路，gif 演示如下：



代码如下：

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
            return head
        cur=self.reverseList(head.next)
        head.next.next=head
        head.next=None
        return cur
```

# Day15 程序员为什么学算法

我们思考和讨论如下话题：

程序员一定要学算法吗，说出你为什么要学习算法？

可以参考：<https://www.zhihu.com/question/290268306>

## 为什么要学算法

近来经常有朋友问，程序员需要学算法吗？为什么需要学算法？不会算法也能找个Java开发岗位软件所以就别浪费时间了。如果真要学，算法感觉很高深，需要数学，可是我数学不好，所以放弃它吧？

面对这些疑问，我昨日在星球里留作业想听听星友们怎么看，程序员为什么要学习算法。来，一起看看他们的回答。

回答1

风岚风清玉琢玉 昨天 23:00

老实说，学算法的收获里算法（方法）固然很重要，但思维的成型其实更重要。

学好算法，在不知不觉中，你甚至会有软件工程的优化意识，或者说再学软件工程会有良好的反馈循环。

简单点讲，也许你思维敏捷数学逻辑敏感，能自己想出很多算法，但是还有一些算法是几代人的

努力，顿悟这种算法可能还是有点难度的。好吧，就算大佬十分天才，那么掌握算法后，再去动用这份天分不香吗？

不要浪费自己的天赋，早点站在巨人的肩膀上，早点站得更高。

回答2

eternal 昨天 23:05

语言只是一种外壳，而算法是一个程序的灵魂，一个好的算法不仅可以实现具体的功能，更是对人的解题方法，思维境界的全面提升。

回答3

龙人浚 昨天 23:18

程序员学算法的意义可能在于以后面对待解决问题有更多的思路和方法，提高自己解决问题的能力。

回答4

infrared62\* 昨天 23:30

Day 15 程序员一定要学算法吗，说出你为什么要学习算法？

程序员不一定要学算法，软件行业现在已经是，即使不学算法，也可以做出一些东西出来的样子。

而我选择学，是因为计算机的世界里充到到处都是算法，我想通过算法去探求这个世界的真理。

回答5

Jack闹 昨天 23:31

day15：为什么要学习算法

通过最近一段时间的学习，以及之前参加数学建模比赛、写论文的经历，我发现，算法当然遇到问题再有针对性地学习，但是这样的学习总有种空中楼阁的感觉，学习的效率并不高，最怕的是遇到问题根本找不到该用什么算法，或者找到太多也不会判断哪个更好。

所以，我感觉算法还是需要一定基础的，需要一定系统学习的积累与实践，这样才能更好地触类旁通，选择好的算法解决问题。

希望通过算法刷题向各位星友学习，共同进步！

回答6

勇往直前 18 小时前

Day15 程序员为什么要学习算法

对于在校生而言，强大的算法能力能助你斩下个竞赛的奖项，给你的评优保研增加砝码。

对于求职者而言，算法能力能成为你拿到优质offer的敲门砖。字节跳动应该是很多人梦寐以求offer吧，他们的面试就非常注重算法能力的考察（当然，绝大多数大厂都是这样）。

回答7

Okra 15 小时前

day 15

学习算法一方面是为了国外面试，另一方面是为了锻炼自己的计算机思维，从而在复杂的情况下更好地分析。

回答8

聂磊 11 小时前

day15 学习算法为了掌握程序基本思想、基本方法。

回答9

徐嫄 11 小时前

打卡015

我学算法为了多用用脑子

回答10

北方 10 小时前

打卡Day15

学习算法能够开拓眼界，更好的应用计算机现有的资源完成遇到的问题。

回答11

小门神 10 小时前

培养逻辑思维能力，锻炼智力。

回答12

诚Slime 10 小时前

day15打卡：讨论程序员为什么要学习算法

作为一个科班出身的研一学生，本科的时候没有把重心放在学业上，也没有深究过算法，真的很后悔以前没有去打ACM，这不仅能提升自己的思维，能力以及对问题思考的深度，一种看见问题快速建模的能力，真的很重要(工作，升学也有很大帮助)。

现在自己读研期间也产出了论文，如果你算法水平高，对于数据预处理来说这个过程会变快质量也会上升，以及模型架构都会提出更好的解决方案，也可以有更优的实现。

算法是对你核心能力的提升，可能不是立竿见影，但也是潜移默化的，你感觉不到，也不妨碍他的存在！

愿不负韶华！

回答13

莱布妮子 9 小时前

Day15（讨论程序员为什么要学习算法）打卡：因为有时编程问题并不是都能用简单的API实现，不能实现的时候就需要自己造轮子，造轮子时算法知识就是必不可少的。

回答14

馨爸 9 小时前

Day15作业题：讨论程序员为什么要学习算法

前面的内容已经讨论了算法的定义

算法说白就是解决问题的一种逻辑法则，时而精简时而信手拈来，但每一个算法都有不同的思维精髓。

其实，不管是程序员亦或是数据处理亦或是人生，都需要学习算法、学习算法思维，使自己变得更清晰、更有逻辑。优秀的算法其速度及逻辑可以精妙绝伦...

诸如前面星友所说，坚持每天学习一点，不负韶华。

回答15

川顺 8小时前

day15打卡，算法可以练习思维，在特定的大量数据计算的场景可以用算法来解决问题，加快程序运行速度！

总的来说，算法用处还是挺大的，可以让你慢慢了解程序的本质，当家才知柴米油盐酱醋茶，所以你在设计算法的时候才会想着如何节省空间，加快速度，而不是敲完代码就不管了！

回答16

\u5f20\u5111 8小时前

学习算法是为了更有效的解决问题

回答17

张=小红= 7小时前

【打卡】第十五天。算法对我这个数学专业的学生来说，就是一种解决问题的方法。

方法的重要性在于能够在时间性或者空间性上面比起常规解法有领先。

作为学生的话，学习算法更多就是让我在以后就业先人一步，毕竟你说你会python，然后你就只会调个库，算法什么都不会，那只能算个业余爱好者吧。

回答18

冬云瑞雪映松竹 7小时前

day15 打卡

算法就像武功中的内力。内功深厚才能习得上乘武功。-

回答19

tate 6 小时前

算法学的好能够提高解决问题的速度及效率！

回答20

金金金 5 小时前

Day-15 为什么要学算法？

学习算法更多的是为了培养解决某类问题的算法思想。

可以说，在生活遇到的每一个具体问题就是不同的，我们需要看清问题的本质，学会将问题逐步分解，在分解的过程中就可以慢慢思考出问题的解决方案。算法更像是一种方法论（或者说内功心法）。

回答21

箱子 5 小时前

感觉算法是一种思想吧，要不然只能做一些“体力活”，还是为了提高自身竞争力。

回答22

LFeng 3 小时前

day15 #思考#

程序员为什么要学习算法 咖啡 学习算法有助于开拓思维，更有利子解决实际问题

回答23

张德春 1 小时前

Day15

培养规范的处理问题的思维

回答24

闪～～星～ 1 小时前

感觉学习算法是为了培养对问题的敏感性，提供一种思维方式。

回答25

Bruce 10分钟前

算法确定了程序操作的具体工作流程，是程序等我灵魂。

---

以上就是截止今晚7点星友回答，大家一致认为学习算法为了提升思维，更高效的解决问题，在造轮子时懂算法写出的代码更高效，更有竞争力，如果算法一点不懂，说的极端点只能做一些“体力活”。

其中回答17一针见血：方法的重要性在于能够在时间性或者空间性上面比起常规解法有领先。作为学生的话，学习算法更多就是让我在以后就业先人一步，毕竟你说你会python，然后你就只会调个库，算法什么都不会，那只能算个业余爱好者吧。

## Day17 算法好坏度量大O记号

我们都知道描述机器学习算法好坏的指标常用什么ROC，AUC，精确率和召回率等，机器学习算法也是算法，除了这些，描述算法好坏的一个必备基础指标：时间复杂度，衡量时间复杂度的常见方法：大O记号。

今天的作业题：  
 $\sqrt{4n^4 + 3n^2 + 2n + 10000}$  在大O记号下，等于多少？是  $O(2n_2)$ ？还是其他？你是怎么得出来的？

一般按照步数严格计算得到  $T(n)$ ， $n$  为计算规模，但是  $T(n)$  往往比较复杂，比如为：

$\sqrt{4n^4 + 3n^2 + 2n + 10000}$ ，如果每一个算法都这么去计算衡量效果，这显然不利于交流，所以 1894 年 Paul Bachmann 提出了大O记号。

### 1 数学定义

$T(n) = O(f(n))$ ，进一步表达为对于任意  $c > 0$ ，当计算规模  $n$  足够大时，都满足下面式子：

$$T(n) < cf(n)$$

根据如上定义，我们可以看出来，大O标记下的 $f(n)$ 不关心系数 $c$ ，并且 $f(n)$ 是对T(n)的一种放大，由此得到是算法的最坏情况，这也是我们最关心的，一个算法的最坏情况是什么。

## 2 大O记号

$$T(n) = \frac{1}{\sqrt{4n_4 + 3n_2 + 2n + 10000}}, \text{ 大O记号下等于多少?}$$

根据定义，我们这样放大 $T(n)$ ：

$$\begin{aligned} T(n) &< \frac{1}{\sqrt{4n_4 + 3n_4 + 2n_4 + 10000n_4}} \\ &< \frac{1}{\sqrt{104n_4}} \\ &< \frac{1}{\sqrt{104}} * n_2 \end{aligned}$$

根据定义只要满足 $T(n) < cf(n)$ ,  $T(n) = O(f(n))$ , 所以 $T(n)$ 为 $O(n_2)$ ，即最坏时间复杂度为 $O(n_2)$

## 3 基本原则

参考星友 聂磊 2小时前的回答：

Day16（计算时间复杂度）学习时间复杂度的基本原则

1. 只有常数项，认为其时间复杂度为 $O(1)$
2. 顺序结构，时间复杂度按加法进行计算
3. 循环结构，时间复杂度按乘法进行计算
4. 分支结构，时间复杂度取最大值
5. 判断一个算法的时间复杂度时，只需要关注最高次项，忽略最高次项的系数，且其它次要项和常数项也可以忽略
6. 一般所分析的算法的时间复杂度都是指最坏时间复杂度

忽略次要项和常数项，以及最高项的系数，得出时间复杂度为 $n_4$ 开根号，即 $O(n^2)$

请列举时间复杂度分别为

$O(1)$ ,

$O(\log n)$ ,

$O(n \log n)$ ,

$O(n)$ ,

$O(n_2)$ ,

$O(n!)$  的算法

写出几种常见复杂度对应的算法，星友们给出的答案都很准确，在这里参考星友聂磊的答案：

时间复杂度：

$O(1)$  常见操作：哈希查找，数组取值，常数加减乘除

$O(\log n)$ : 二分查找

$O(n)$  计算数组中所有元素和，最大，最小

$O(n \log n)$ ：归并排序，堆排序，希尔排序，快速排序

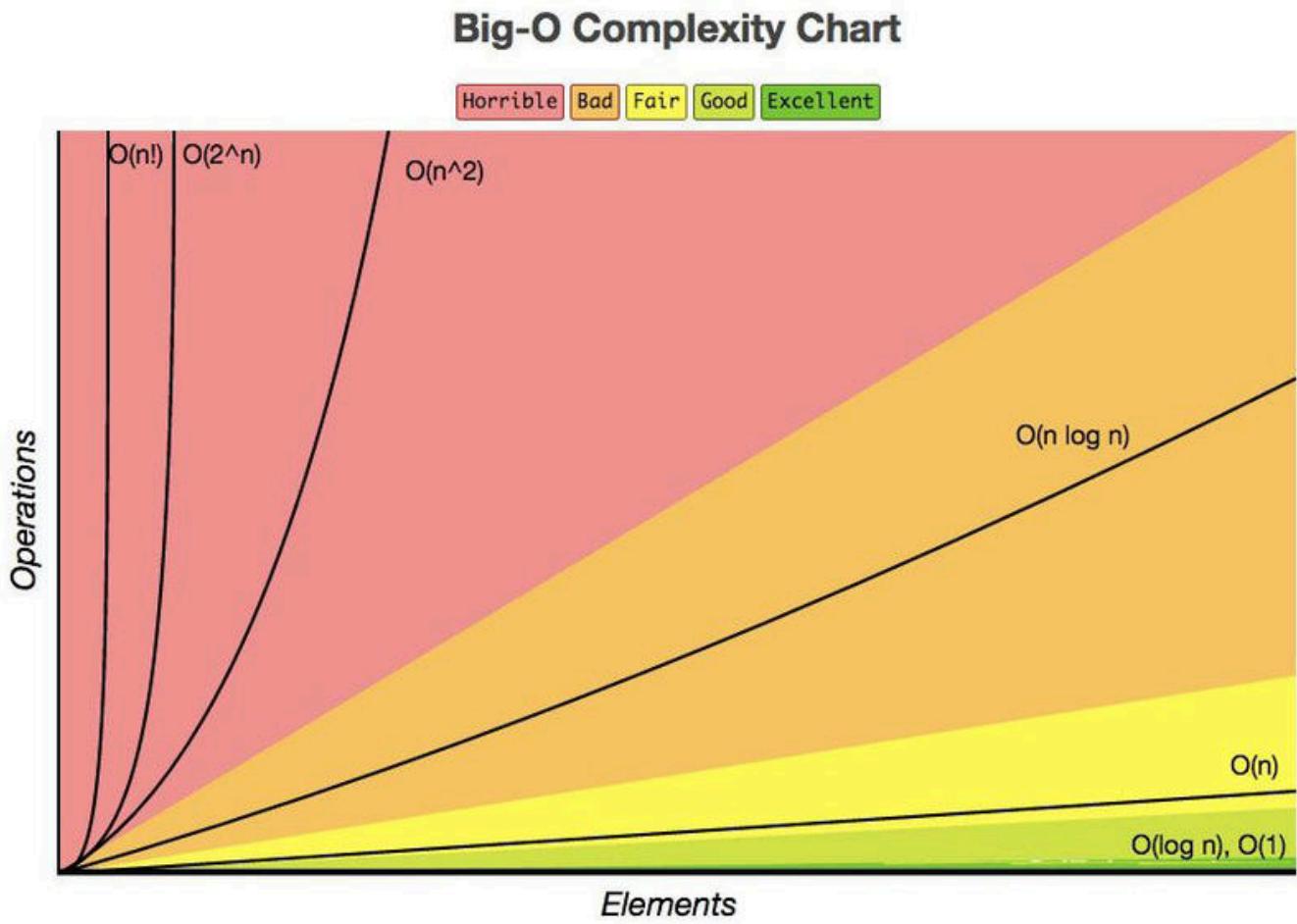
$O(n^2)$ ：冒泡，选择排序

$O(n_3)$ ：三元方程暴力求解

$O(2_n)$ ：求解所有子集

$O(n!)$ ：全排列问题。比如：给定字母串，每个字母必须使用，且一次组合中只能使用一次，求所有组合方式；另外还要经典的旅行商 TSP 问题

下面这幅图太直观了：



其中，复杂度为： $O(2_n)$ ， $O(n!)$  是难解的问题， $O(n \log n)$ , $O(n^2)$ 都是可以接受的解， $O(n)$ , $O(\log n)$  的解是梦寐以求的。

## Day18 二分查找

二分查找算法，binary search algorithm，也称折半搜索算法、对数搜索算法

它的使用前提：是一种在有序数组中查找某一特定元素的搜索算法。

请补全下面二分查找算法：

```
# 返回hkey在数组中的索引位置
def binary_search(arr, left, right, hkey):
    """
    arr: 有序数组
    left: 查找区间左侧
    right: 查找区间右侧
    hkey: 带查找的关键码
    备注: left, right 都包括在内, 找不到返回 -1
    """

```

要求补全上述代码

注意事项：

1. 迭代中，一定注意while判断中等号问题
2. 二分查找的代码还是很容易写出bug

## 迭代二分查找

代码参考星友 Leven：

```
def binary_search(arr, left, right, hkey):
    while left <= right:
        mid = left + (right-left) // 2

        if arr[mid] == hkey:
            return mid
        elif arr[mid] > hkey: # 严格大于
            right = mid - 1
        else: # 此处严格小于
            left = mid + 1

    return -1 # 表示找不到
if __name__ == "__main__":
    sorted_list = [1, 2, 3, 4, 5, 6, 7, 8]
    result = binary_search(sorted_list, 0, 7, 4)
    print(result)
```

# 递归二分查找

```
def binary_search(arr, left, right, hkey):
    if len(arr) == 0:
        return -1

    if left > right:
        return -1

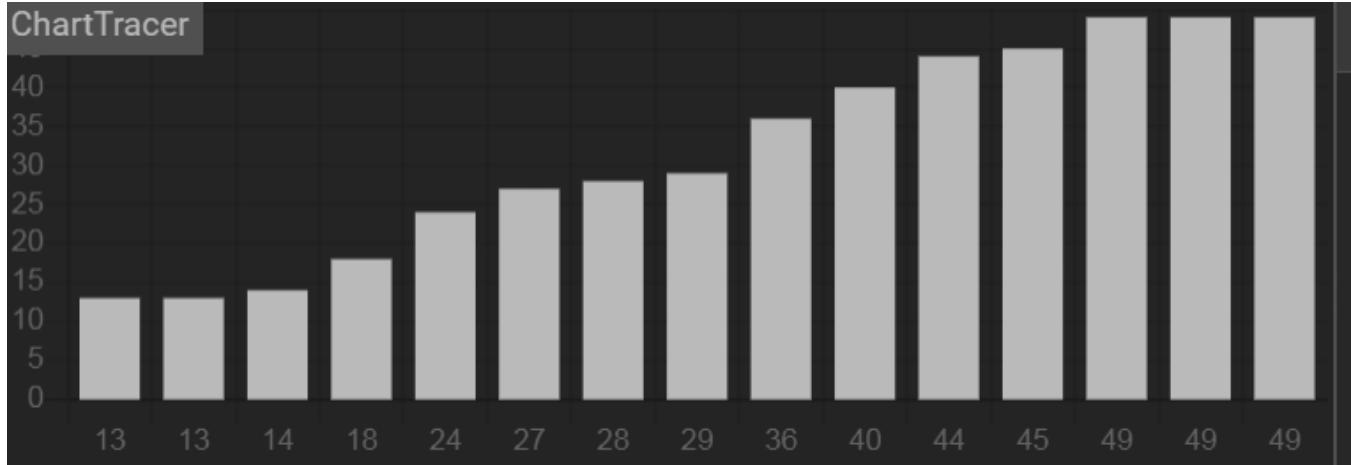
    mid = left + (right-left) // 2

    if arr[mid] == hkey:
        return mid
    elif arr[mid] < hkey: # 严格小于
        return binary_search(arr, mid+1, right, hkey) # 折半
    else:
        return binary_search(arr, left, mid-1, hkey)

if __name__ == "__main__":
    sorted_list = [1,2,3,4,5,6,7,8]
    result = binary_search(sorted_list, 0, 7, 4)
    print(result)
```

## 更多演示动画

能找到关键码：

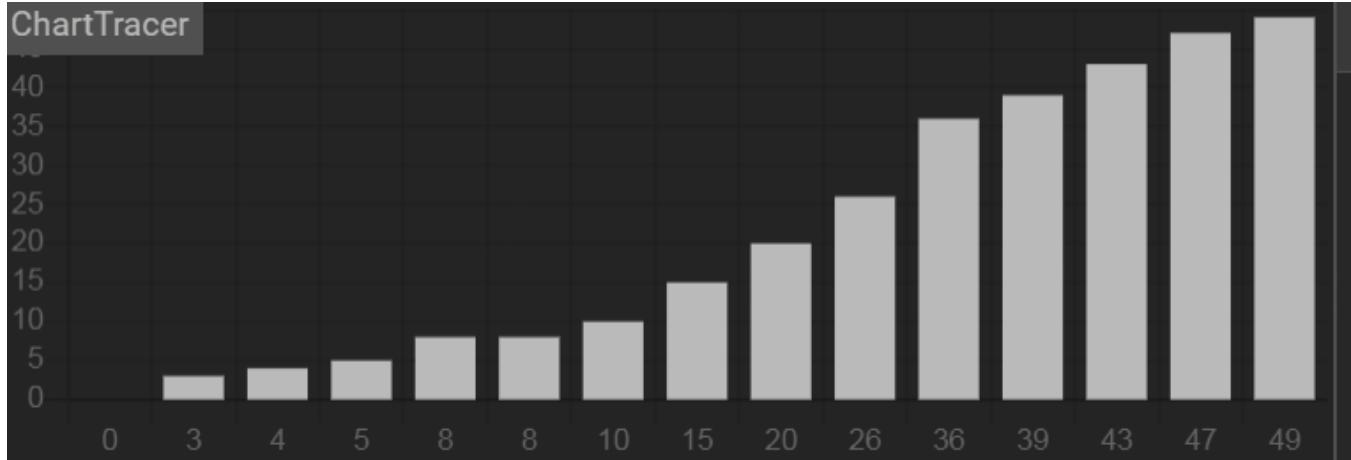


**Array1DTracer**

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	13	13	14	18	24	27	28	29	36	40	44	45	49	49	49

**LogTracer**

不能找到关键码：



Array1DTracer

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3	4	5	8	8	10	15	20	26	36	39	43	47	49

LogTracer

## Day19 合并两个有序数

合并两个有序数组 left 和 right：

```
def merge(left,right):
    #补全代码
    #
    return temp
```

## 分析过程

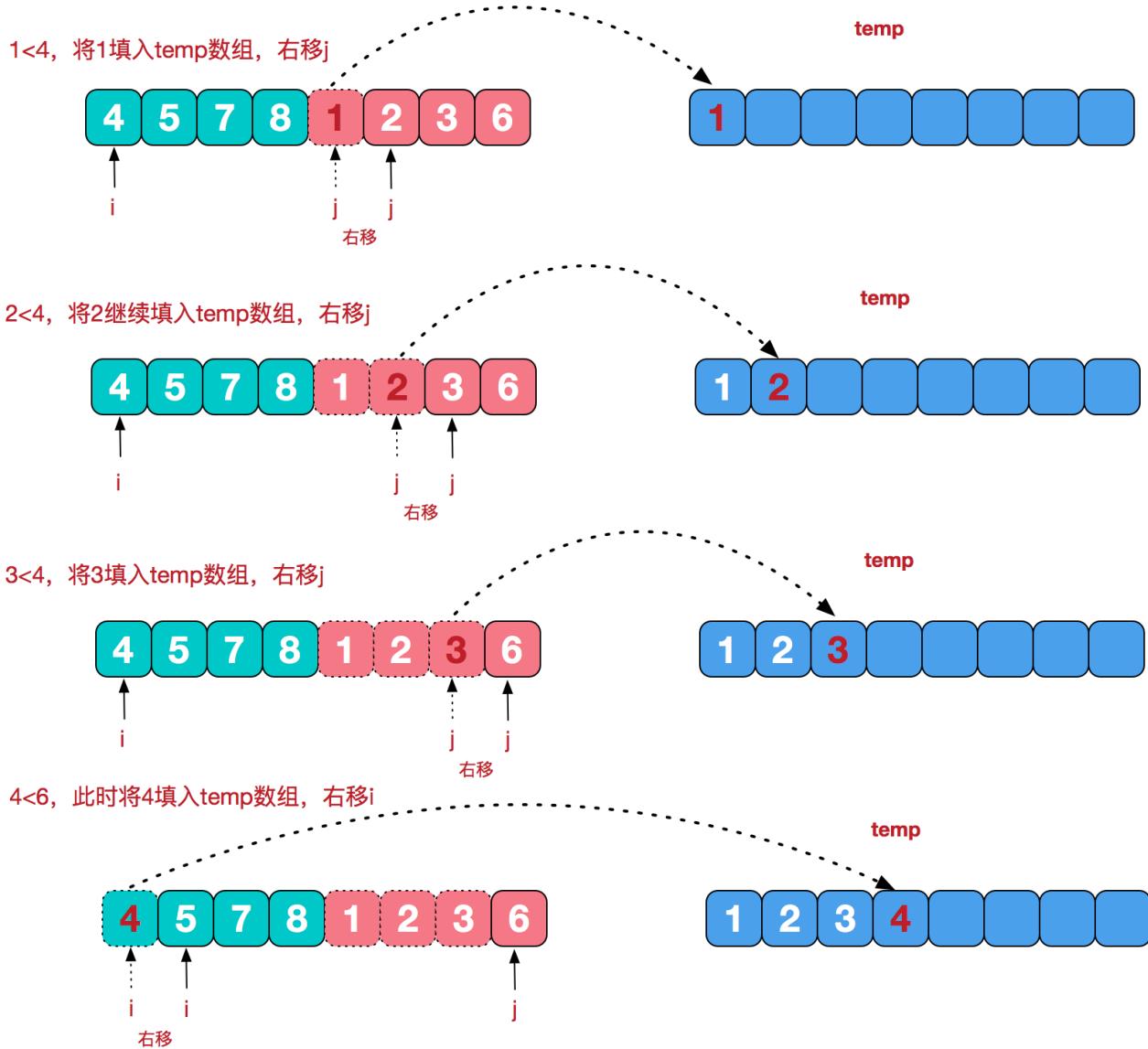
合并两个有序数组

利用两个数组原本已经有序的特点：

```
def merge(left, right):
    i = 0
    j = 0
    temp = []
    while i <= len(left) - 1 and j <= len(right) - 1:
        if left[i] <= right[j]:
            temp.append(left[i])
            i += 1
        else:
            temp.append(right[j])
            j += 1
    temp += left[i:] + right[j:]
    return temp

print(merge([1,3,4],[2,3,3]))
```

思路可参考示意图：



## Day20 归并排序算法

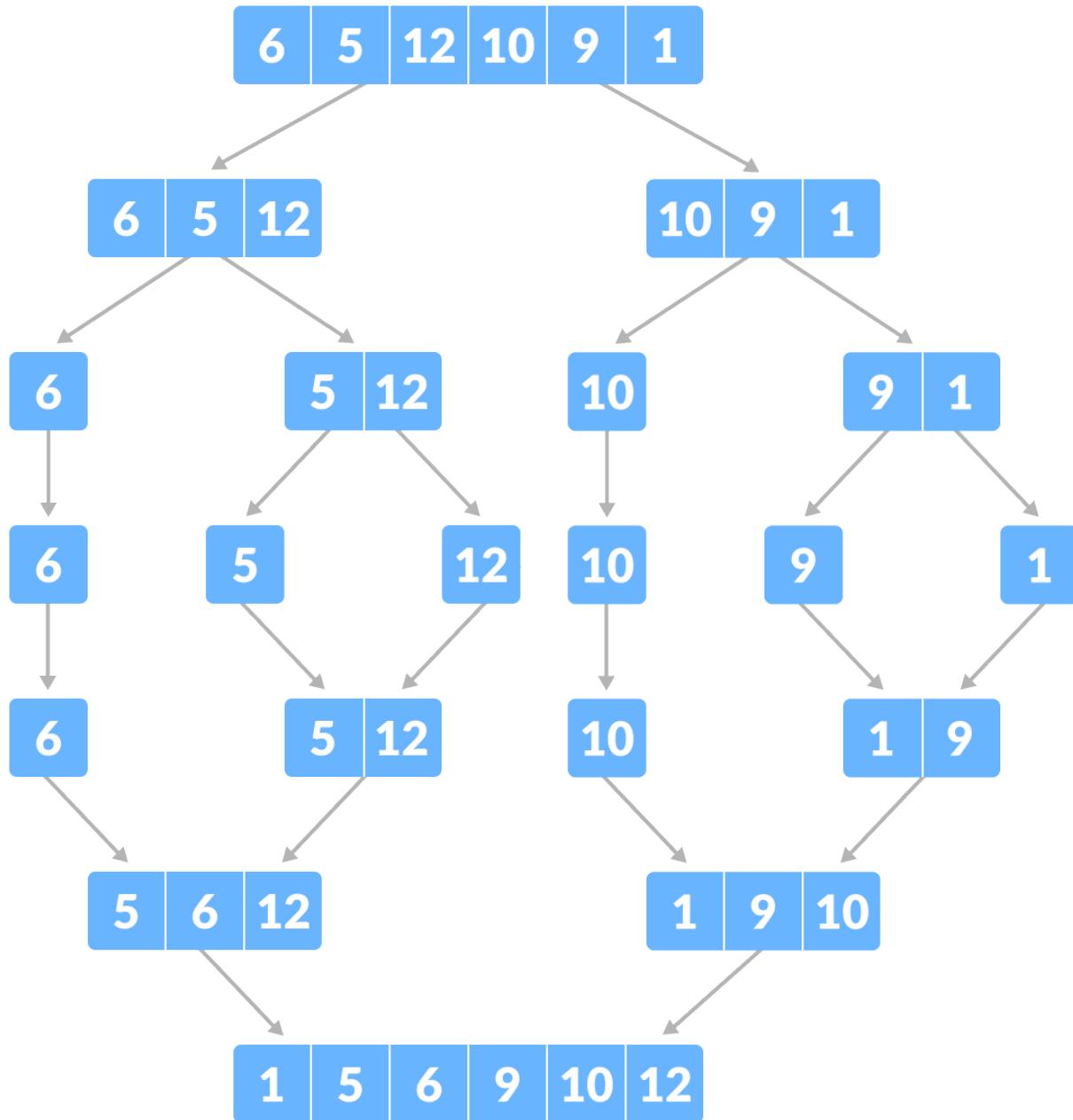
归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的分治（divide-and-conquer）策略（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的答案"修补"在一起，即分而治之）。

归并排序算法的核心正是 Day 19 的合并两个有序数组，补全如下代码：

```
def merge_sort(lst):
    #
    #
    #
    return lst_sorted
```

归并排序两阶段：

先分，直到长度1，然后再合：



归并排序的详细讲解，可参考：<https://www.programiz.com/dsa/merge-sort>

## 分析过程

归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的分治（divide-and-conquer）策略（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之）。

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2 # Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+= 1
            else:
                arr[k] = R[j]
                j+= 1
            k+= 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i+= 1
            k+= 1

        while j < len(R):
            arr[k] = R[j]
            j+= 1
            k+= 1
```

验证调用：

```
# Code to print the list
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end = " ")
    print()

# driver code to test the above code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print ("Given array is", end ="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end ="\n")
    printList(arr)
```

归并排序的详细讲解，可参考：

<https://www.programiz.com/dsa/merge-sort>

## Day21 21天刷题总结

很多星友都一直坚持到现在，21 天整，都说 21 天养成一个习惯，据此推断，相信你们养成了刷题的习惯~~

# 刷题总结



Yaya Guo

1小时前

作业

...

#DAY21

1. 不断学习积累，善于总结思考
2. 每个人都有自己的短板，善于发现自己的缺点，做到及时查缺补漏
- 3.按照星球计划进行学习，不要掉队
- 4.老师安排的学习节奏也很好，算法不仅仅写代码，更多的需要反复琢磨思考
- 5.希望经过一段时间的学习，自己的代码能力，思维方式有所改变

| 查看作业题目 >



闪~星~

1小时前

作业

...

21天了?都没啥感觉，时间过得真快。在星球里学习并不一定需要学习什么高深的东西，个人感觉只要是找到了可以一起学习的同伴，我认为小组学习大于个人埋头苦干，坚持就是胜利。

| 查看作业题目 >



查看详情 >



孙颖颖颖颖颖颖

1小时前

Day21 🤘

作为一个本硕非科班(本科电子科学技术，硕士信通) 出身的人 本身自己也是有一些拖延症(直白点就是有点懒🤣) 这21天的刷题 有很多朋友一起 还能看到很多大神的解题思路 很受启发 最近也一直在刷力扣 复习ML和DL的一些面试题 过得很充裕 从一个算法小白 到现在看到题目能有自己的思路 相信坚持下去一定会让我在下半年的秋招找到一份满意的工作💪 坚持！向大佬们学习💪



金金金

1小时前

作业



Day-21 刷题总结，心得体会

学习在于积累与坚持，个人认为执行力远比计划要重要。做了计划但没有执行，与没做计划相比不会有有多大区别。

坚持每天刷题与积累，慢慢地养成习惯，随着时间的推移，相信总会有好结果。

| 查看作业题目 >



查看详情 >



箱子

2小时前



#Day21

每一天的作业都会想一想，但是有可能还是思路上差一些，有时候就算想到了，可能也不是很pythonic，有时候也是网上找找，希望能够学到更多的东西，让自己的想法更多一些。



查看详情 >



B+AI

2小时前

作业

#算法刷题# day21，我参加这个的目的是为了证明自己可以坚持做一件事很久，我不是计算机专业的，学岩土的。算法之前没接触过，这几天下来，真是惊叹于算法的精妙，真的像读一本好书，我也希望自己放下负担，享受拓展思维的愉悦！

| 查看作业题目 >



聂磊

2小时前

作业

day21本人非计算机专业，纯粹的业余爱好，每天地铁上，手机刷题学习算法和python编程，感觉对数据结构和算法有了新认识，群里牛人很多，学习了很多新知识和新方法，顺道也辅导下小朋友编程，一举多得。

| 查看作业题目 >



查看详情 >



周祺华-马周

2小时前

作业

很喜欢这样的形式，就是最近加班严重，回家就已经12点多了，还要写报告，所以压力山大

平常的工作量，这个难度，是可以完成的

希望这阵子过后，可以赶上进度

| 查看作业题目 >



诉说

2小时前

...

时间还真是挺快的，已经21天了，其实我挺擅长坚持的，所以我也谈不上是不是有习惯了，只是每天回家必须要做的事情就是做作业。忙碌但是不会放弃。

因为现在是在做数据挖掘，而且本身是转行的，计算机编程基础确实差一些，一直想要补，总感觉还没有找到更合适的方式。刷题似乎也是一种，但是总感觉还差了些什么。也许是实战？

目前振哥的题目都还是偏简单的，也许可以尝试附加一道进阶题目？或者周末可以来一个专题？

另外，我觉得题目来至哪里无所谓的，就是leetcode的原题目就挺好，振哥提供的是一个学习小组。

收起

| 查看作业题目 &gt;

...



LFeng

3小时前

...

day21 #总结# ☕

现在每天都尽量留一段时间给星球，除非加班太晚

养成习惯，每天进步一点，期待365天在星球刷题中都不忘初衷，坚持下去！

总结



查看详情 &gt;



zing

7小时前



...

感觉21天的收获很大，感谢振哥，我觉得重要的一点区分python与C的数组  
(list)，就是python切分的灵活

| 查看作业题目 >



查看详情 >

...



□

8小时前

#打卡# 时间很快，都已经21天了，觉得最大的收获就是可以跟着，一起讨论，自己的不是最好的，集思广益，觉得这样刷题，很丰富自己！

| 查看作业题目 >

打卡



查看详情 >

...



Bruce

8小时前



有的算法题看起来很简单，但实现起来往往回遇到各种bug。现在做了几天题目，感觉比以前处理各种bug问题更得心应手。

| 查看作业题目 >



查看详情 >



Leven

9小时前

作业

...

Day 21

时间过得真快，一下子在星球就已经二十多天了，过去的这些日子过得都很充实，每天都有在进步，也很感谢球主@zhenguo 创造了这么个环境让大家一起努力成长，希望各位都能坚持下去，互相学习，一起进步。Giao！

| 查看作业题目 >



Okra

9小时前

作业

...

day 21

一下子已经过去了21天，虽然说大多数题目的基本写法都能非常快速的构思出，但看了其他星友的写法每次都在迫使自己写出更高效的代码，非常有启发与挑战性。

| 查看作业题目 >



查看详情 > e1



风岚风清玉琢玉

11小时前

作业

...

最大的帮助就是让我打开了Leetcode。以前一直说要刷Leetcode，也放在收藏夹最显眼的位置上，但事实上就是一直说一直没做。

万事开头难，一旦开头就犹如孙子兵法兵势篇所说——“如转圆石于千仞之山者，势也”。

我相信一切都会越来越好，星球里的大家，一起加油吧！！！

| 查看作业题目 >



查看详情 >



corch

12 小时前

## Day21

看着每天星球里许多球友积极打卡，自己也是动力十足，即使刚开始落后了几天，也抽空都补回来了。

之前有过比较零散的刷题，这次准备配合星球打卡，系统性学习一遍数据结构和算法。

| 查看作业题目 >



查看详情 >

还有更多星友的21天打卡总结，在此不一一列举。

总之，看到大家有收获，所以与大家一起坚持下去，大的指导方向不变。按照有些星友的反馈，会增加进阶题目，同时周末会增加算法学习经验分享等。

## Day22 递归相反顺序打印字符串

前面的归并排序，用到递归。

递归是计算机科学中的一个重要概念。它是计算机算法的基础。接下来几天，我们详细的探讨递归的原理，如何更高效的使用递归解决实际问题。

今天先从一个基础题目体会递归的原理：使用递归以相反的顺序打印字符串。

```
def reverse_print(s):
    #
    #补充完整
    #
```

# 递归方法一

```
def reverse_print(s):
    if len(s) <= 1:
        return s
    return reverse_print(s[1:]) + s[0]
```

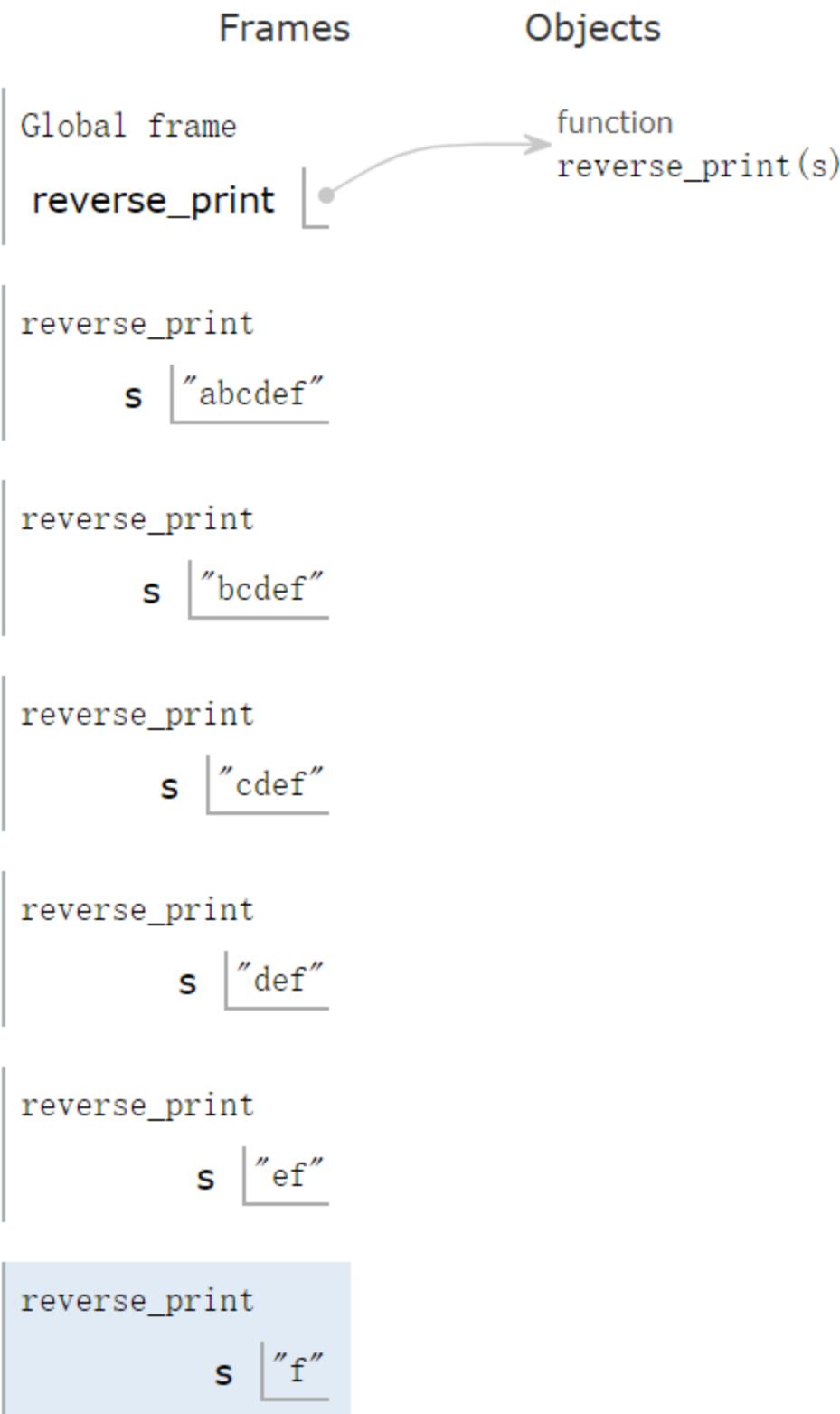
使用递归，往往代码会很简洁，但若不熟悉递归，理解起来就会相对困难。

下面借助示意图解释以上递归过程：

假如这样调用 `reverse_print` :

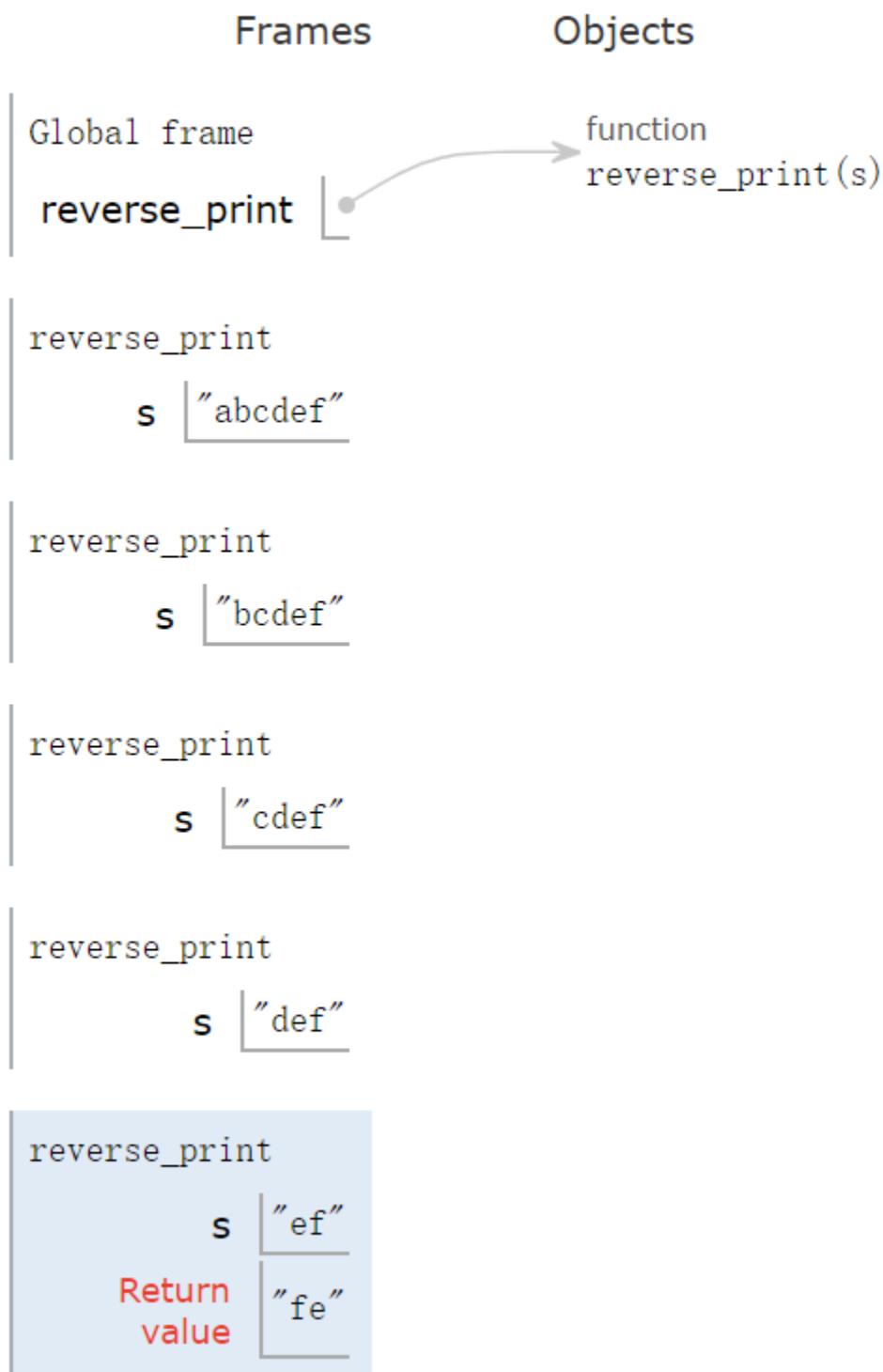
```
reverse_print('abcdef')
```

那么递归过程时，函数 `reverse_print` 会不断入栈，示意图如下：

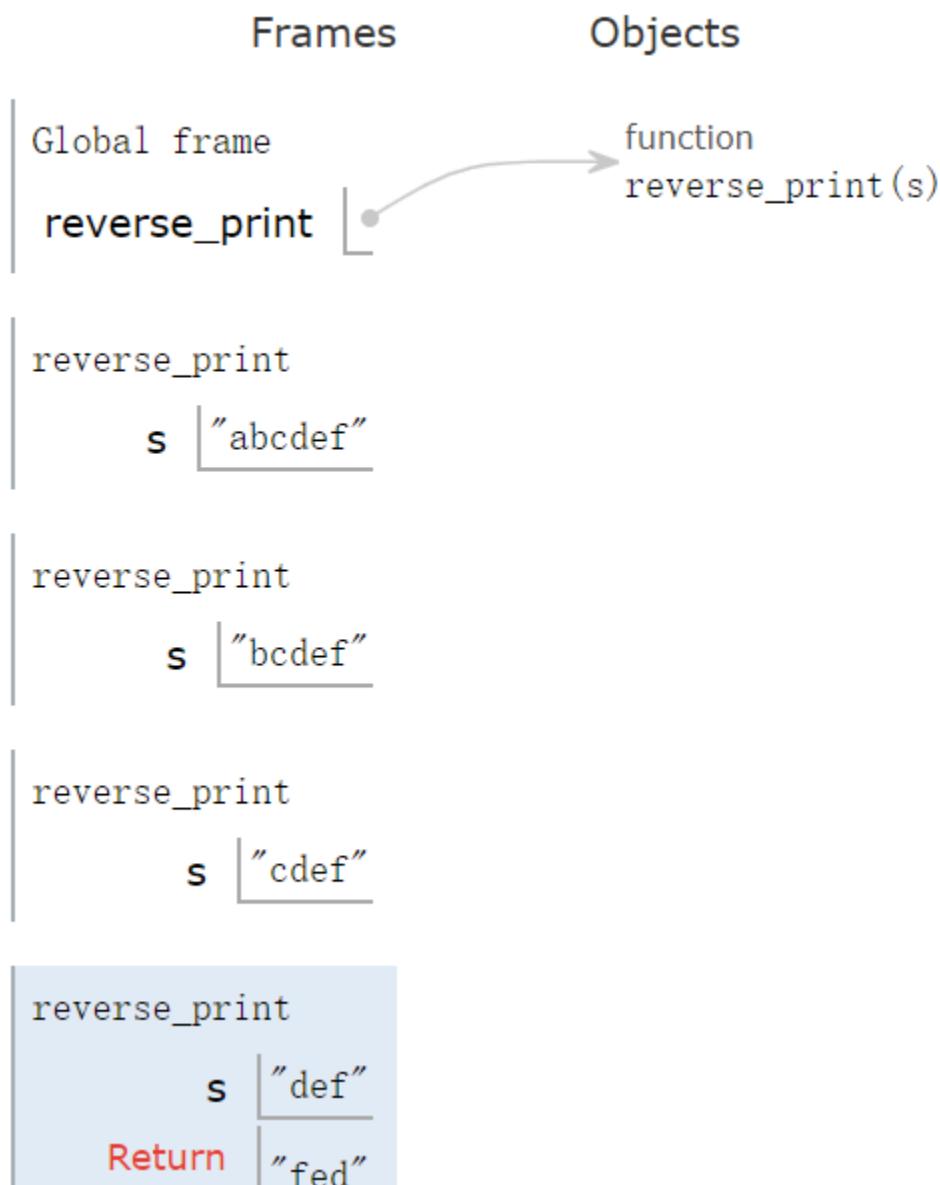


此时栈顶为入参 `f` 的函数，位于示意图的最底部。

因为它满足了递归返回条件 `len(s) <= 1`，所以栈顶函数首先出栈，并返回值 `f`，下一个即将出栈的为入参 `ef` 的函数，其返回值为 `fe`，如下所示：



依次出栈：



## Frames

## Objects

Global frame

reverse\_print

function  
reverse\_print(s)

reverse\_print

s "abcdef"

reverse\_print

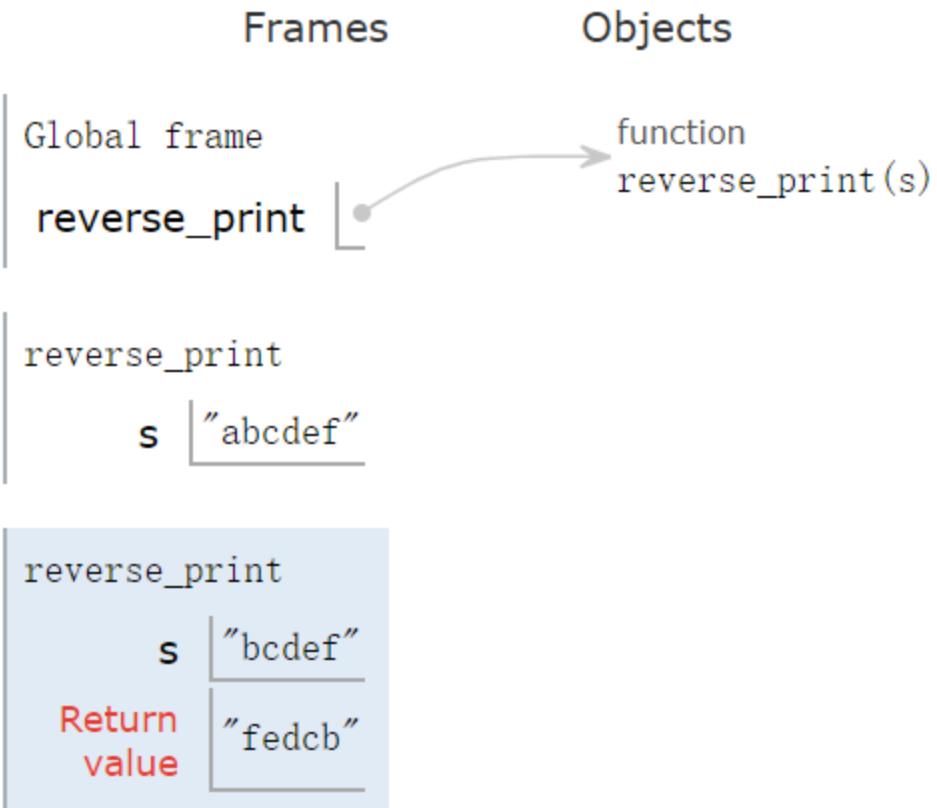
s "bcdef"

reverse\_print

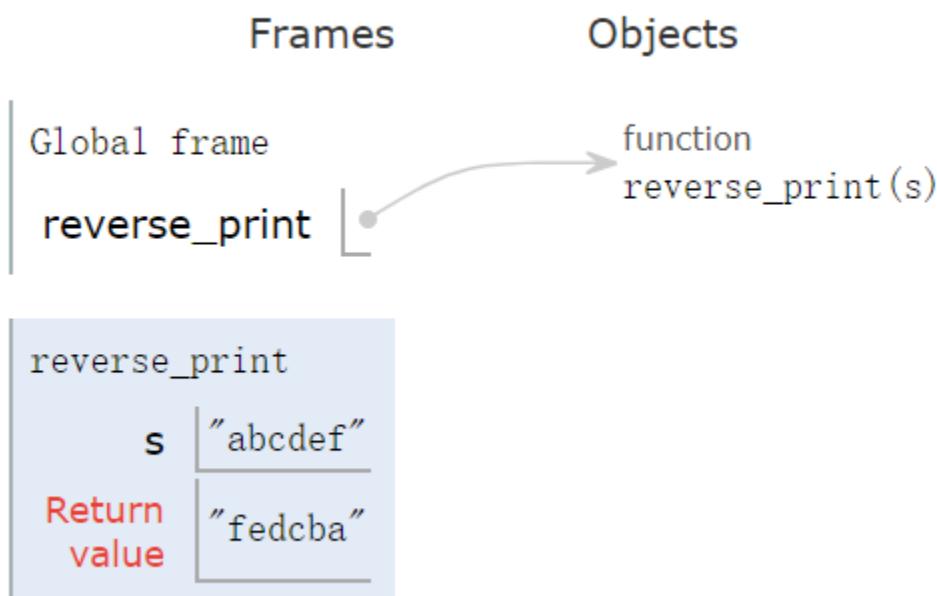
s "cdef"

Return  
value

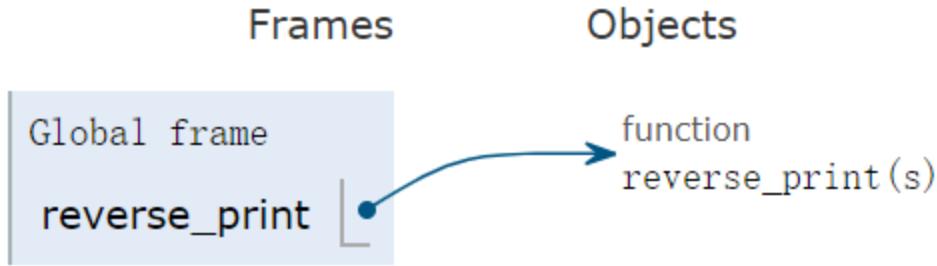
"fedc"



最后一个留在栈的 `reverse_print`，即将返回我们想要的结果：



它也出栈后，我们变得到结果 `fedcba`



以上就是使用递归反转打印字符的方法。

其他使用递归反转字符串的方法，大家多看看其他星友的解法即可。

## Day23 递归两两交换链表节点

给定链表，交换每两个相邻节点并返回其头节点。

例如，对于列表  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ，我们应当返回新列表  $2 \rightarrow 1 \rightarrow 4 \rightarrow 3$  的头节点。

请补充下面函数：

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

class Solution:
    def swapPairs(self, head: ListNode):
        pass # 请补充
```

## 分析过程

给定链表，交换每两个相邻节点并返回其头节点。

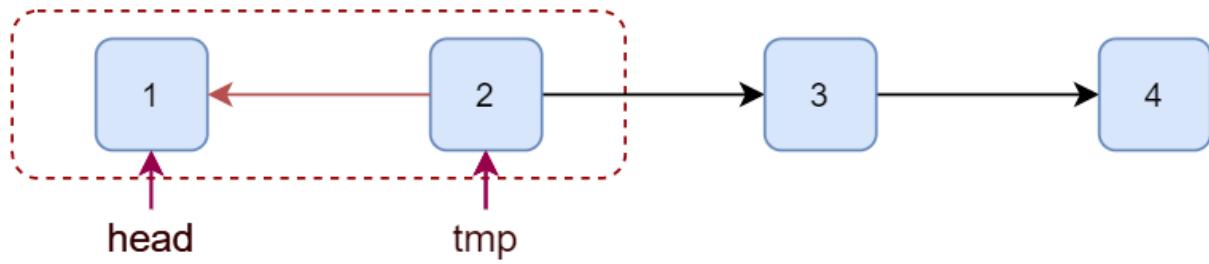
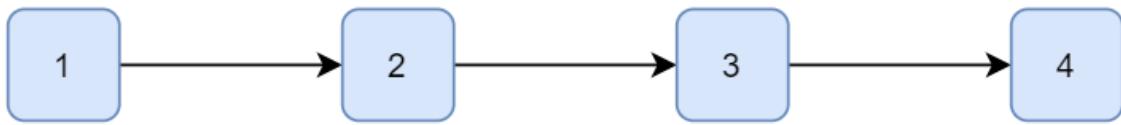
例如，对于列表  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ，我们应当返回新列表  $2 \rightarrow 1 \rightarrow 4 \rightarrow 3$  的头节点。

请补充下面函数：

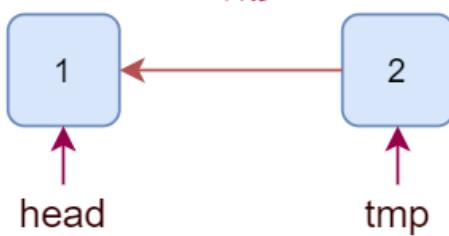
```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def swapPairs(self, head: ListNode):
        pass # 请补充
```

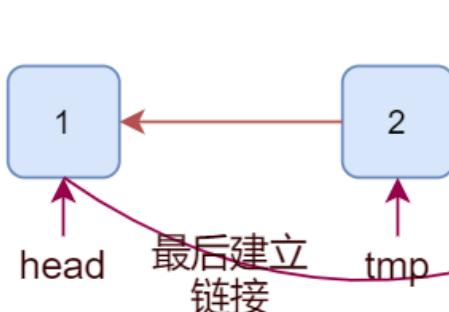
使用递归的解题思路，见下面示意图：



问题规模  
减少



剩余链  
swapPairs



表头 r

剩余链  
swapPairs

兑现为代码如下：

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
            return head
        tmp = head.next
        r = self.swapPairs(tmp.next)
        tmp.next = head
        head.next = r
        return tmp
```

连上完整的验证代码：

```

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

class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
            return head
        tmp = head.next
        r = self.swapPairs(tmp.next)
        tmp.next = head
        head.next = r
        return tmp

if __name__ == "__main__":
    # create ListNode from list a
    a = [1, 2, 3, 4, 5]
    head = ListNode(a[0])
    tmp = head
    for i in range(1, len(a)):
        node = ListNode(a[i])
        tmp.next = node
        tmp = tmp.next
    # swap pairs
    snode = Solution().swapPairs(head)
    # check result
    tmp = snode
    while tmp:
        print(tmp.val)
        tmp = tmp.next

```

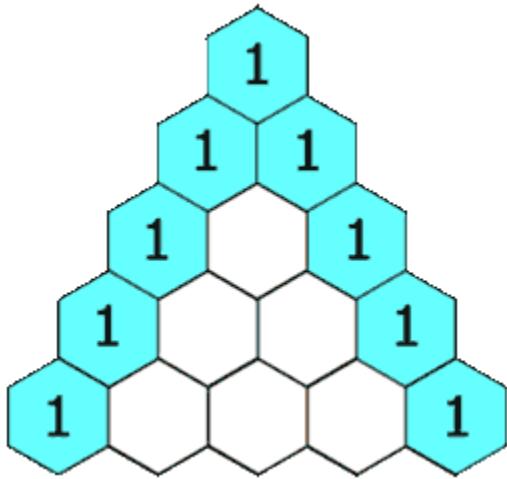
递归使用的诀窍：

每当递归函数调用自身时，它都会将给定的问题拆解为子问题。递归调用继续进行，直到到子问题无需进一步递归就可以解决的地步。

刚刚接触递归，使用起来会比较别扭，大家不妨结合昨天的递归调用栈，再多练习几道递归题目，相信会越来越熟练。

# Day24 递归生成杨辉三角

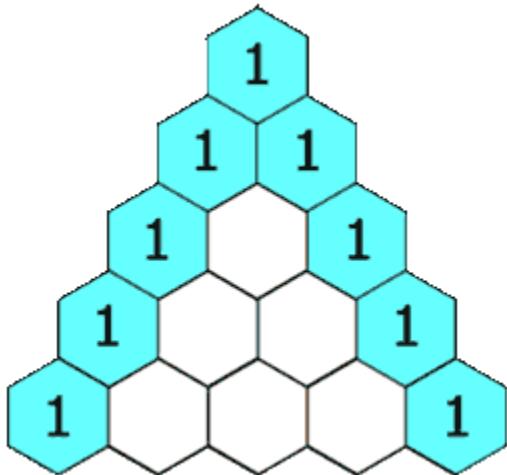
给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。



请补全下面代码：

```
class Solution:  
    def generate(self, numRows: int) -> List[List[int]]:  
        pass
```

给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。



## 分析过程

已知杨辉三角第  $i-1$  行，生成第  $i$  行为：

```
[1] +
[yanghui[-1][i-1] + yanghui[-1][i] for i in range(1, numRows-1)] +
[1]
```

完整代码：

```
class Solution():
    def generate(self, numRows):
        if numRows == 0:
            return []
        elif numRows == 1:
            return [[1]]
        else: # 调用自身生成前 numRows - 1 行的杨辉三角
            yanghui = self.generate(numRows - 1)
            # 根据倒数第二行再生成最后一行:
            last_row = [1] + [yanghui[-1][i-1] + yanghui[-1][i] for i in range(1, numRows-1)] +
            yanghui.append(last_row)
        return yanghui
```

递归总结

在实现递归函数之前，有件重要的事情需要解决：找出递推关系。

## Day25 递归求斐波那契数列前 N 项

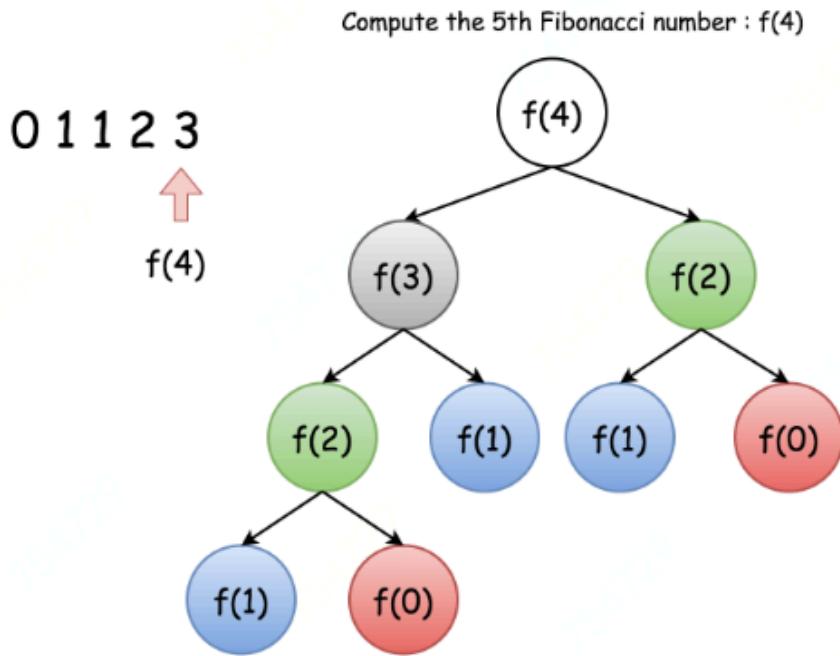
下面再进一步，学习递归的其他知识。

通常情况下，递归是一种直观而有效的实现算法的方法。但是，如果使用不合理，会造成大量的重复计算。

例如求斐波那契数问题时，参考星友 infrared62 的解释：Fibonacci Number，使用递归来解，首先如果直接递归会用很多重复子问题计算，画完二叉树会发现是指数级的时间复杂度，每一层的计算比上一层多2倍。

下面的树显示了在计算  $F(4)$  时发生的所有重复计算（按颜色分组）。

下面的树显示了在计算  $F(4)$  时发生的所有重复计算（按颜色分组）。



那么，你有什么办法能消除某些重复计算呢？很自然的一个想法，将中间结果存储在缓存中，以便以后可以重用它们，而不需要重新计算。

这是一种经常与递归一起使用的技术。

## 分析过程

通过求斐波那契数问题，体会如何消除递归计算中的重复计算问题。

具体代码实现：

```

def fib(self, N):
    history = []
    def recur(N):
        if N in history:
            return history[N]
        if N < 2:
            result = N
        else:
            result = recur(N-1) + recur(N-2)
        history[N] = result
        return result

    return recur_fib(N)

```

补全以上代码，返回斐波那契数列前 N 项。

星友 infrared62 还提出一个缓存的方法：在Python中也可以用语言特性 `@lru_cache` 来自动缓存。

代码参考下面：

Fibonacci数的递归解法，使用记忆化搜索

```

class Solution:
    memo = {}
    def fib(self, N: int) -> int:
        if N < 2:
            return N
        if N - 1 not in self.memo:
            self.memo[N - 1] = self.fib(N - 1)
        if N - 2 not in self.memo:
            self.memo[N - 2] = self.fib(N - 2)

        return self.memo[N - 1] + self.memo[N - 2]

```

Python语言特性可以使用注解LRU cache来缓存重复子问题的结果

```

from functools import lru_cache
class Solution:
    @lru_cache
    def fib(self, N: int) -> int:
        if N < 2:
            return N

        return self.fib(N - 1) + self.fib(N - 2)

```

Day26 现实中一名算法工程师日常

如果你是学生，还未毕业工作，你可能会畅想着将来成为一名算法工程师，看起来高大上，待遇各方面都不错。那么现实中，一名算法工程师的日常又是什么呢？你可以想象一下然后打卡。如果你是算法工程师，那么平时大部分时间在做什么，也欢迎打卡留言。

## 精选回答

首先，大概看下算法工程师日常做什么：

算法工程师的日常 星友 LFeng 回答

业务场景分析 需求分析 数据处理 建模 调试 应用 反馈调整 总结报告等

这个总结很精炼。

参考星友 箱子 回答

我感觉算法工程师有相当一部分时间是在处理数据。算法工程师也是为项目服务，项目也是为最后的解决方案服务，解决方案也是为了解决现实生活中的实际问题。为了让现实生活中数据呈现出背后隐藏的信息或者趋势，才需要一个算法来支撑，但是我们能得到的却是杂乱无章的东西。整理数据，优化数据结构，我感觉也需要很多的工作量。

具体来说，参考星友 北方 回答

借鉴于知乎 1. 对接业务方，第一个步骤是去和业务方进行对接，对接的过程中要去了解业务方的需求，业务的真实痛点，很多时候对接的业务方并不了解算法能做哪些事情，因此，在对接的过程中业务方会提自己想要的目标是什么，而此时算法工程师需要凭借经验去评估这个方向的业务价值，评估这个方向是否适合算法来做，现在是否是切入的好时机。

2. 数据的盘点，对于算法来说，数据的重要性不言而喻，很多时候调半个月的参数，模型的效果都不如引入一份高质量数据来得效果好。阿里算是数据的基础设施做的很好的一家公司，数据在不同平台的流转工作相对方便，另外阿里也有很多基础数据，然而，实际在工作的过程中还是会碰到没有数据，或者数据质量不佳的情况，在这种情况下一方面需要对数据做出大量预处理的工作，另一方面，需要去思考如何依赖现有数据对模型进行评估，进而去评估业务效果。

3. 建模，包括以什么样的思路去解决这个问题，预测模型效果增益，特征抽取，特征处理，选用何种模型，效果评估，模型迭代，这部分可能是在我的人之中算法工程师的工作，而

在实际工作中，这部分工作如果能占用30%的精力，已经是很高的比例了。

4.模型上线，取决于依赖何种上线方式，有的时候很简单，可能产出的结果只是一张结果表，业务方下游直接引用即可，有的时候涉及到大量的工程工作，需要考虑模型的性能，复杂度。延时，可解释性什么的也会需要去进行考虑。

这四个过程中就是算法的日常工作，精力分配大概是30%，10%，30%，30%，这只是一个毛估估的精力分配比例，有可能一两周都是处于和业务方开会过程中，也有可能某一个小项目从头到尾都比较确认，直接当天就能从第一步走到第四步。

这个回答，确实很中肯。

算法工程师需要具备的能力，可以大概参考星友：孙颖颖颖颖颖的回答：

举我的例子 一名合格的图像处理算法工程师 不但要有扎实的计算机基础 什么c++和python 是必须要熟练使用的 还有一些深度学习的框架如tf torch 还有一些图像处理的算法 opencv 等等 然后就是细分方向了 人脸识别 目标检测等等 在学校还是以学术复现顶会论文为主 出学校就是根据自己的方向从事相关的工作吧 解决问题 根据项目要求设计模型 优化模型 在项目上有想法就发发专利和论文 感觉算法工程师重在idea吧 挺难的。

大家一步一步来，慢慢掌握

## Day26 递归的时间复杂度分析

大家参考下面网址：

<https://leetcode-cn.com/explore/original/card/recursion-i/259/complexity-analysis/1222/>

回答一个问题：如何分析递归的时间复杂度？

递归确实能让代码变得更加简洁，让代码看起来更加优美，但是稍不留神，就会写出时间复杂度为指数级的代码。所以使用递归，必须要留意时间复杂度分析。

## 分析过程

里面主要讲到两类时间复杂度分析，一种以相反顺序打印字符串的递归，这种时间复杂度的求解较为直观，因为每次问题规模  $n$  都会缩减 1，并且每次只打印 1 个字符，故时间复杂度为：

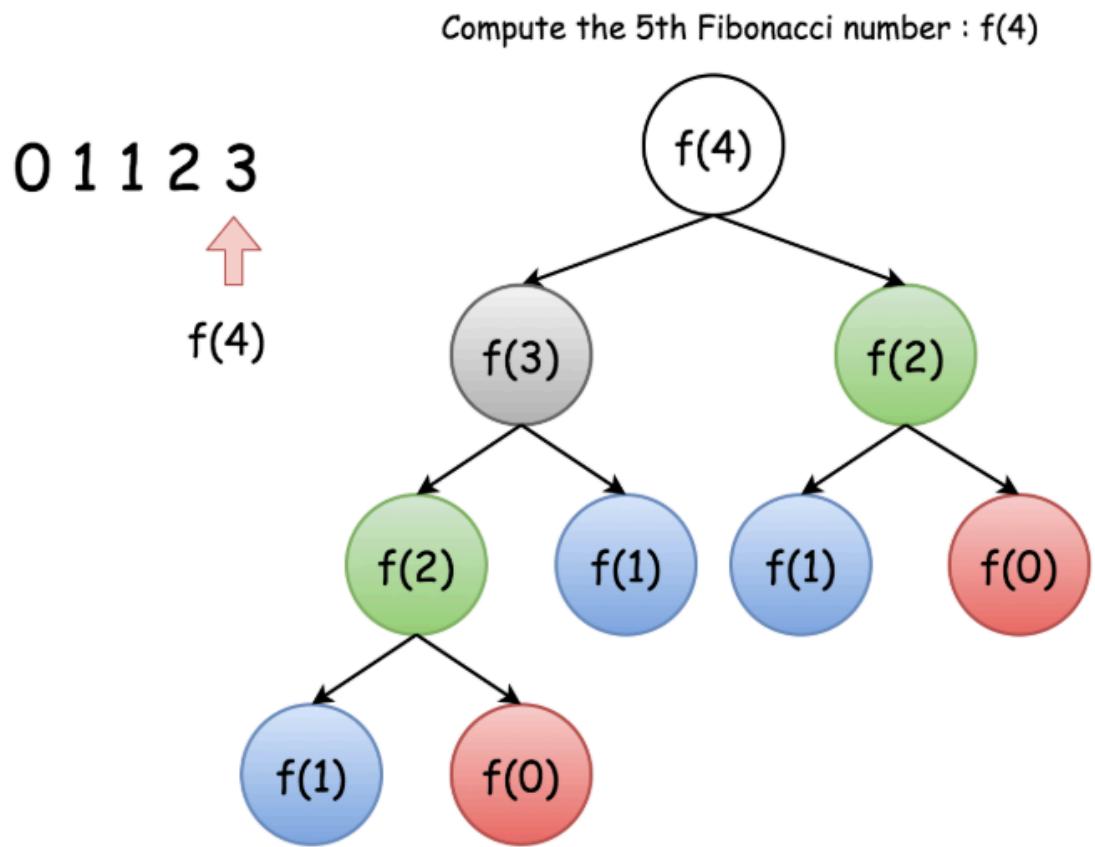
$$O(n) = O(n) \times 1 = O(n)$$

但是，以求解裴波那契数列为为代表的递归，时间复杂度的求解就不那么直观了，文中给出一个很好的求解示意图：

$$\text{因为递归的方程 : } f(n) = f(n - 1) + f(n - 2)$$

以求解数列前 4 项为例，在求解  $f(4)$  是需要求解出  $f(3)$  和  $f(2)$ ，求解  $f(3)$  时又需要求解  $f(2)$  和  $f(1)$ ，以此类推。

整个过程可以绘制为下图的二叉树：



我们知道二叉树模型的节点个数与层数的关系为指数次幂，所以递归的时间复杂度为： $O(2_n)$

如果不做优化，时间复杂度为指数级的算法基本是难解，或者不是一个真正意义上的可行解， $2_{50}$  就已经是一个很恐怖的数字：

$$1.1258999068426e_{15}$$

所以使用递归再次告诉我们：记忆化技术或称为缓存技术的重要性。

以上就是两类典型的递归时间复杂度。

## Day28 0-1 背包问题

0-1 背包是一个经典的组合优化问题，其中的思想非常重要。今天我们以一个简单的例子，先来体会 0-1 背包问题。

有一个最大承重量为  $w$  的背包，第  $i$  件物品的价值为  $a1[i]$ ，第  $i$  件物品的重量为  $a2[i]$ ，将物品装入背包，求解背包内最大的价值总和可以为多少？

例子：

$a1 = [100, 70, 50, 10]$ ,  $a2 = [10, 4, 6, 12]$ ,  $w = 12$ , 背包内的最大价值总和为 120，分别装入重量为 4 和 6 的物品，能获得最大价值为 120

补全下面代码，返回求解的最大价值：

```
def f(a1, a2, w):  
    pass
```

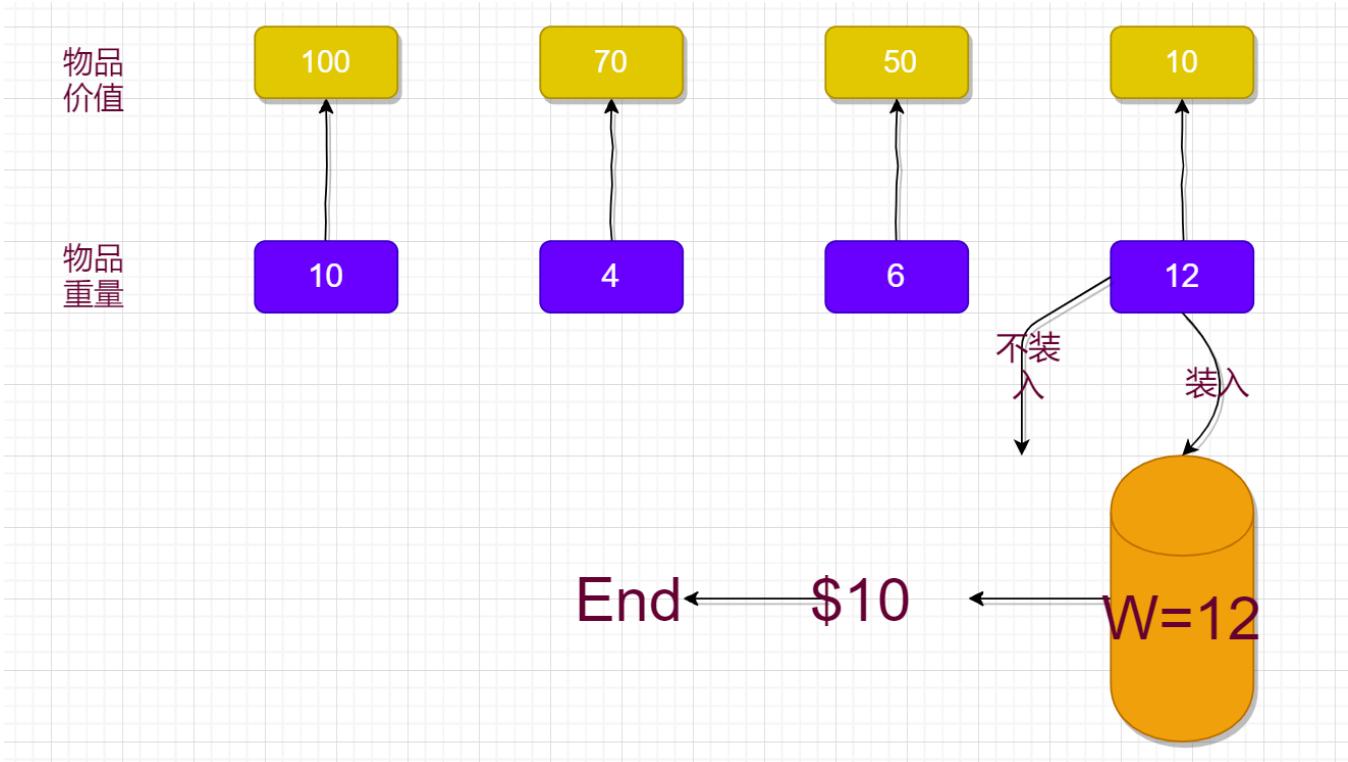
## 分析过程

如下图所示：

第一行物品价值

第二行物品重量

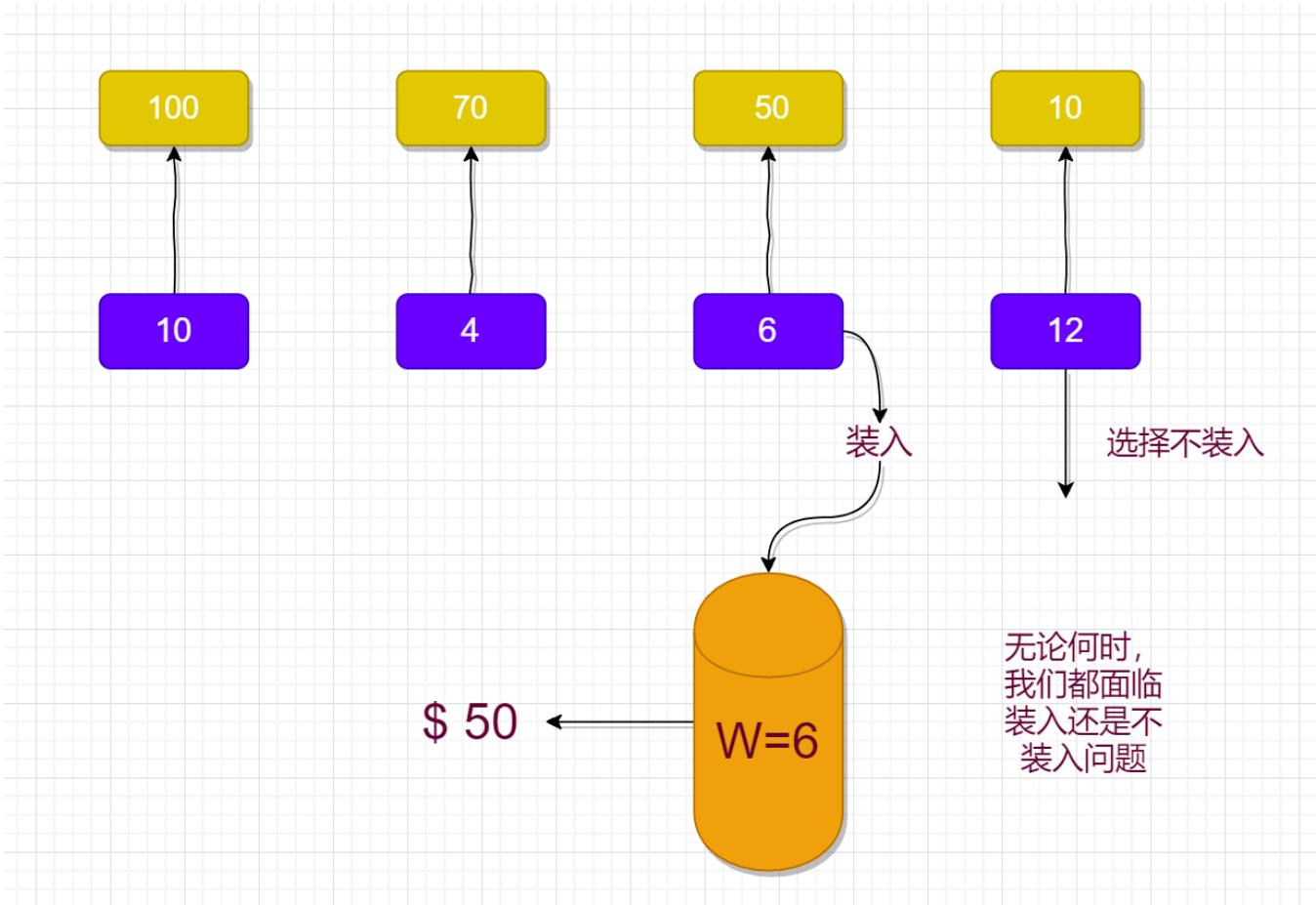
我们从最右侧开始决策是否装入重量为 12 的物品：



背包可装最大重量恰好为 12

- 0. 如果选择装入此物品，背包内物品价值为 10，并且已经不能再装入，因此得到一种可行解：价值为 10
  1. 如果选择不装入，我们的视线移动到下一个物品决策上，同样地我们会面临装入还是不装入的两个可选择项：

如果选择装入，创造 50 价值，并且还能最多装入重量为 6 的物品：



以此类推。

你看，无论何时，当视线移动到下一个物品时，我们都会面临装还是不装的问题，称此类问题为：

## 求解

以上过程转化为如下求解代码：

```

a1 = [100, 70, 50, 10]
a2 = [10, 4, 6, 12]
w = 12

def f(i, w):
    # 基本情况:
    if w == 0 or i < 0:
        return 0
    elif a2[i] > w:
        return f(i-1, w)
    # 递归:
    return max(a1[i] + f(i-1, w-a2[i]),
               f(i-1, w))

r = f(3, w)
print(r) # 120

```

基本情况包括：

1. 背包可装载重为 0 时，表明它不能再装物品了，自然价值为 0
2.  $i < 0$  表明没有物品可装入了，自然价值也为 0
3. 当  $a2[i] > w$  时，表明此物品太重，背包剩余空间装载不下，表明此物品不能装入背包，我们的视线自动移动到下一个物品，即返回： $f(i-1, w)$

递归情况：

到这里表明， $a2[i]$  能装入背包，就看我们是否选择它：

0-1 选择问题：

装： $a1[i] + f(i-1, w-a2[i])$ ，产生  $a1[i]$  大小的价值，代价背包剩余空间变小，还能装载  $w-a2[i]$

不装： $f(i-1, w)$

决策：选择较大的

以上问题递归树的完整示意图如下：

# Day29 复习 0-1 背包问题

如果觉得昨天作业不好理解，或者之前拉下作业的星友，借助Day29的机会弥补一下。已经全都搞定的星友，Day29放松一下吧。

# Day30 理解递归的特例：尾递归

0-1 背包问题的递归解法有些星友理解起来有些困难，所以Day29我们放慢脚步，单独留出这一天好好理解背包问题。

算法的魅力：它会让你念念不忘，几天后必有回响，然后编程水平就会提升一个小小的 level

今天我去油管上翻阅一下0-1背包讲解视频，觉得下面这个讲解较为形象，特意再在这里帖一下：

图1(左上角)：背包问题示意图

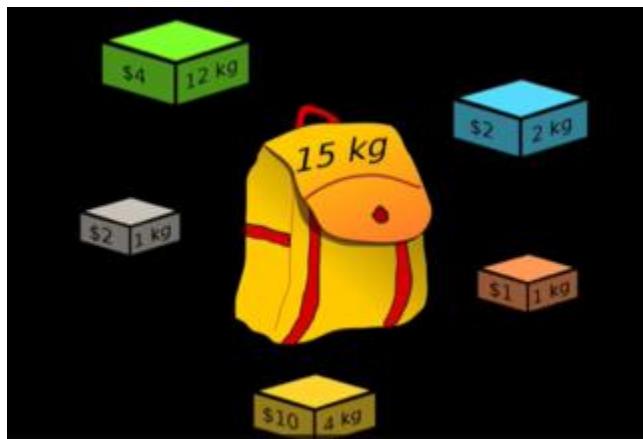


图2：待求解问题，例子

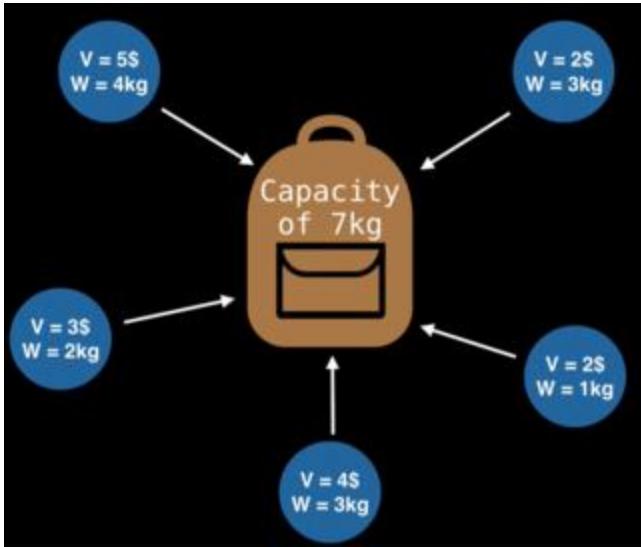


图3：求解表格，这是动态规划的求解，不是我们这周训练的递归的求解方法。现在这里提一下动态规划，后面会重点讲到。

		Capacity							
		0	1	2	3	4	5	6	7
Empty		0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	2	4	4	4	4
$v_3=4, w_3=3$	3	0	2	2	4	6			
$v_4=5, w_4=4$	4								
$v_5=3, w_5=2$	5								

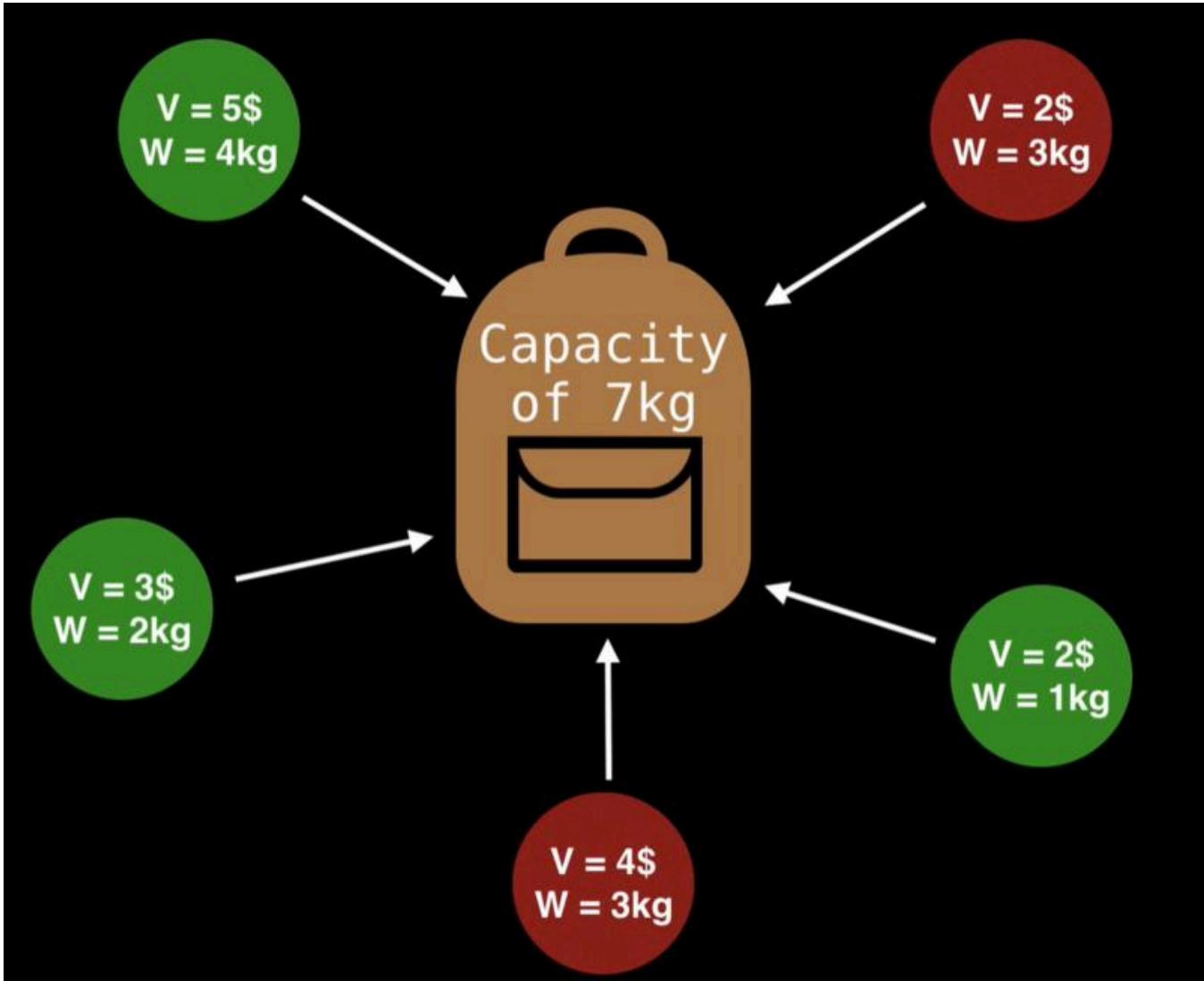
图4：全部求解完成

		Capacity							
		0	1	2	3	4	5	6	7
Empty		0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	2	4	4	4	4
$v_3=4, w_3=3$	3	0	2	2	4	6	6	6	8
$v_4=5, w_4=4$	4	0	2	2	4	6	7	7	9
$v_5=3, w_5=2$	5	0	2	3	5	6	7	9	10

图5：检验价值7如何得来的

	0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	4	4	4	4
$v_3=4, w_3=3$	3	0	2	4	6	6	6	8
$v_4=5, w_4=4$	4	0	2	2	4	6	7	9
$v_5=3, w_5=2$	5	0	2	3	5	6	7	9

图6：最后选择放置到背包里的三个物品，绿颜色表示：



## 分析过程

使用递归会使代码变得非常简洁，但是递归利用不慎，很容易就会出现：stack overflow 栈溢出的问题，这是因为通常的递归也需要消耗在系统调用栈上产生的隐式额外空间，这算是我们使用递归所付出的代价。

但是有一类递归非常特殊，它不受此空间开销的影响。它就是一种特殊的递归情况：尾递归。

那么，满足哪些条件才算是尾递归呢？下面的两种代码示例，哪个是尾递归，哪个是一般的递归情况呢？

写法1：

```
def sum1(ls):
    if len(ls) == 0:
        return 0
    return ls[0] + sum1(ls[1:])
```

写法2：

```
def sum2(ls):
    def helper(ls, acc):
        if len(ls) == 0:
            return acc
        return helper(ls[1:], ls[0] + acc)
    return helper(ls, 0)
```

递归调用是递归函数中的最后一条指令。并且在函数中应该只有一次递归调用。

大家注意：最后一行语句和最后一条指令的区别：

下面代码中 `sum1` 函数最后一条语句也是 `sum1` 函数，但是最后一条指令显然是加法操作。所以它不是尾递归！

```
def sum1(ls):
    if len(ls) == 0:
        return 0
    return ls[0] + sum1(ls[1:]))
```

下面函数 `sum2` 中的子函数 `helper` 的最后一条指令也是 `helper`，所以它是尾递归：

```
def sum2(ls):
    def helper(ls, acc):
        if len(ls) == 0:
            return acc
        return helper(ls[1:], ls[0] + acc)
    return helper(ls, 0)
```

总结：非尾递归中，在最后一次递归调用之后有一个额外的计算。

# Day31 递归快速幂算法：Pow(x,n)

求  $x$  的  $n$  次幂，一般解法时间复杂度为： $O(n)$ ，你能使用递归写出  $O(\log n)$  的解法吗？

补全如下代码，返回  $x$  的  $n$  次幂：

```
def pow(x, n):
    pass
```

## 分析过程

求  $x$  的  $n$  次幂，一般解法时间复杂度为： $O(n)$ ，你能使用递归写出  $O(\log n)$  的解法吗？

补全如下代码，返回  $x$  的  $n$  次幂：

代码主要分三块：

1 递归的基情况，这是必须考虑的

2 处理负次幂

3 递归一次，次幂数减半一次。原理： $x_{2n} = (x_n)^2$ ，所以计算  $x_{2n}$  只需求出  $x_n$

故代码如下：

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        # 基情况
        if n == 0:
            return 1.
        # 处理负次幂情况
        if n < 0:
            x, n = 1/x, -n
        # 递归减半
        t = n//2
        p = self.myPow(x, t)
        return p**2*x if n&1 else p**2
```

# Day32 递推专题总结

递归的求解步：

1 定义递归函数；

2 写下递归关系和基本情况；

3 使用memoization（记忆化）以消除重复计算问题(如果存在)

4 特殊情况：尽可能地实现 尾递归（tail recursion）函数，以优化空间复杂度。

作业题：使用递归合并两个有序列表

```
输入: 1->2->4, 1->3->4  
输出: 1->1->2->3->4->4
```

补全下面代码：

```
# Definition for singly-linked list.  
# class ListNode:  
#     def __init__(self, val=0, next=None):  
#         self.val = val  
#         self.next = next  
class Solution:  
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:  
        pass
```

# Day33 合并两个有序链表

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        # 递归的基线条件
        if not l1:
            return l2
        if not l2:
            return l1

        if l1.val <= l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
```

注意前提：2个链表都是有序的

方法：使用递归

递归使用：要找base case,

问题规模单调递减：每递归一次，其中一个链表长度减去1，所以时间复杂度为  $O(m+n)$ ， $m,n$  分别表示链表l1,l2的结点个数。

```

class singleList():
    def mergeTwoLists(self, l1, l2):
        # base case
        if not l1:
            return l2
        if not l2:
            return l1

        if l1.val <= l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2) # 递归一次 l1.next执行一次
            return l1

        if l2.val < l1.val:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2

```

要找递归方程： $f(m + n) = f(0).next -> merge(m + n - 1)$ ，转化为代码就是分两种情况：

```

if l1.val <= l2.val:
    l1.next = self.mergeTwoLists(l1.next, l2)
    return l1

if l2.val < l1.val:
    l2.next = self.mergeTwoLists(l1, l2.next)
    return l2

```

这道题虽然看似简单，但是在考场上准确写出来可能并不是那么容易，大家多多思考以上两个if条件。

这段时间我们会集中强化过去学习过的数据结构，主要通过做leetcode题目。

今天作业题：寻找插入位置

```

class Solution(object):
    def searchInsert(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """

```

示例 1:

输入: [1,3,5,6], 5

输出: 2

示例 2:

输入: [1,3,5,6], 2

输出: 1

## Day34 寻找有序数组元素插入位置

已知数组有序，重要前提。

题目给出两个例子：

示例 1:

输入: [1,3,5,6], 5

输出: 2

示例 2:

输入: [1,3,5,6], 2

输出: 1

大家注意体会：第一个例子，等元素值相等时靠前插入。

时间复杂度为  $O(n)$  的解比较容易想出来，直接从头遍历，若待插入元素  $target$  满足：

$$nums[i] < target \leq nums[i + 1]$$

则元素  $target$  的插入位置为  $i + 1$

但是  $O(n)$  时间复杂度不是最优解法。此处使用两个指针 `left` 和 `right`，获得  $\log(n)$  时间复杂度解法。

因为数组是有序的，所以如果  $target$  满足：

$$target < nums[(left + right) // 2]$$

则  $target$  只可能位于区间  $[0, (left + right) // 2]$ ，

这样搜索的区间每遍历一次就减半。同理如果  $target$  满足：

$$target > \text{nums}[(\text{left} + \text{right}) // 2]$$

则  $target$  只可能位于区间  $((\text{left} + \text{right}) // 2, \text{right}]$ ，

注意体会下面两个指针的使用方法，这套思路经常会用到。

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

        return left
```

## Day 35 寻找有序数组元素插入位置，若重复靠后插入

输入: [1,3,5,6], 待插入元素为 5 输出: 3

算法题非常灵活，就像算法本身一样，充满变幻，理解透，方能灵活应用。

对照重复靠前插入的代码：

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

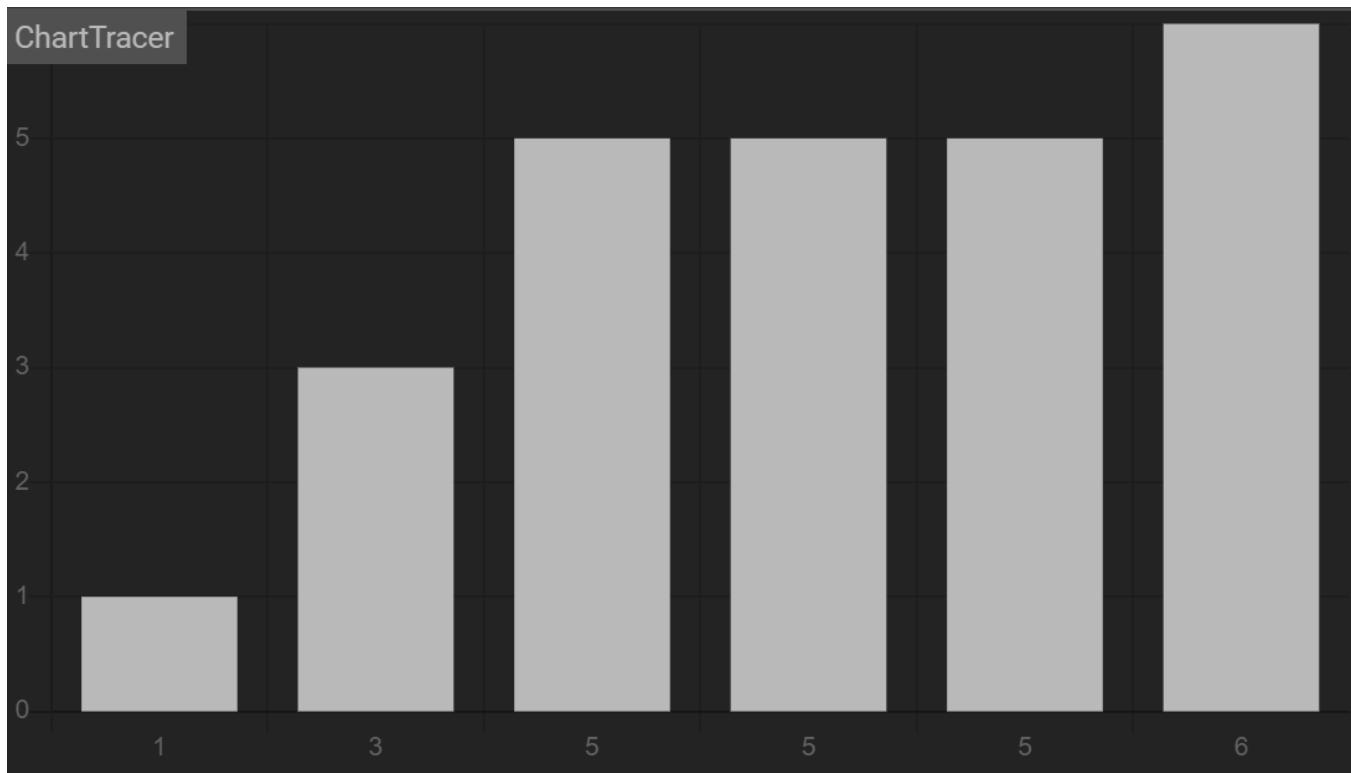
        return left
```

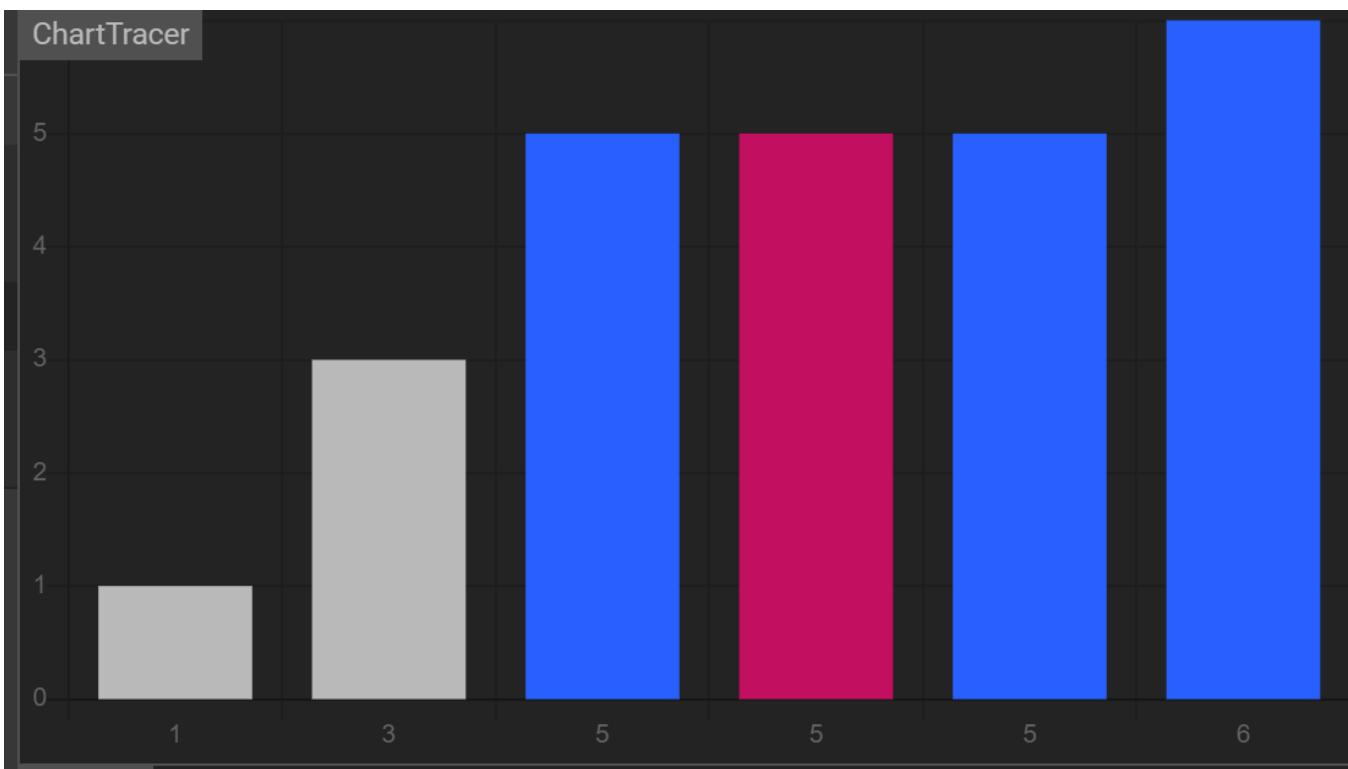
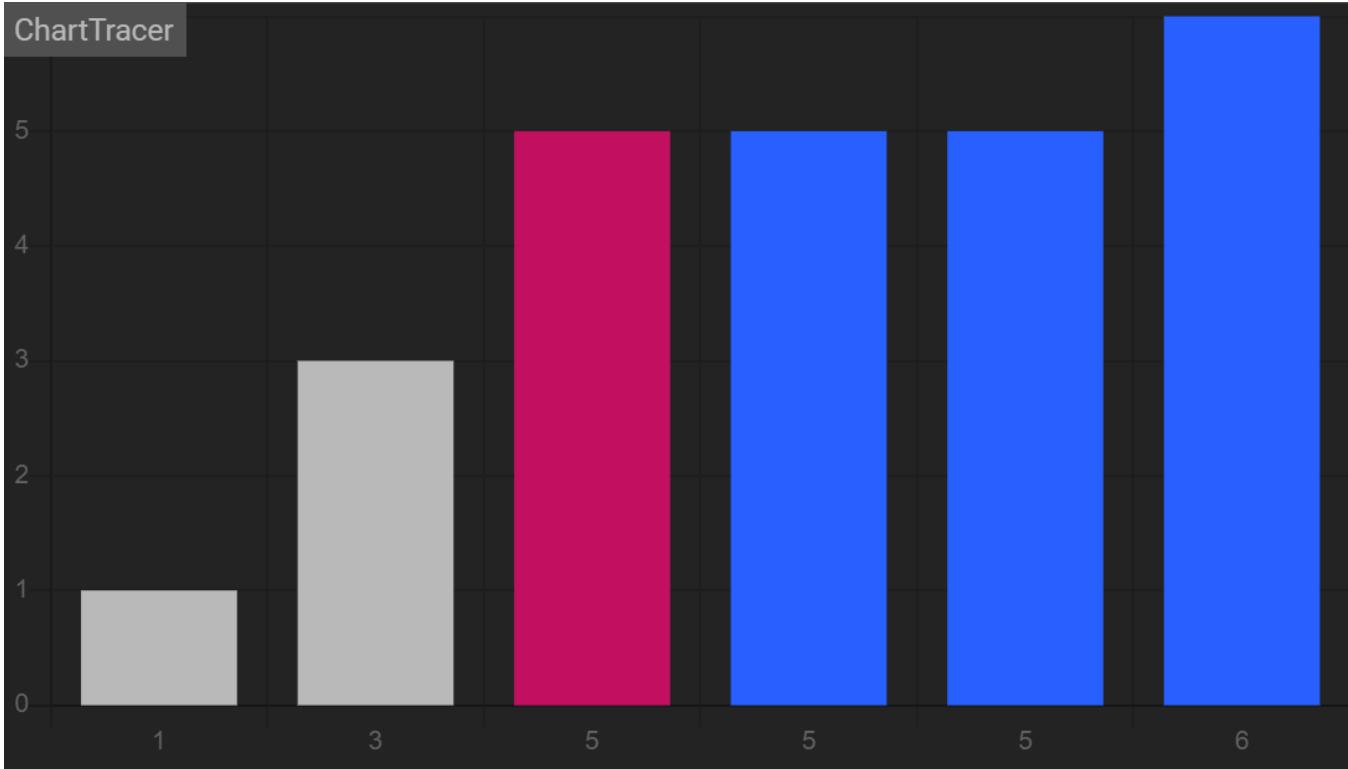
只需修改出现重复元素时的处理部分。

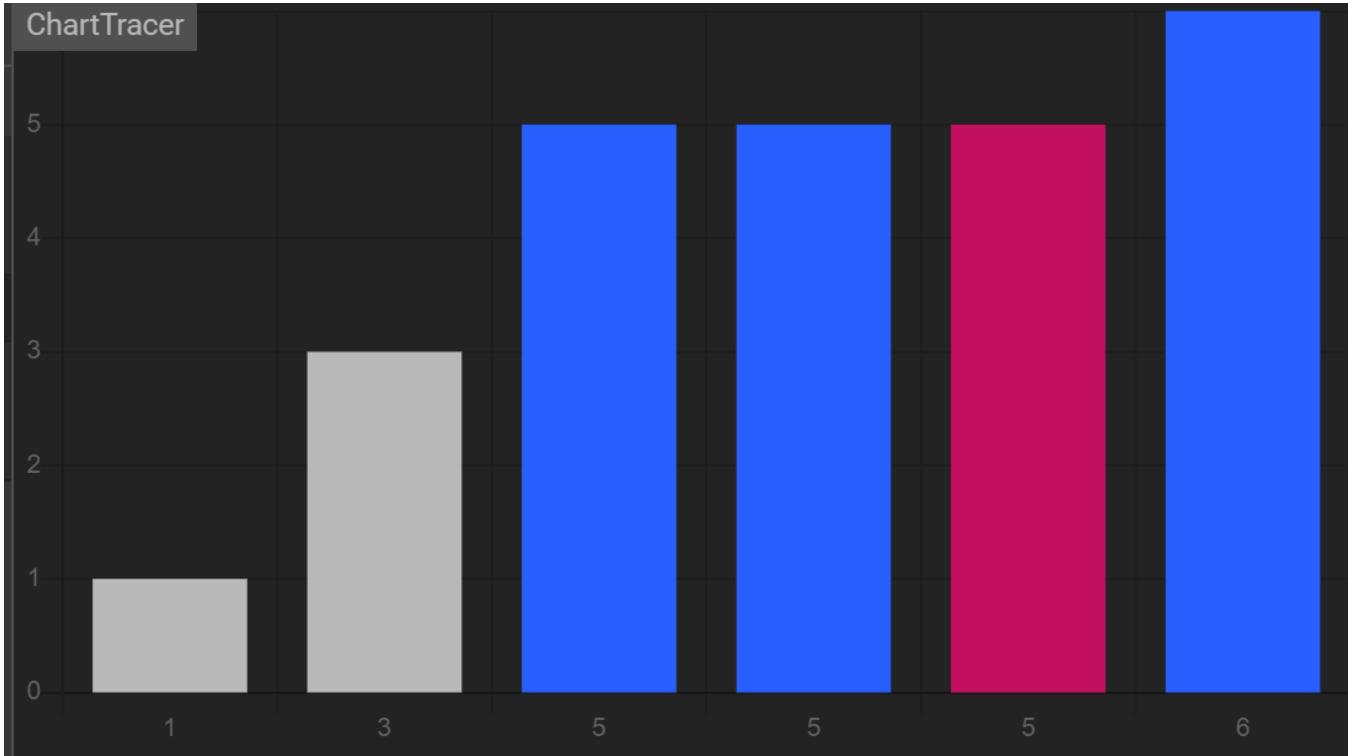
思考过程：

参考这个例子：[1,3,5,5,5,6]，待插入元素 `target` 为 5 时，`mid = 2`, 此时命中待插入元素 5，但是需要比较后面元素是否有 5，直到后一个元素不等于5为止，注意数组访问越界。

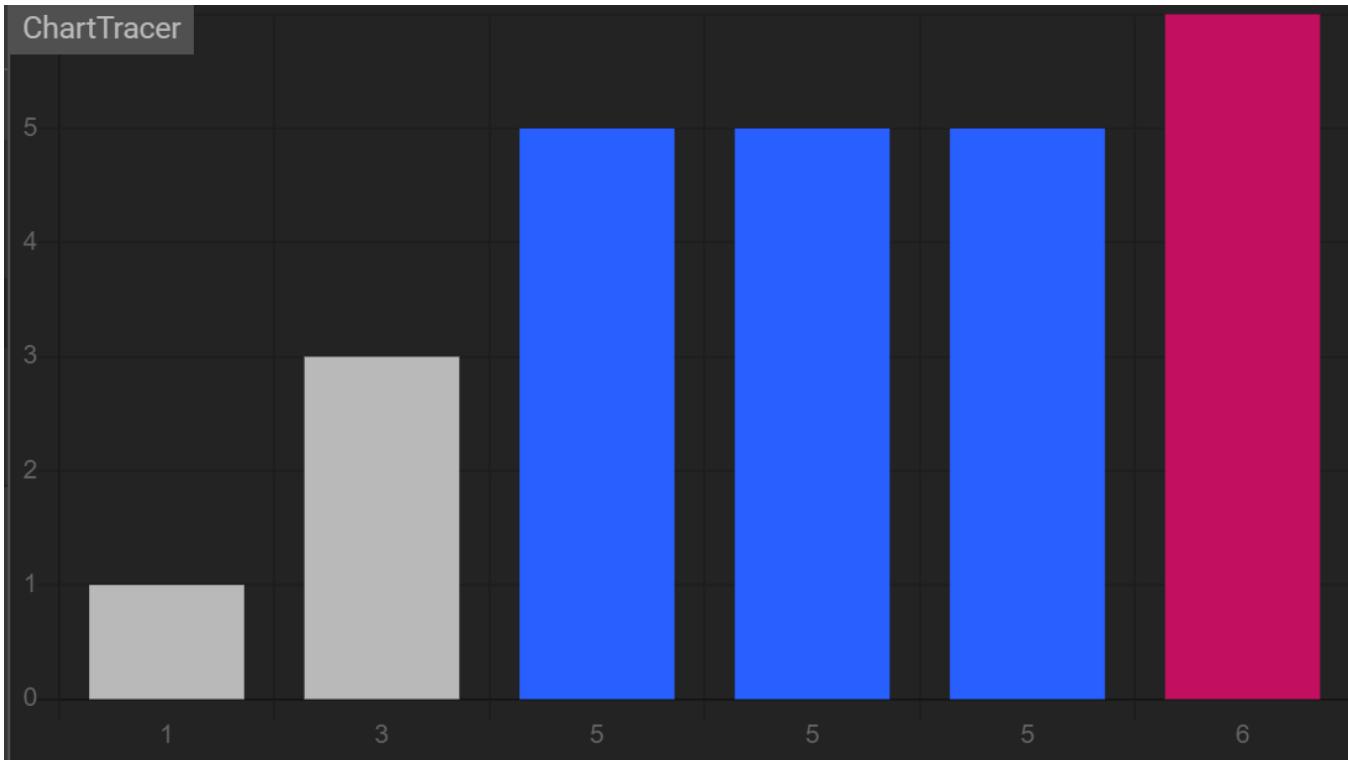
整个搜索过程可分解为如下几步：







找到元素最终插入位置：



代码迭代为：

```

class Solution(object):
    def searchInsert(self, nums, target):
        left,right = 0,len(nums)-1
        while left<=right:
            mid = (left+right)//2
            if nums[mid] == target:
                # 需要比较后面元素是否有`target`，直到后一个元素不等于target为止
                while mid + 1 < len(nums) and nums[mid+1]==target:
                    mid+=1
                # 向后插入
                return mid + 1
            elif target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1

        return left

```

一口气来到 **Day35**，明天休整一天，之前作业没来得及做的可以趁机补充。明天我会汇总过去 35 天的所有作业链接，到时候发给大家，可以回头多多复习。

## Day36 合并两个数组

今天开始再做出一点改变：以后统一星球里发作业，公众号推送答案，这样文章会更加清晰。

考虑到这个题目稍有难度，我们留有2天，2天的作答时间，截止明天晚上前，到时我会参考星友的提交和我的答案，附有较为详细的解题分析，到时推送出来。

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

输入: [1,3],[2,6],[8,10],[15,18]

输出: [1,6],[8,10],[15,18]

解释: 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].

示例 2:

输入: [1,4],[4,5]

输出: 1,5

解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。

补全下面代码：

```
class Solution(object):  
    def merge(self, intervals):
```

说明，这道题看似简单，实际需要考虑周全。比如出现这样的区间：

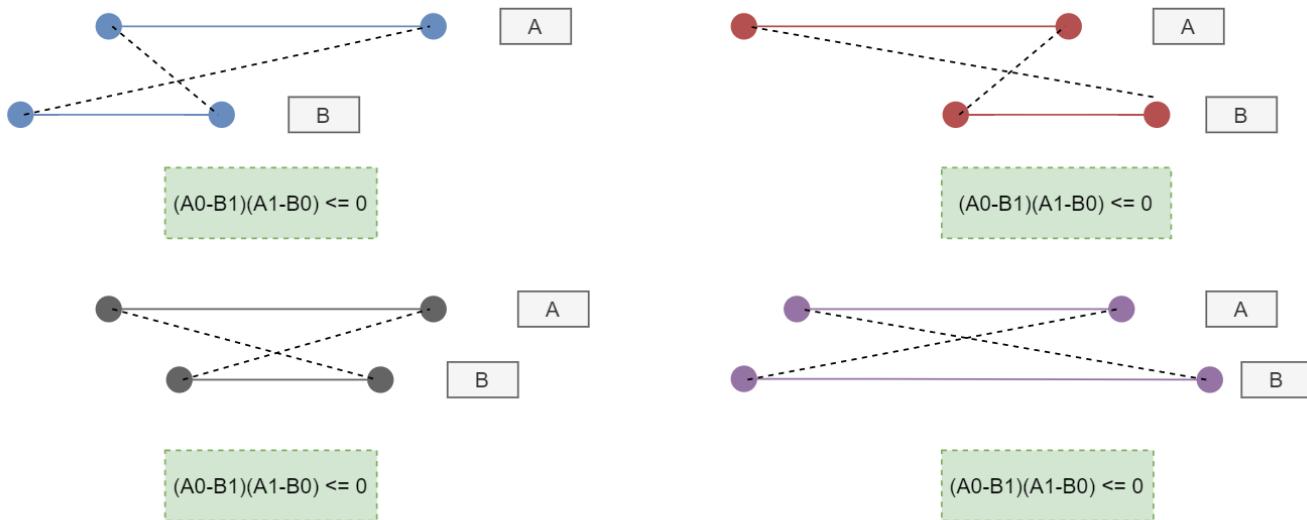
1. 输入为 []
2. 1,4],[2,4]
3. 边界情况 1,3],[3,4]
4. 复杂情况 1,2],[3,4],[5,6],[0,10],[7,12]

同时要求时间复杂度尽可能低。

分析

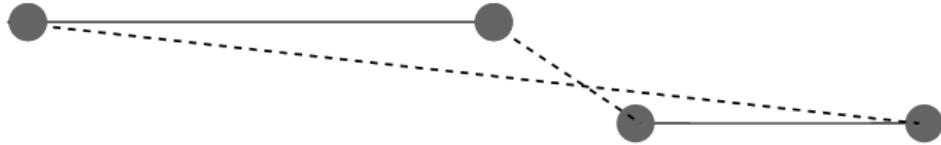
**S1**：首先考虑构成本题的基本结构：如何判断两个线段是否有交集：

两个线段有交集的四种情况：

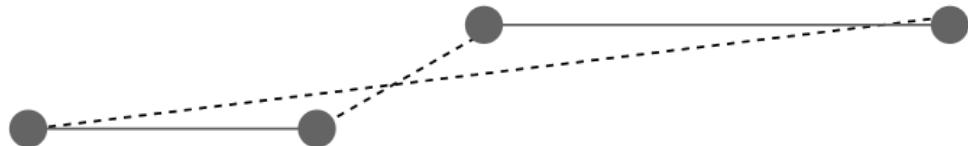


统一满足以上不等式条件： $(A_0 - B_1)(A_1 - B_0) \leq 0$

两个线段无交集的情况：



$$(A_0 - B_1)(A_1 - B_0) > 0$$

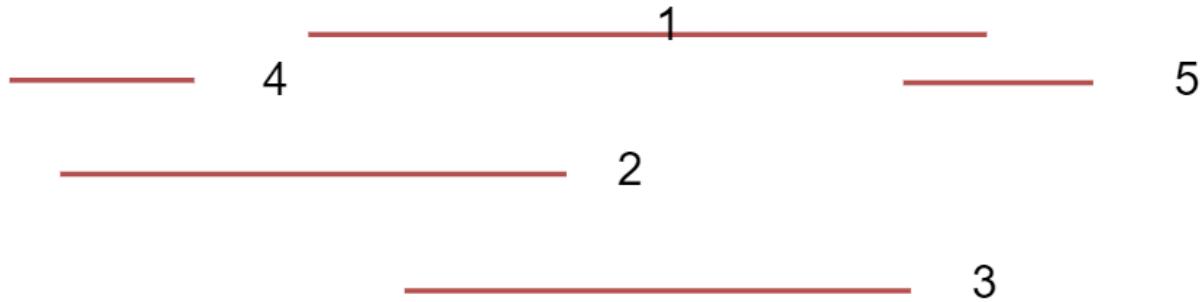


$$(A_0 - B_1)(A_1 - B_0) > 0$$

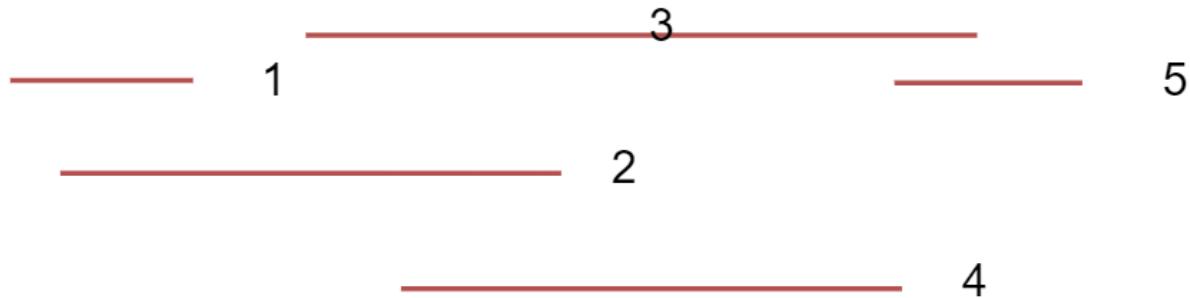
统一满足以上不等式条件:  $(A_0 - B_1)(A_1 - B_0) > 0$

**S2**：合并一堆线段

如下一堆线段，图中标号为线段的编号：



为了使得操作更加有序，第一感觉按照线段的左端点从小到大排序：



这样确保第  $i$  个线段左端点不大于第  $i + 1$  个线段左端点。

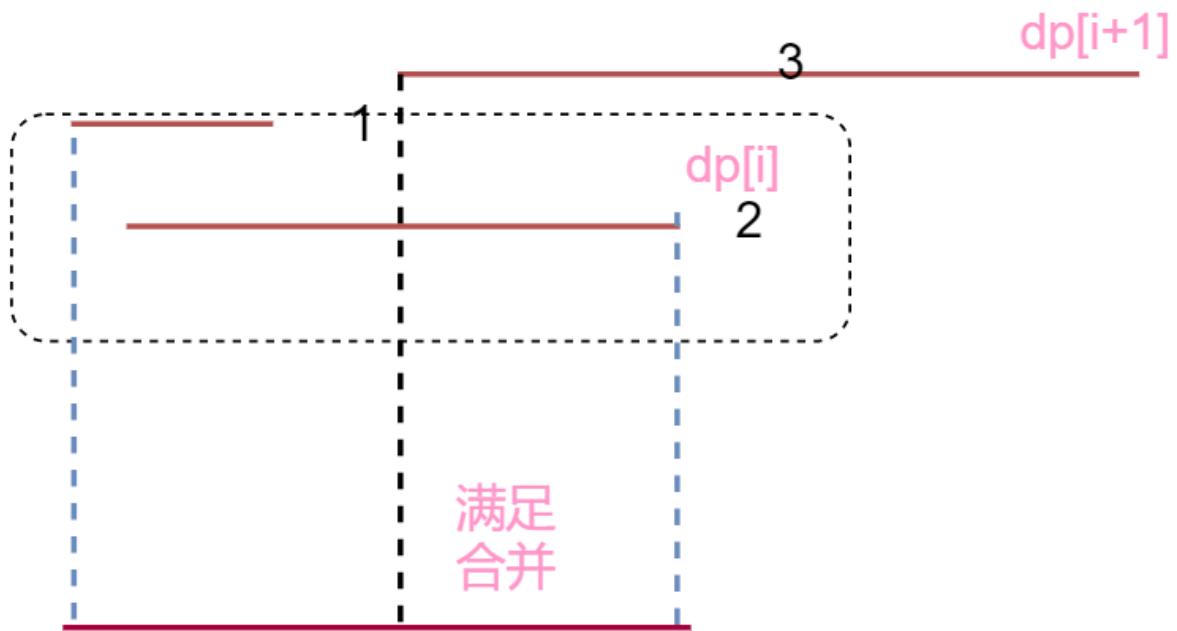
**S3**：设置数组  $dp$

设数组  $dp[i]$  表示合并第  $i$  条线段后的解，即此时数组  $dp$  内的各条线段都已经不能再合并。

下面就变为：

已知  $dp[i]$ ，如何合并第  $i+1$  条线段，进而推导出  $dp[i+1]$ 。

若数组  $dp$  中最后一个元素的右端点不小于第  $i+1$  条线段的左端点，则可与之合并，即满足：



$$dp[-1][1] \geq intervals[i + 1][0]$$

否则无法合并，直接加入到 `dp` 中。

转化为代码：

```

if item[0] <= dp[-1][1]:
    dp[-1][1] = max(dp[-1][1], item[1]) # 合并第i+1段
else:
    dp.append(item) # 无法合并, 直接加入到dp中

```

`dp` 数组的初始值为排序后的第一条线段。

**S4**：转化代码

```

class Solution(object):
    def merge(self, intervals):
        if len(intervals)==0: return intervals
        a = sorted(intervals,key=lambda x: (x[0],x[1]))
        dp = [a[0]]
        for item in a[1:]:
            if item[0] <= dp[-1][1]:
                dp[-1][1] = max(dp[-1][1],item[1])
            else:
                dp.append(item)
        return dp

```

一般排序时间复杂度为  $O(n\log n)$ ，后面 for 循环时间复杂度为  $O(n)$ ，所以综合时间复杂度为  $O(n\log n)$ 。

本题目先试用排序降低一部分分析难度，后面使用动态规划思想，问题变化为：已知  $dp[i]$  如何求出  $dp[i+1]$  问题。

## Day37 移动零

### 题目

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

示例：

输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数

### 分析

必须在原数组上操作，不能拷贝额外的数组；同时尽量减少操作次数，说白了就是想叫我们写出

更好的算法。

如何分析？

观察

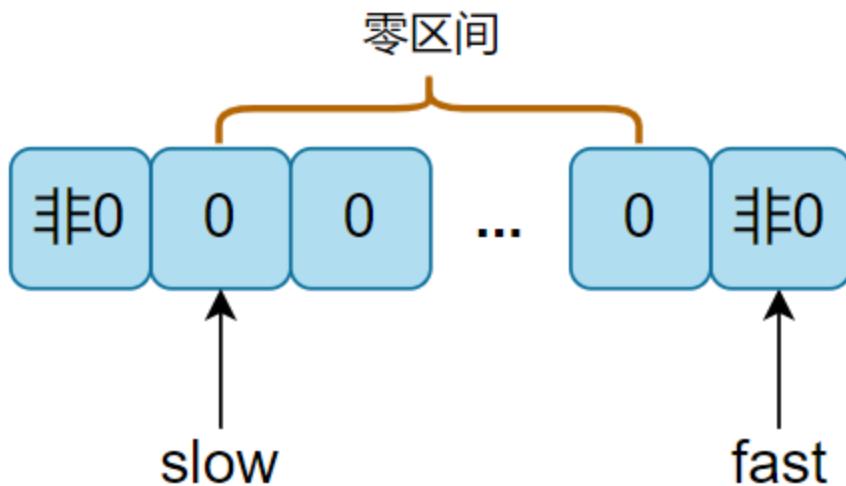
输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

整个过程就是0元素不断后移，非零元素不断前移的过程，所以算法每步操作的目标便是：逐渐达成这个分布规律。

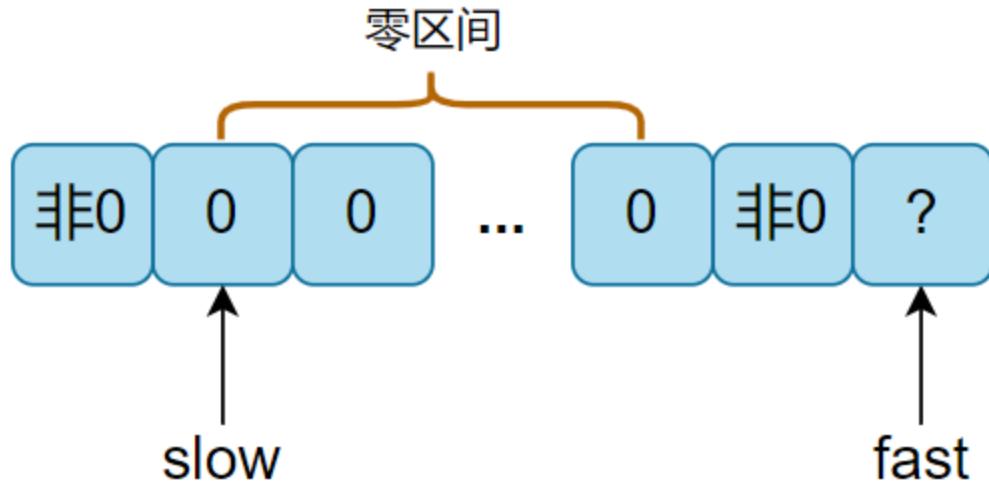
怎样优化操作？

假设两个指针 `slow` 和 `fast` 分别指向连续零区间的第一个0，最后一个0的后一个位置，如下图所示：



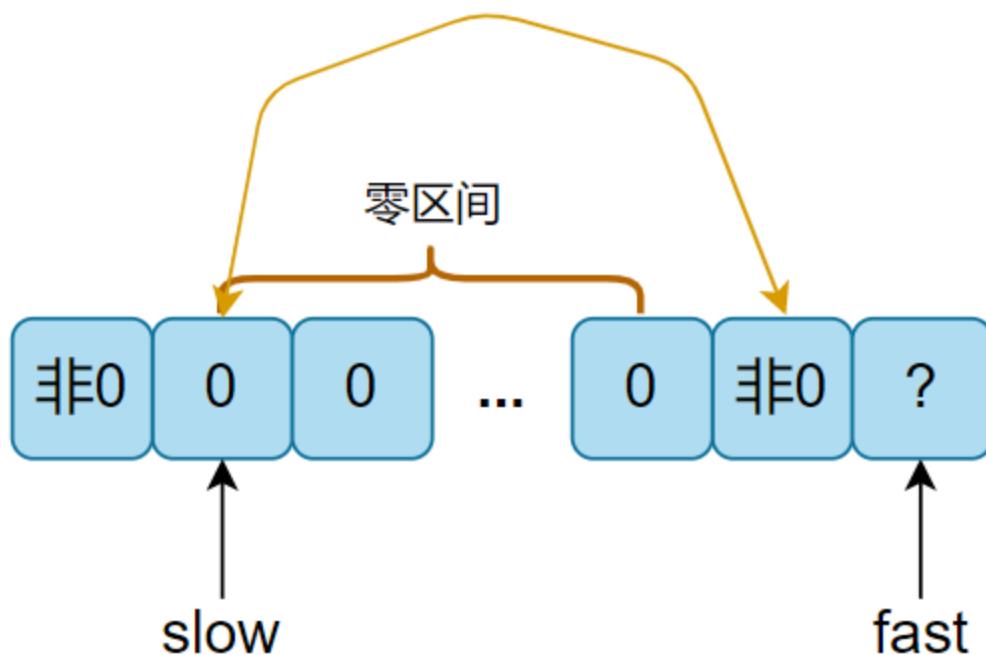
那么，`fast-slow` 正是索引从 `0~fast` 区间范围内0元素的个数。

`fast` 指向下一个元素：

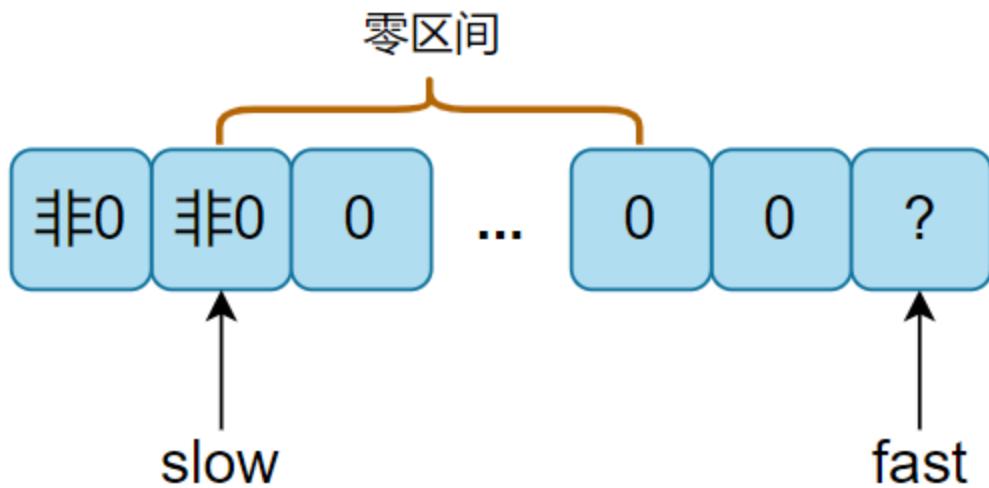


若打问号元素为 **0**，根据每步操作的目标是非零元素前移，零元素后移。所以迭代到此处时它已经为**0**元素，所以至少肯定不用前移，那么就保持原地不动。

若打问号的元素取值非 **0**，根据每步操作的目标是非零元素前移，零元素后移。因为 slow~fast 这块都为 **0**，所以为了目标，非零元素要和第一个 **0**交换，这样不就实现非零元素前移，零元素后移的目标了吗



交换后：



你看确实前进一步了吧。

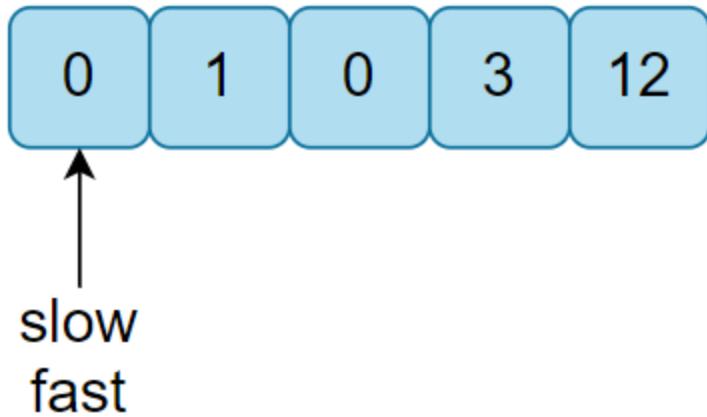
## 求解代码

以上分析过程就是此问题的一个中间状态的操作分析，是从第  $i$  次迭代状态到第  $i+1$  次迭代状态的变化过程。

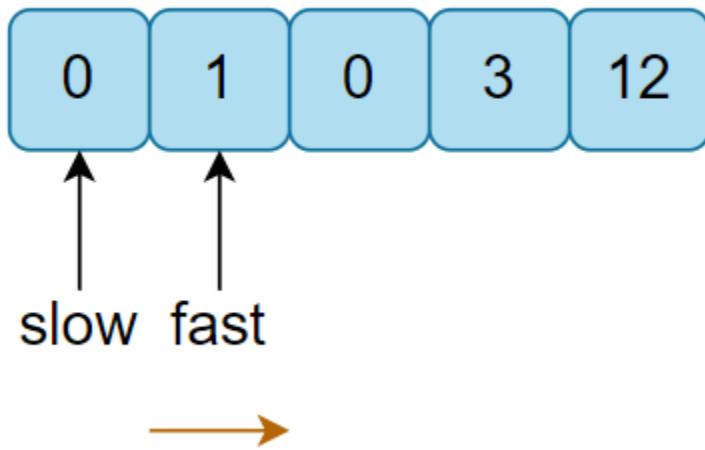
有了这个状态更新方程，因此就会很自然的写出下面的代码：

```
class Solution(object):
    def moveZeroes(self, nums):
        fast, slow=0, 0 # 分别指向连续零区间的最右侧、最左侧
        while fast<len(nums):
            # if nums[fast]==0 do nothing
            if nums[fast]!=0:
                # if fast == slow shows zero isn't found yet
                if fast > slow:# zero exists
                    nums[slow], nums[fast] = nums[fast], 0
                slow += 1
            fast += 1
```

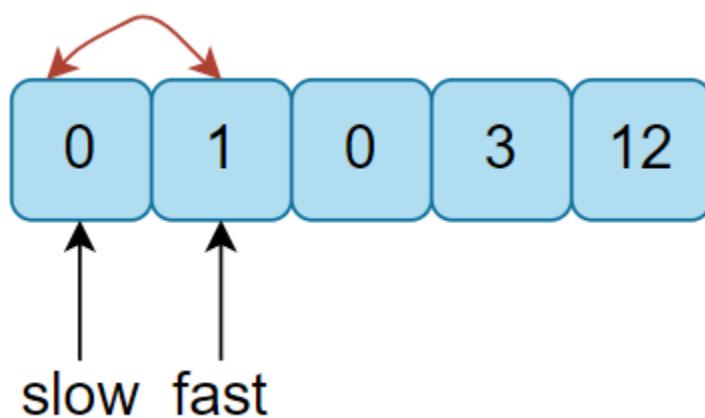
S1，设定两个指针初始分别指向第一个元素：



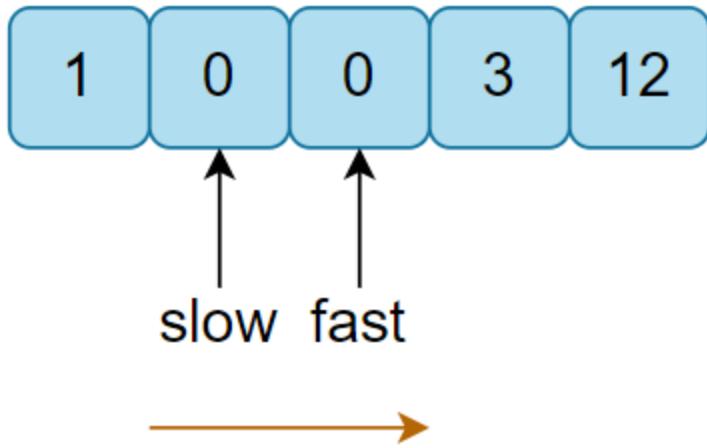
S2, 第一个元素等于0, 仅 fast 前进1步 :



S3, 下一次迭代时, fast 指向元素不为0, 则交换 :

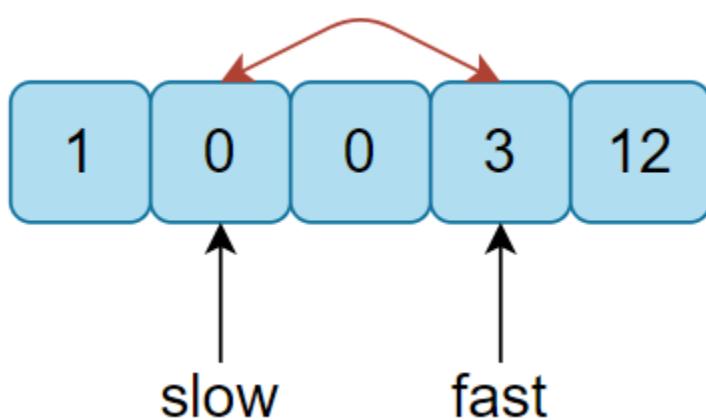
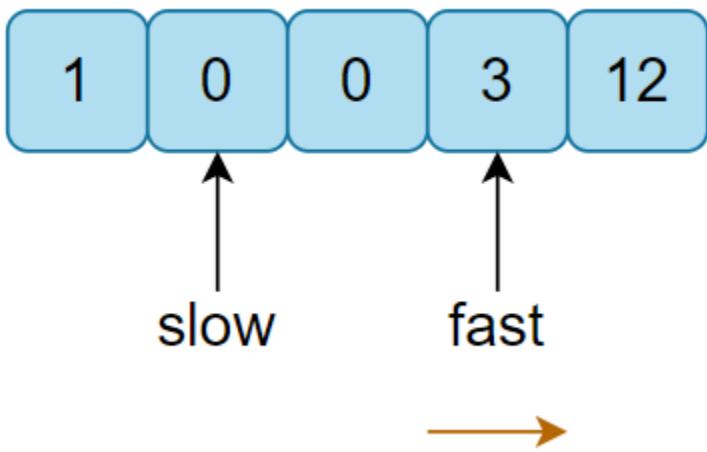


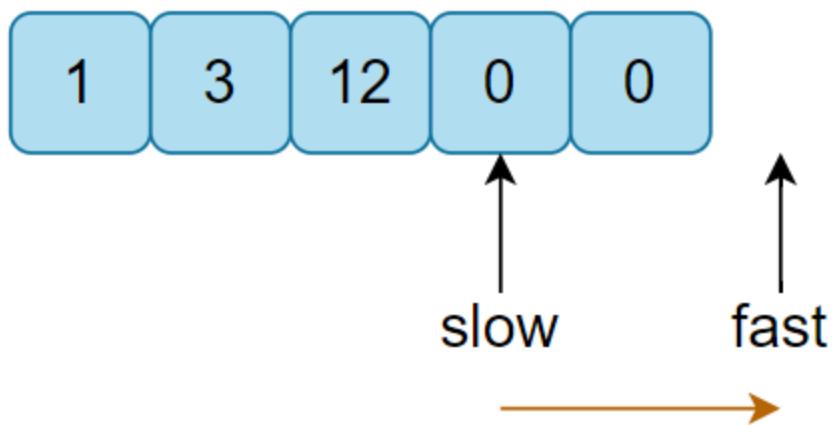
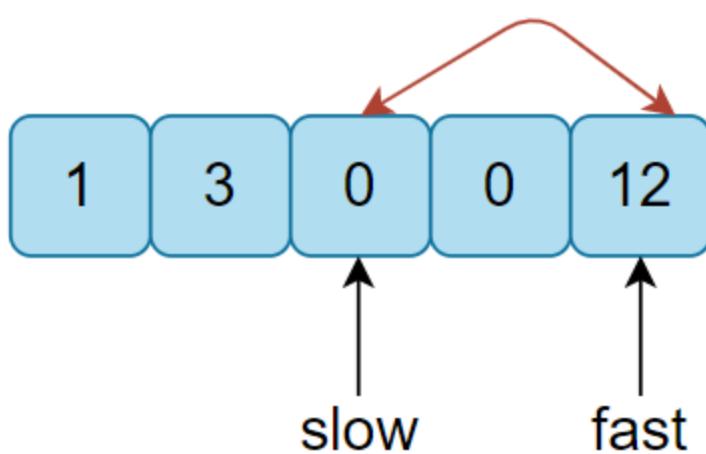
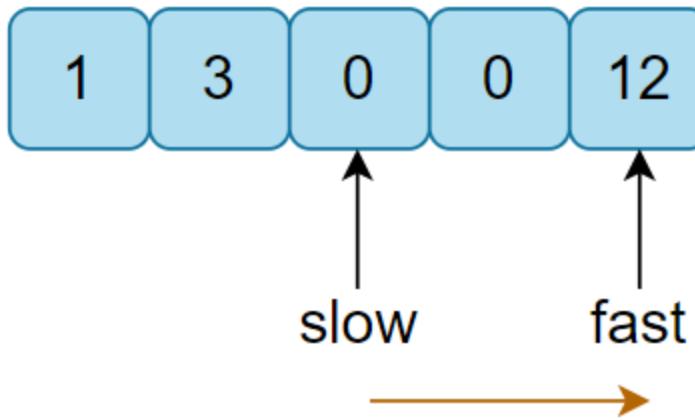
同时 slow 和 fast 同时都前进一步 :



S4，此时元素等于 0，此情况重复步骤 S2，因此重复上面操作。

依次类推，罗列出中间的各个状态：





`fast` 到头，程序结束。

可以看到 `slow` 指向连续零区间的第一个0，`fast` 指向连续零区间的最后一个0的后一个位置。

这与文章中分析中间状态的过程一脉相承，验证分析过程是准确的。

以上就是 移动零 这道题的详细分析。

# Day38 最大连续1的个数

给定一个二进制数组，计算其中最大连续1的个数。

示例 1:

输入: [1,1,0,1,1,1]

输出: 3

解释: 开头的两位和最后的三位都是连续1，所以最大连续1的个数是 3.

注意：

输入的数组只包含 0 和 1。

输入数组的长度是正整数，且不超过 10,000。

请补全下面代码：

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

分析

假定：

假定第 0~i 个元素，连续1的最大长度为 maxi

迭代到第 i 个元素时，连续1的长度为 count

$f(i+1)$  表示迭代完第  $i+1$  个元素时连续1的最大长度，则分两种情况：

若第  $i+1$  个元素为 0，则  $f(i+1) = \max(maxi, count)$ ，连续1的最大长度 count 置 0

若第  $i+1$  个元素为 1，则  $f(i+1) = \max(maxi, count+1)$ ，连续1的最大长度 count 加 1

代码

根据以上分析，写出如下代码：

```
class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        maxi, count = 0, 0
        for num in nums:
            if num == 0:
                maxi = max(maxi, count)
                count = 0
            else:
                maxi = max(maxi, count + 1)
                count += 1
        return maxi
```

星友孙颖颖颖颖颖，还写出一个解法，非常不错。

巧妙之处在于下面这个公式：

$$sum_1 = sum_1 \times i + i$$

它实现了遇1加1，遇0置0的功能，非常巧妙：

```
class Solution(object):
    def continueOne(self, nums):
        sum1, res = 0, 0
        for i in nums:
            #遇1加1，遇0置0
            sum1 = sum1 * i + i
            if sum1 > res:
                #记录连续1的长度
                res = sum1

        return res
```

# Day39 找到重复数

## 1 题目

天这个题目，大家务必注意理解清题目，然后再做。

给定一个包含  $n + 1$  个整数的数组  $\text{nums}$ ，其数字都在 1 到  $n$  之间（包括 1 和  $n$ ），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1：

输入: [1,3,4,2,2] 输出: 2

示例 2：

输入: [3,1,3,4,2] 输出: 3

一定注意以下几点要求：

1 不能更改原数组（假设数组是只读的）。

2 只能使用额外的  $O(1)$  的空间。

3 时间复杂度小于  $O(n^2)$ 。

4 数组中只有一个重复的数字，但它可能不止重复出现一次。

补全代码：

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

## 2 分析

以上4个稍微苛刻点的条件，若想都满足，本文提供两种求解方法。

### 2.1 二分查找

通常来说，二分查找是对值的二分。

就本题，在已知的条件下，却非常适合对索引二分。

因为题目假定元素取值范围： $[1, n]$ ，而数组长度为  $n+1$ ，所以  $\text{nums}[i]$ ,  $i$  最大为  $n$ ，都不会越界。

那么如何利用好这个条件，是本题求解的关键。

我们思考如下问题：

如果所有元素都不重复，那么不大于  $i$  的个数一定为  $i$  个，考虑如下序列：

$[1, 4, 2, 3]$

不大于 2 的个数为 2，不大于 3 的个数为 3.

如果出现不大于  $i$  的个数大于  $i$ ，则意味着一定有重复元素，且重复元素值不大于  $i$ . 考虑如下序列：

$[1, 4, 2, 4, 4]$

不大于 4 的元素个数为 5 个，则表明重复元素一定位于  $[1, 4]$  区间，而不可能出现在  $(4, 5]$  区间。

因此，归纳出根据索引的二分条件找到重复元素的方法：

找到中间索引，即  $\text{mid} = 0 + \text{len}(\text{nums}) - 1$

若  $\text{nums}$  中不大于  $\text{mid}$  的元素个数小于或等于  $\text{mid}$ ，则表明重复值位于  $[\text{mid} + 1, \text{len}(\text{nums}) - 1]$  若大于  $\text{mid}$ ，则一定位于区间  $[0, \text{mid} - 1]$  中

兑现为代码如下：

```
class Solution(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums)-1
        while left <= right:
            mid=(left+right)//2
            count = sum(i<=mid for i in nums) # 生成器迭代求和
            if count <= mid:
                left = mid +1
            else:
                right = mid-1
        return left
```

二分法的代码一定要注意：等号和left,right取值的变化，我看有些星友的代码写成：right=mid，这种肯定是有问题的，因为下一轮迭代mid的值不变，程序进入死循环。

再有，下面的等号是必须要有的，因为个数等于mid，也意味着重复值不可能在[0,mid]里：

```
if count <= mid:
    left = mid +1
```

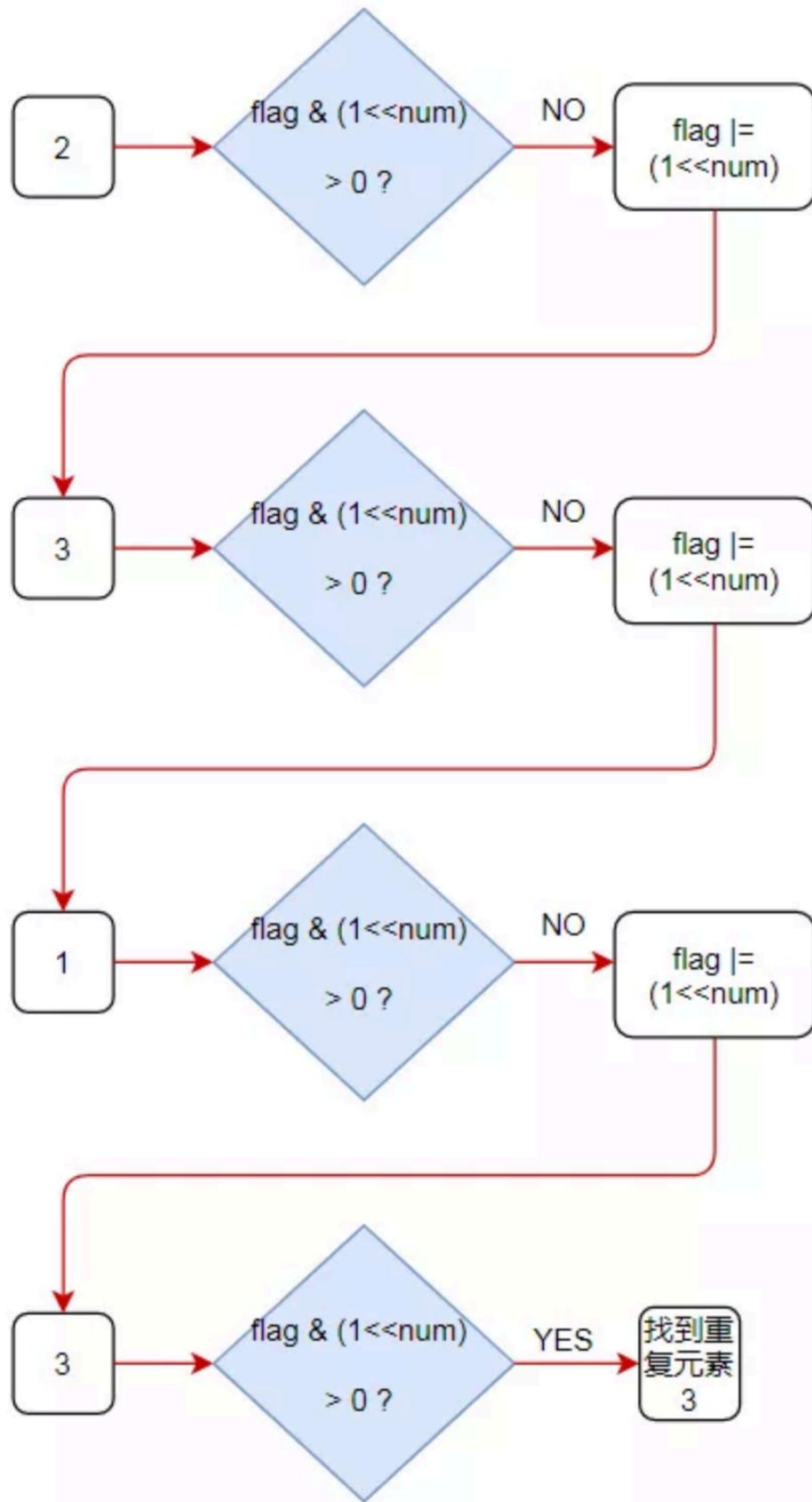
第二种解法：

位运算

大家思考下面的输入序列：[2,3,1,3]，如何使用位运算找出重复值：其中， $1 \ll num$  实现1向左移动num位；| 表示对应位的或运算& 表示对应位的与运算

flag初  
始值 0

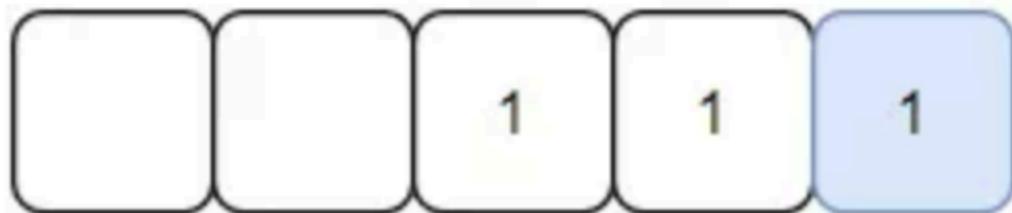
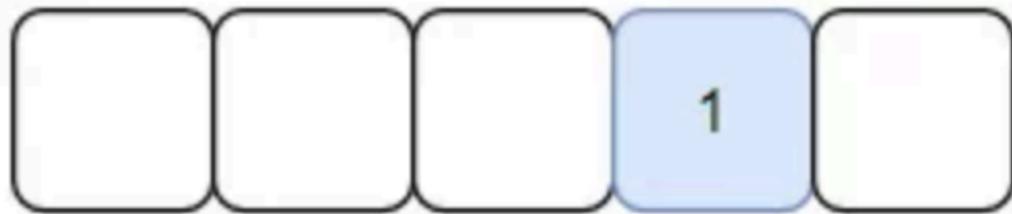
输入  
元素



每次迭代时， $\text{flag} |= (1 \ll \text{num})$  实现各个数字填充到对应位，通过这种巧妙的编码方法后：1 被

编码为  $1 \ll 1$ , 2被编码为  $1 \ll 2$ , 3被编码为  $1 \ll 3$ , 依次类推。

如下[2,1,3]运算完成后 flag 等于如下最后一行：



巧妙的标记着数字2,1,3.

这样当元素3再来时，再与flag后若大于0，就确保它是重复值：

`flag & (1<<num) > 0 ?`

这样代码如下所示：

```
class Solution(object):
    def findDuplicate(self, nums):
        flag = 0
        for num in nums:
            if flag & (1<<num) > 0:
                return num
            flag |= (1<<num)
return
```