我是封面

打印留白，我是背面

目录

第一章 写在前面

最初的想法是把刷题过程中最常见的、最基础的算法记录一下，比如排序算法、树的遍历、图的遍历等，然后打印出来上下班地铁上看看，利用碎片时间多次复习把它们“硬编码”到脑海里，以希望能够把这些内容掌握足够熟练后，就算做不到触类旁通也不要在基础上栽跟头，没想到越整理越多，这篇文档也就百十来页，但却耗费了我极大的精力，很多章节第二次看的时候推翻第一次写的，第三次看的时候又推翻第二次写的，让人头大。说实话如果这篇文档没什么人看倒也无所谓，我担心的是有人认真看了，结果我整理的是错的，那会给我带来极大的内疚感，所以**我希望任何一个下载此文档的人，带着批判的态度去看，汲取你所需的，正确的知识**。

这篇文档的内容没什么深度，虽然我算是科班出身，但我从没想过我会去写代码，直到毕业三年同学们的薪资普遍是我的三倍之多，我成了酸菜鱼（又酸又菜又多余），此时的我想转码真的是心有余而力不足，只能从基础一点一点啃。希望整理的内容能对大家有所帮助。

关于用什么代码，没必要一定分出哪种代码最好，我虽对 Java 不太了解，但在国内，我一看到阿里的研发岗就知道十有八九是和 Java 有关，如果是头条的研发岗，大概率和 Python 有关，不是说他们不用其他语言，是对这种语言的岗位需求量更大，岗位需求量大就意味着你可以尝试不同的组去面试，你的机会就更多，**如果你不是出类拔萃的候选者，机会的多少对你很重要**。当然这是国内的情况。

工作中用什么语言，很多时候你没有选择，需要用什么你就学什么，但为了面试选择哪门语言，要考虑的因素很多，你的目标公司，你的目标岗位，你所处区域的互联网环境等等。当然**如果确定你的意向公司重点是看算法，和用什么语言无关，那就看自己喜好了。但没有哪种语言是万能的，如果你体会不到一种语言的缺点，就说明对这种语言的理解不够深刻，或者用的不够多**。

Python 最大的特点是简洁的代码和丰富的库函数，**很多人追求一行代码，但是刷题和面试中一定不要用复杂的一行代码，或者说在你对 Python 足够熟练之前不要这么做**。我曾花费很多的时间去看 StefanPochmann 的代码，但当我刷第二次的时候，这是什么鬼？我又花费了很多时间去重新理解，我意识到这么优雅的代码并不适用于现阶段的我，**pythonic 是其次的，逻辑清晰才是最重要的**。如果面试官不用 Python，你要和他怎么解释比较复杂的一行代码？逻辑清晰的一行代码让人眼前一亮，比如 `nums = [i for i in range(10)]` 生成 0 到 9 的一维数组，用什么语言的基本都能看懂，复杂的一句话要慎用。

这篇文档的排版是有些问题的，比如行间距太小看着吃力，最初我的注意力是放在代码排版上，直接从 VS 粘过来，然后尽量让代码不要跨页，等到发现行间距问题的时候，几乎无力回天了，改一下行间距后面的代码基本上就面目全非，所以，大家将就吧。

最初的想法是把刷题过程中最常见的、最基础的算法记录一下，比如排序算法、树的遍历、图的遍历等，然后打利用碎片时间多次复习把它们“硬编码”到脑海里，以希望能够把这些内容掌握足够熟练后，就算做不到触类旁通也不要把基础忘掉，没想到越整理越多，这篇文档也就百十来页，但却耗费了我极大的精力，很多章节第二次看的时候推翻第一次写的，第三次看的时候又推翻第二次写的，让人头大。说实话如果这篇文档没什么人看倒也无所谓，我担心的是有人认真看了，结果我整理的是错的，那会给我带来极大的内疚感，所以**我希望任何一个看到此文档的人，带着批判的态度去看，汲取你所需的，正确的知识**。

工作中用什么语言，很多时候你没有选择，需要用什么你就学什么。**但没有哪种语言是万能的，如果你体会不到一种语言的缺点，就说明对这种语言的理解不够深刻，或者用的不够多**。

Python 最大的特点是简洁的代码和丰富的库函数，**很多人追求一行代码，但我的建议是：一定不要用 复杂的 一行代码，或者说在你对 Python 足够熟练之前不要这么做**。我曾花费很多的时间去看 StefanPochmann 的代码，但当我刷第二次的时候，这是什么鬼？我又花费了很多时间去重新理解，我意识到这么优雅的代码并不适用于现阶段的我，**pythonic 是其次的，逻辑清晰才是最重要的**。逻辑清晰的一行代码让人眼前一亮，比如，要生成 0 到 9 的一维数组，nums = [i for i in range(10)]，用什么语言的基本都能看懂，这种简单的一句话能让代码更简洁，但复杂的一句话一定要慎用。

这篇文档的排版是有些问题的，比如行间距太小看着吃力，最初我的注意力是放在代码排版上，直接从 VS 粘过来，然后尽量让代码不要跨页，等到发现行间距问题的时候，几乎无力回天了，改一下行间距后面的代码基本上就面目全非，所以将就着看吧。

第二章 易混淆概念

2.1 子串

2.2 子序列

1. 子串：原字符串的一部分，空字符也属于子串。bc 是 abcd 的子串
2. 子序列：**子串要求连续，子序列可以不连续，但字符顺序不能变**，bd 是 abcd 的子序列。
3. 子串的个数是 n(n+1)/2+1，时间复杂度是 O(n^2)，每个字符都不同的情况下，包含 1 个字符的子串共 n 个，包含 2 个字符的子串共 n-1 个，包含 3 个字符的子串共 n-2 个，还有一个空串。
4. 子序列的个数是 2^ n 个（包含空串），计算时候可以用动态规划：

   假设子序列的前 k 个数的子序列个数为 d(k)，那么前 k - 1 个子序列的个数就为 d(k - 1)个子序列，从 k - 1 到 k 的变化是怎样的呢？

   1、假设数组 a[N]第 k 个数为 a[k]，**如果 a[k] 与前面的 k - 1 个数都不相同**，那么就有 : d(k) = d(k - 1) + [d(k - 1) + 1] = 2d(k - 1) + 1，为什么呢？可以这样想，对于前 k - 1 项的子序列个数为 d(k - 1)，那前 k 个数，无非就是在前 k - 1 项的基础上多加了一个 a[k]，这就有 d(k - 1)个，还有一个 a[k]自己，所以还要加 1；

   2、**假设 a[k] 与前面的 k - 1 个数其中一个相等**，那就会有重复的部分，如何找到重复的部分呢，假设离 k 最近的一个与 a[k]相等的数为第 t 个 a[t] = a[k]，即序列(a[1], a[2], ……,a[t],……,a[k - 1],a[k])，a[t] = a[k]；我们已经知道序列(a[1], a[2], ……,a[t])的序列个数为 d(t)，那么 d(t - 1)就是重复的部分，因为 d(t - 1)加上 a[t]和加上 a[k]是相同的，为什么只需要找到离 k 最近的 t 使得 a[t] = a[k]？我们是从 1 - n 对数组进行遍历的，计算 d(i)的 i 就是从 1 到 n 依次计算的，那么第一次遇到 a[k] = a[t]的情况满足条件：有且仅有一个 t 使得 a[t] = a[k]，比如序列(1, 2, 3, 2, 4, 2)，分别计算 d(1),d(2),d(3),d(4),d(5),d(6)；我们在计算 d(4)的时候发现 a[4] = a[2]（假设下标从 1 开始），所以 d(4) = 2*d(3) - d(2 -1) = 2d(3) - d(1)；当计算 d(6)的时候也有 a[6] = a[4] = a[2]，但是由于我们前面已经把 a[2]重复的部分减掉了，所以不需要再减，d(6) = 2 * d(5) - d(4 - 1) = 2d(5) - d(3).

比如 491. Increasing Subsequences，回溯求不重复的递增子序列，对应时间复杂度，总共有 n 个数字，对于每个数字都有两个选择：留下或丢掉，所以 2 * 2 * 2 * 2 ..... = O(2 ^ n)，这也是所有子序列的个数。

```
def findSubsequences(self, nums):
    if len(nums) < 2: return []
    res = []
    self.dfs(nums, 0, [], res)
    return res

def dfs(self, nums, start, path, res):
    if len(path) >= 2:
        # 注意浅拷贝，否则 path 发生变化会影响已添加到 res 中的数组
        res.append(path[:])
    # 用于去重
    visited = set()
    for i in range(start, len(nums)):
        if (not path or path[-1]<=nums[i]) and nums[i] not in visited:
            visited.add(nums[i])
            path.append(nums[i])
            self.dfs(nums, i+1, path, res)
            # 弹出上一个入栈的元素，回溯到下一个元素，
            # 若上一行传参用 path+[nums[i]]，则不必执行下面的 pop
            path.pop()
```

但是像 300 题求最长上升子序列，只求最大长度，可以用 DP 可以把时间复杂度降到 O(n^2)，用贪心 + 二分查找可以降低到 O(NlogN)。

原文链接：

https://blog.csdn.net/thebestdavid/article/details/11908961

https://www.xuebuyuan.com/3195723.html，

https://zhuanlan.zhihu.com/p/80757470

2.3 全排列

n 个数的全排列为 n!

第三章 Python 基础

3.1 面向对象

1. **全局变量：**在模块内、在所有函数外面、在 class 外面，这就是全局变量。如果对全局变量只读取不更新的话，不需要加 global，但要更新全局变量的话要用 global 声明一下。

2. **局部变量：**在函数内、在 class 的方法内（未加 self 修饰）。

3. **类(Class)：**用来描述**具有相同的属性和方法的对象的集合**。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。在类的声明中，**属性是用变量来表示的**，所以提到的属性其实也就是变量。

4. **静态变量：**也称为类变量，类变量定义在类中且在函数体之外，也就是在 class 内但在 def 外，一般用于定义常量和默认值。不管在类的内部还是外部，都建议用**【类.变量名】的形式获取类变量**，外部还可以用【实例名.变量名】来获取。类变量在整个类的代码块里面类似是作为全局变量，类变量通常不作为实例变量使用。

5. **实例变量：实例变量就是用 self 修饰的变量，一般在“__init__”中首次出现**。对于类，有了静态变量，可以供内部和有继承关系的父子之际使用，同步；但实例之间各自的私有变量（局部变量）就要靠实例变量了，实现了动态绑定，多态特性。内部调用时都需要用【self.实例变量】的形式，外部调用时用【实例名.变量名】的形式。

```
class Person():
    person = 'alive'   # 这是类变量
    def __init__(self,name ,age):
        self.name = name # 这是实例变量

if __name__=="__main__":
    p = Person('张三' , 18)
    """类变量 """
    print(Person.person)   #【类.变量名】的形式输出结果：alive
    print(p.person)        #【实例名.变量名】的形式输出结果：alive
    """实例变量 """
    print(Person.name) #【类.变量名】的形式，错误，
    print(p.name)      #【实例名.变量名】的形式输出结果：张三
```

6. 为什么不直接在类下面写，然后使用 cls.xxx 访问，self 和 cls 区别？

类属性（变量）是所有实例共享一份，需要看情况，需要多实例并且属性互不干扰需要使用 self，需要所有实例保持使用同一个属性使用类属性。就比如上面代码，“张三”或“李四”是 Persion 的两个实例，他们共有的属性是 alive，那 alive 就需要用类变量，他们的名字年龄不同，就需要用实例变量。

【python 什么时候加 self，什么时候不加 self】

7. **私有变量**：python 中并没有私有变量，但是通过取名可以部分实现私有变量的效果。不希望被外部访问到的属性取名时，前面应该加上两个下划线“__”，这不仅仅是个标志，而且是一种保护措施。

8. **方法：类中定义的函数**。

3.2 可变/不可变对象

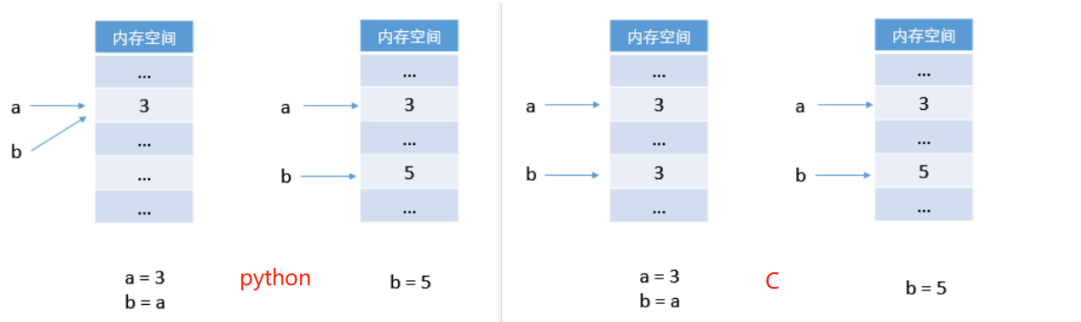
Python 中的变量存放的是对象引用，一个变量发生变化，可能是对象本身发生了变化，也可能只是改变了引用而对象本身没变。

3.2.1 不可变对象

不可变对象有数字、字符串和元组。对于不可变对象，对象本身一旦创建就不可变，只能改变对象的引用，看下面代码：

```
if __name__ == '__main__':
    a = 3
    b = a
    b = 5
    print("a:", a)  # a: 3
```

对比一下 Python 和 C 的实现方式：



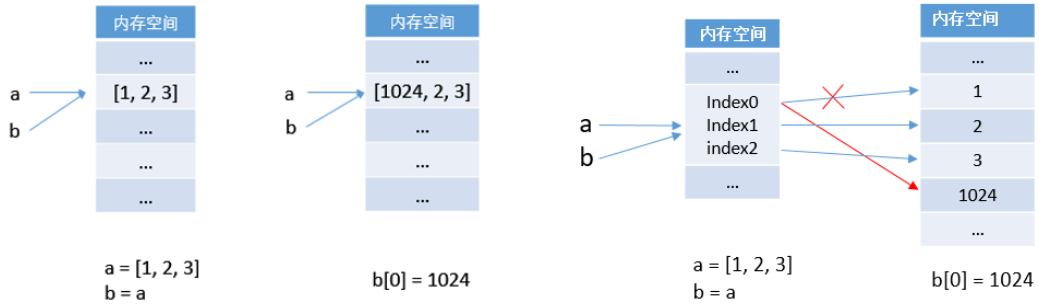
也由于数字是不可变对象，所以 python 中没有 C 语言的自增（++）和自减（--）运算符，它只能从一个对象指向下一个对象，可以这样写 a += 1。

3.2.2 可变对象

可变对象有列表、字典和集合。可变对象在创建后可以被修改：

```
if __name__ == '__main__':
    a = [1, 2, 3]
    b = a
    b[0] = 1024
    print("a:", a)  # a: [1024, 2, 3]
```

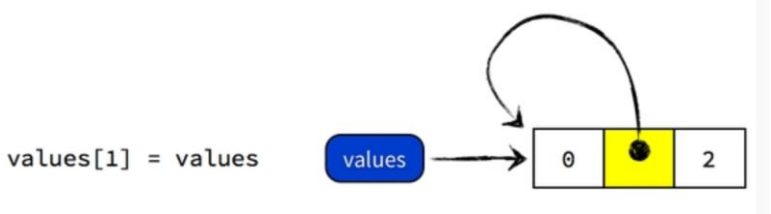
其实列表里面的数字依然是不可变对象，被改变的并不是这个数字，而是列表下标指向的引用：



来看一个例子：

```
if __name__ == '__main__':
    values = [0, 1, 2]
    values[1] = values
    # values[1] = [0, 1, 2]
    # values[1] = values[:]
    # values[1] = values.copy()
    print(values)   # [0, [...], 2]
```

我们预想的结果是 [0, [0, 1, 2], 2]，但实际上是 [0, [...], 2]，首先 values 指向了对象 [0, 1, 2]，接着又把 values 的第二个下标指向了 values 所引用的对象：



想把 values[1] 修改成理想状态的话可以用注释部分的代码，1. 直接指定要修改成 [0, 1, 2]，虽然这和 values 看起来一样，但因为是可变对象，所以实际上两个列表的 id 并不同；2. 浅拷贝，切片和 copy()其实都是浅拷贝。浅拷贝是拷贝父对象，也指向父对象的引用，但它本身的 id 和父对象不同。

Python 中 id 是对象的唯一标识符，id() 函数用于获取对象的内存地址，来看看如下几个对象的 id：

```
if __name__ == '__main__':
    v1 = [0, 1, 2]    # 2378553397760
    v2 = v1           # 2378553397760
    v3 = [0, 1, 2]    # 2378553399488
    v4 = v1[:]        # 2378553884480
    v5 = v1.copy()    # 2378553884544
    print(id(v1), id(v2), id(v3), id(v4), id(v5))
```

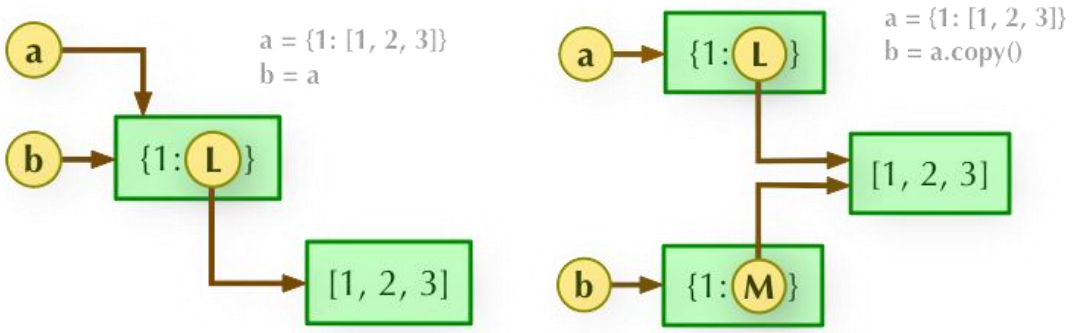
可以看出普通赋值的时候，v1 和 v2 的内存地址是一样的，所以不管修改 v1 还是 v2，另一个也会跟着改变；v3 看起来虽然和 v1 一样，但因为是可变对象，所以虽然它们元素下标指向一样的数字，但 v3 和 v1 的内存地址本身不一样，v4 和 v5 都是浅拷贝，它们的 id 也和 v1 不同，所以 v3、v4 和 v5 不受 v1 和 v2 的影响。

3.3 浅/深拷贝

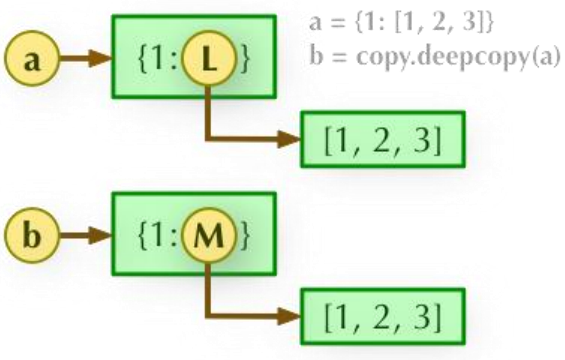
浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。

深拷贝(deepcopy)：copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。

普通赋值和浅拷贝的区别如下，可以看到子对象其实是一样的：



深拷贝和浅拷贝的区别体现在多层嵌套时候，如下字典的 value 是列表：



可以看到深拷贝把子对象也拷贝了一份，浅拷贝和深拷贝仅仅是对组合对象也就是可变对象来说的，所谓的**组合对象就是包含了其它对象的对象，如嵌套的列表、字典、类实例等**。而对于数字、字符串以及其它“原子”类型，没有拷贝一说，产生的都是原对象的引用，和赋值是一样的。

多维数组的初始化**不推荐使用 list * n 的形式**，例如下面二维数组：

```
>>> b = [[0] * 3] * 4
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> print(id(b[0]), id(b[1]), id(b[3]))
2524376102984 2524376102984 2524376102984

>>> b[0][1] = 5
[[0, 5, 0], [0, 5, 0], [0, 5, 0], [0, 5, 0]]
```

list * n 是浅拷贝，b[1], b[2], b[3] 都是 b[0] 的浅拷贝，它们指向的是同一地址，而且又是可变对象，所以任一个子元素被修改，都会影响到其它三个，一维数组则没事。

最好用 for 循环来创建：

```
>>> b = [[0 for _ in range(3)] for _ in range(4)]
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> print(id(b[0]), id(b[1]), id(b[3]))
2326272240328 2326272238088 2326272375624

>>> b[0][1] = 5
[[0, 5, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

通过循环赋初始值是一种比较推荐的初始化方法，不会因为对象引用的问题产生 bug。

【Python 赋值、浅拷贝和深拷贝】

3.4 模块和包的引入

Python 模块（module），一个 .py 文件就是个 module，比如 sample.py 其中文件名 smaple 为模块名字。

Python 包（package），一个带 __init__.py 的文件夹。

```
# 引用一个模块 module
import module

# 引用模块 module，用别名 md 代替。代码中只能写成 md，不能写 module
import module as md
```

```
# 仅引用 module 中的 xxx 函数。代码中可直接写成 xxx，无需使用 module.xxx
from module import xxx

# 引用 module 中的 xxx 函数，用别名 yyy 代替，代码中只能直接写成 yyy，不能写 xxx
from module import xxx as yy
```

import 语句可以在程序的任何位置使用，可以在程序中多次导入同一个模块，但模块中的代码仅仅在该模块被首次导入时执行，后面的 import 语句只是简单的创建一个到模块名字空间的引用而已。sys.modules 字典中保存着所有被导入模块的模块名到模块对象的映射。这个字典用来决定是否需要使用 import 语句来导入一个模块的最新拷贝。

from module import * 语句**只能用于一个模块的最顶层**.特别注意，由于存在作用域冲突，**不允许在函数中使用 from 语句**。

注意内置函数和内置模块的区别：内置函数是任何模块都可以直接调用，内置模块需要 import 模块之后才能调用模块里面的函数。

不同的模块中可以存在相同的变量名或者函数名，但是不能与内置函数名或者内置模块名字重复，避免造成冲突。

3.5 迭代器和生成器

挖坑

3.6 装饰器

挖坑

3.7 常用库函数

这一节提到的库函数，基本都是做题过程中常见的，其实我不建议做题过程中使用现成的库函数，**可以不用但不能不懂**，比如使用频率非常高的栈和队列，Python 中 list 就可以模拟栈和队列，但并不高效，比如 list.pop(0),时间复杂度是 O(n)，但双向队列 collections.deque 中的 popleft()方法却可以把复杂度优化到 O(1)。这些很常见的方法还是要理解，具体的使用场景详见“基础算法”一章。

3.8 随机数 random

Python 中的 random 模块包括以下方法：

1. random.random()，**左开右闭**，产生 0 到 1.0 之间的随机**浮点数**。

2. random.uniform(s,e)，**左闭右闭**，产生 s 到 e 之间的随机**浮点数**，可以指定区间，弥补了 random.random()。
3. random.randint(s,e)，**左闭右闭**，产生 s 到 e 之间的随机**整数**。
4. random.choice('abcd')，**左闭右闭**，从序列中返回一个的随机**元素**。
5. random.randrange(s,e,step) ，**左闭右开**，生成从 s 到 e 之间，符合步长为 step 的随机**整数**，如 random.randrange(1,6,2)，可能返回 1、3、5 中任意一个。
6. random.shuffle(x)，用于将一个列表中的元素打乱。
7. random.sample(sequence,k)，从 sequence 中随机获取 k 个元素，作为一个片段返回，sample 函数不会修改原有序列。

3.8.1 小根堆 heapq

Python 中提供的 **heapq 是小根堆**，有两种方式创建堆：
1. 使用一个空列表，用 heapq.heappush()函数把值加入堆中。
2. 使用 heap.heapify(list) 把列表转换成堆结构。

```
"""方法一"""
nums = [2, 3, 5, 1, 54, 23, 132]
heap = []
for num in nums:
    heapq.heappush(heap, num)  # 加入堆
print(heap)     # [1, 2, 5, 3, 54, 23, 132]

"""方法二"""
nums = [2, 3, 5, 1, 54, 23, 132]
heapq.heapify(nums)
print(nums)     # [1, 2, 5, 3, 54, 23, 132]

"""用 heappop 实现堆排序"""
print([heapq.heappop(nums) for _ in range(len(nums))])
# out: [1, 2, 3, 5, 23, 54, 132]
```

比如求最大/小的元素，top K 之类的问题，这时候堆排序的优势就出来了。用堆排序在 N 个元素中找到 top K，时间复杂度是 O(N log K)，空间复杂度是 O(K)。冒泡插入等排序算法是按线性顺序排列的，是 O(n)的复杂度（需要对 n 个数据进行移位处理）,堆排序利用的是完全二叉树，所以更高效。

heapq 的一些常用方法：
1. heapify(list)，将列表转换为小根堆的数据结构。
2. heappush(heap, x)，将 x 加入堆中。

3. heappop(heap)，从堆中弹出最小的元素。
4. heapreplace(heap, x)，弹出最小的元素，并将 x 加入堆中。
5. heappushpop(heap, x)，先把 x 加入堆中，再弹出最小的元素。
6. heapq.nlargest(n, heap)，返回 heap 中前 n 个最大的元素。
7. heapq.nsmallest(n, heap)，返回 heap 中前 n 个最小的元素。

3.8.2 双向队列 deque

虽然 list 可以通过 pop(0)和 append()来对两端进行插入删除，但 pop(0)的复杂度是 O(n)，想要优化到 O(1)就需要双向队列。

collections.deque() 是类似列表（list）的容器，实现了在两端快速添加（append）和弹出（pop），需要一个 iterable（迭代对象）作为参数，返回一个新的双向队列对象，如果 iterable 没有指定，新队列为空。**deque()支持线程安全**。

```
from collections import deque
if __name__ == '__main__':
    q = deque([1, 2, 3])
    q.append(4)       # [1, 2, 3, 4]
    q.appendleft(0)      # [0, 1, 2, 3, 4]
    q.popleft()       # [1, 2, 3, 4]
```

当然 deque 还包含其它很多方法，有兴趣可以自己查看一下。Python 的 **deque 对象以双向链表实现**，所以在指定位置插入删除是 O(1)，但查找指定元素的性能很差，耗时为 O(n)。

3.8.3 defaultdict

在普通 dict 中，如果使用不存在的 key 会发生 KeyError 报错，想避免这样的问题一般有三种方法，比如 dict 中并没有"a"这个 key，要操作"a"：

1. 先判断"a"是否在 dict 中，再进行其它操作。
2. 用 dict.get(key, default=None)，存在的话会返回 value，否则返回默认值，不指定的话默认是 None。
3. 用 collections.defaultdict(method_factory)，method_factory 可以是 list、set、dict、int、str、tuple，默认为 None。上面两种方法在简单的字典操作中可以应付，但比较复杂的时候，可能需要多次 if 逻辑判断来避免报错，代码量稍多，而使用 defaultdict()就很简单了，任何未定义的 key 都会默认返回一个 method_factory，具体使用的时候你就会发现它的便捷。还是那句话，你可以使用便捷的库函数，但也必须知道普通 dict 中操作不存在的键值会报错。

```
from collections import defaultdict
d1 = defaultdict(list)
```

```
d2 = dict()
print(d1['a'])  # []
print(d2['a'])  # KeyError: 'a'
```

3.8.4 Counter

collections.Counter()提供了计数功能，元素存储为字典 key，其计数存储为字典 value，访问不存在的 key 的时候，返回值为 0，有三种创建方式：

```
from collections import Counter
# 从一个可 iterable 对象（list、tuple、dict、字符串等）创建
c = Counter('gallahad')

# 从一个字典对象创建
c = Counter({'a': 4, 'b': 2})

# 从一组键值对创建
c = Counter(a=4, b=2)
```

使用方式如下：

```
# 统计单词数
words = 'alex hello world hello world hello'
w = collections.Counter(words.split())
print(w)
# Counter({'hello': 3, 'world': 2, 'alex': 1})
```

它还有一个 most_common(n)方法，返回一个有前 n 多的元素的列表：

```
from collections import Counter
c = Counter('abracadabra')
c.most_common(3)
#[('a', 5), ('r', 2), ('b', 2)]
```

如果 n 被忽略或者为 None，返回所有元素，相同数量的元素次序任意，需要 value 能够进行排次序。

3.8.5 Queue 模块

队列分类：

1. 线程队列 Queue — FIFO(先进先出队列)，即哪个数据先存入，取数据的时候先取哪个数据，同生活中的排队买东西

2. 线程队列 LifoQueue — LIFO(先进后出队列)，即哪个数据最后存入的，取数据的时候先取，同生活中手枪的弹夹，子弹最后放入的先打出

3. 线程队列 PriorityQueue — PriorityQueue(优先级队列)，即存入数据时候加入一个优先级，取数据的时候优先级最高的取出。

【python 线程队列 Queue-FIFO（35）】

3.9 常用内置函数

1. 有返回值：sorted()、reversed()、strip()、split()、join()、list()返回新串，所以需要一个变量来接收。
2. 无返回值：sort()、reverse()没有返回值，是对源对象进行操作。可以看一下细节差别：

```
s = [3, 6, 2, 7, 1, 9]
print(sorted(s))    # [1, 2, 3, 6, 7, 9]
print(s.sort())     # None
print(s)        # [1, 2, 3, 6, 7, 9]

# TypeError: 'NoneType' object is not iterable，没返回值，报错
for n in word.reverse():
    print(w)

# 9 1 7 2 6 3
for n in reversed(word):
    print(n)
```

3. sorted()、reversed()可以处理**任何可迭代对象**，返回一个排序或反转的迭代器。sort()、reverse()只是**列表的内置方法**。
4. strip()是返回移除字符串头尾指定的字符生成的**新字符串**，不改变类型。split()是返回分割后的字符串**列表**，通过指定分隔符对字符串进行切片，第一个参数是分隔符，分隔符默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。第二个参数是分割次数，默认为 -1，即分隔所有。
5. 拼接字符可以用 join() 和 "+"，简单拼接时候用 "+"，对一组元素进行拼接时候用 join()，如果在一个循环中用 "+" 拼接效率会很低，因为字符对象是不可变对象，每一次执行 "+" 操作就会创建一个新的字符串对象，最好是先收集所有的字符串片段然后再将它们连接起来，如 "".join(list)。
6. **type()不考虑继承关系**，不会认为子类是一种父类类型。**isinstance()会考虑继承关系**，认为子类是一种父类类型。如果要判断两个类型是否相同推荐使用 isinstance()。

```
>>> type('runoob')
<type 'str'>
>>> type([2])
<type 'list'>

>>>a = 2
>>> isinstance (a,int)
True
>>> isinstance (a,str)
False
>>> isinstance (a,(str,int,list))    # 是元组中的一个返回 True
True
```

7.

第四章 Python 细节

4.1 取模

1. **同号**取模，Python/C++/Java 都是**让商尽可能小**，-5 % -2 = -1 ，-5 % 2 = 1 ，5 % -2 = -1。
2. **异号**取模，**python 会让商尽可能小，C++/Java 会让商尽可能大**。

4.2 取整

1. int()，**向 0 取整**，int(-2.5) = -2， int(-1.5) = -1， int(1.5) = 1。

2. round()，银行进位法，四舍六入五取偶。**保留到离上一位更近的一端（四舍六入），如果距离两端一样远，会保留到偶数的一边**。具体就是看保留位的后一位：
   a. 如果小于 5 则舍，例如 round(10.4) = 10。
   b. 如果大于 5 则进，例如 round(10.6) = 11。
   c. 如果等于 5 且其后无数字，则要再看 5 的前一位，是奇数则进，是偶数则舍，例如 round(11.5) = 12 ，round(10.5) = 10 。

   注意：**实际操作中经常会出现与理论不符的结果**，比如 round(2.675, 2) = 2.67，实际上应该是 2.68，这跟浮点数的精度有关。我们知道在**机器中浮点数不一定能精确表达**，因为换算成一串 1 和 0 后可能是无限位数的，机器已经做出了截断处理。那么在机器中保存的 2.675 这个数字就比实际数字要小那么一点点。这一点点就导致了它离 2.67 要更近一点点，所以保留两位小数时就近似到了 2.67。

   所以除非对精确度没什么要求，否则尽量避开用 round()函数。使用 math 模块中的一些函数，比如 math.ceiling（天花板除法）。对浮点数精度要求如果很高的话，可以用 decimal 模块。

   https://www.runoob.com/w3cnote/python-round-func-note.html

3. math.floor()，地板除，**向小取整**。floor(x) 返回一个**小于或等于 x 的最大整数**。双斜杠也是地板除，但 floor() 返回值一定是 int 类型，双斜杠的结果，其数据类型取决于分子分母的符号。

4. math.ceil()，天花板除，**向大取整**。ceil(x) 返回一个**大于或等于 x 的的最小整数**。

5. 斜杠 /，除法，总是返回一个浮点数。

6. 双斜杠//，地板除 //，功能和 math.floor()类似，**只保留整数部分**，向小取整。但如果分子或分母有浮点型，结果就是浮点型。

4.3 and 和 or 优先级

首先要知道优先级顺序：not > and > or ，但是还要考虑短路逻辑，对 python 而言：

1. 表达式从左至右运算，**若 or 的左侧逻辑为 True ，则短路 or 后所有的表达式**（不管是 and 还是 or），直接输出 or 左侧表达式 。
2. 表达式从左至右运算，**若 and 的左侧逻辑值为 False ，则短路其后所有 and 表达式，直到有 or 出现，输出 and 左侧表达式到 or 的左侧**，参与接下来的逻辑运算。
3. 若 or 的左侧为 False ，或者 and 的左侧为 True 则不能使用短路逻辑。

对于单个 and 和 or ：

1. 在**不加括号时候，and 优先级大于 or** 。
2. **x or y 的值只可能是 x 或 y，x 为真就是 x，x 为假就是 y**，因为 x 为假的时候结果由 y 决定，y 是什么结果就是什么。x 为真的话结果就是假。
3. **x and y 的值只可能是 x 或 y，x 为真就是 y，x 为假就是 x**，因为 x 为真的时候结果由 y 决定，y 是什么结果就是什么。x 为假的话结果就是假。

对于，1 or 5 and 4：首先从左向右扫描，若没触发短路逻辑，则先算 5 and 4，但是开始的 1 直接触发短路逻辑，输出 1，后面的 and 看都不看，哪怕是个不合法的变量。

对于，(1 or 5) and 4：先算 1 or 5，1 为真，接下来是 and，and 左侧为 1 没有触发短路逻辑，再算 1 and 4，1 为真,结果就为 4 。

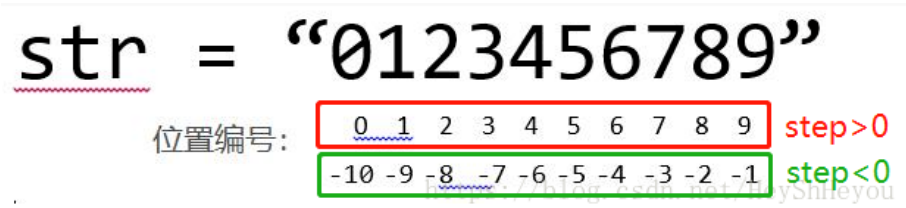
建议编码过程中，按照逻辑加上相应的括号，就不用考虑那么多问题。
https://www.cnblogs.com/irockcode/p/8662957.html

4.4 切片

在 Python 中，**有序结构的数据都可以使用切片操作**，比如 列表、元组、字符串 等，字典是无序的所以不可以。**切片操作产生一个和原对象同类型的副本，是一种浅拷贝，列表的切片还是列表，元组的切片仍是元组**。

range 的范围和顺序其实和切片类似，步长为负就是一个一个减下去。

切片是根据元素索引进行操作的，做逆序处理时候经常用到负数，所以**每个元素都有对应的正负编号，范围区间是左闭右开，而且可以越界**，如图：



当 step > 0 时，位置编号为从左边的 0 为起点开始，往右依次递增（0,1,2,3,4,5…），当 step < 0 时，位置编号为从右边的 -1 为起点开始，从右往左依次递减（…，-5，-4，-3，-2，-1）。

使用切片的几个要点，对于 str[start: stop: step]：

1. 如果 start 或者 stop 大于 len(str)，实际就取 len(str)。
2. 如果 start 省略或为 None，实际取 0。
3. 如果 stop 省略或为 None，实际取 len(str)。
4. 如果 start = stop，那么切片为空，因为切片时左闭右开区间。
5. 如果 step > 0，当 start > stop 时切片为空，区间越界。
6. 如果 step < 0，当 start < stop 时切片为空，区间越界。因为起点是 start，截取方向始终朝 step 走，当 step 是正数，那么朝着 step 走肯定是 stop > start，当 step 是负数那么肯定是 stop < start。比如 str[-6:-2:-1] 就返回空，但 str[-6:-2:1] = "4567"。

https://segmentfault.com/q/1010000011412371

4.5 元组解包

多重赋值、元组解包、迭代解包指的都是同一件事情，多重赋值实际上是创建一个元组，然后循环遍历该元组，并从循环中获取每个 items ，再分别赋值给变量，下面代码是等价的：

```
x, y = 10, 20
x, y = (10, 20)
(x, y) = 10, 20
(x, y) = (10, 20)
```

python 的**多重赋值语句中，各变量之间不会相互影响，也跟赋值顺序无关**，看代码：

```
if __name__ == '__main__':
    a = 3
    a, b = 1, a
    print(a, b)     # 1, 3
```

在 python 中输出的是 1，3，并非 1，1，也就是对 a 的新赋值并没有影响到对 b 的操作，这跟 C 或者其他高级语言不同，如果一个循环中存在多重赋值，那当此循环中，连续赋值的几个变量之间不会相互产生影响，无所谓谁先谁后，比如斐波拉契的迭代法：

```
def fibonacci(n):
    a = b = 1
    if n == 1 or n == 2: return 1
    for _ in range(n-2):
        a, b = b, a + b
        print(a, b)
    return b
```

4.6 is 和 ==

1. is 是身份运算符，也就是比较内存地址（id）是否相同。x is y 就是 id(x) == id(y) ，如果引用的是同一个对象则返回 True，否则返回 False 。id() 用于获取对象内存地址。

2. == 是比较运算符，用来判断两个对象的 value（值）是否相等。

3. 在 IDE 中执行，给 a 和 b 赋相同的值，只有数值型、字符串、元组这种不可变对象的情况下，a is b 才为 True，当 a 和 b 是 list，dict 或 set 型时，a is b 为 False。只要值相等 a == b 都是 True。如果是 a = x 然后 b = a 的形式，那无论可变还是不可变对象，a is b 和 a == b 都是 True。列表，集合等可变对象有浅拷贝和深拷贝的操作，对于深浅拷贝 a is b 都是 False，a == b 都是 True。

4. 对于数类创建的对象，python 默认去比较两个对象的地址，a = test()，b = test()，那么 a is b 和 a == b 都是 False。如果想要类中的方法值相同，==就 True 的话，需要重写 python 中的__eq__方法，通常把这个过程叫做运算符重载，

```
class test(object):
    def __init__(self):
        self.name = "123"
        self.age = 10
    def __eq__(self, other):
        return self.name == other.name and \
               self.age == other.age
```

这样的话 a == b 返回 True，a is b 返回 False。

4.7 If not 和 None

1. 在 python 中 None, False, 空字符串"", 0, 空列表[], 空字典{}, 空元组()都相当于 False ，**空格" "或嵌套空列表如[[]]不是 False**，因此 `if not x:` 对于上面的情况都是 True，所以使用 if not x 这种写法的前提是：必须清楚 x 等于 None, False, 空字符串"", 0, 空列表[], 空字典{}, 空元组()时对你的判断没有影响才行。

2. 在使用列表的时候，如果你想区分 x==[]和 x==None 两种情况的话，此时 `if not x:`将会出现问题。

3. 判断非 None 最好用 `if x is not None`，清晰易懂，这也是谷歌推荐的写法。

   https://www.cnblogs.com/clarenceyang/p/9681653.html

4.8 负数的二进制

Python3 中 bin 一个负数（十进制表示），输出的是它的原码的二进制表示加上个负号。比如：bin(-3) 是 -0b11，hex(-3) 是 -0x11。

但是在 c/c++/java 里面负数都是以补码的形式进行存储的，《计算机原理》显示，计算机内部采用 2 的补码（Two's Complement）表示负数。

这就出现了在 Python 里面需要将负数和 0xffffffff 进行与操作，来去掉负数前面的负号，可以理解为超过 32 位的东西就不进行考虑了，这进行与操作的具体步骤是：如果是正数，直接与；如果是负数，先去掉最前面的负号，再取反，再加 1，再进行与操作。从而得到负数的补码。

因此对于输出的 a 我们也要进行截断，但是不能简单粗暴地直接 &0xffffffff，因为这样做的话-1 加 1 是对了，结果是正数的也没问题，但是如果本来结果是负数的，这样就又出奇怪结果了。

【Python 对于负数的存储方式和 c++/c/java 不一样（二进制中 1 的个数）】

4.9 HashMap

集合和字典都是由 HashMap 实现的，那 HashMap 的时间复杂度为什么是 O(1)？我们知道直接读取一个列表的 index 复杂度为 O(1)，HashMap 其实也利用了这个特性，但是稍微复杂一点。

假如现在有一个哈希表，有 5 个格子，编号分别为 0，1，2，3，4，然后我们想把 key == 11 放进去，那么放到哪个格子呢？有一种方法是，11 除以 5

的余数是 1，那么就放到编号为 1 的格子里去，那么这个时候的哈希映射（映射函数）就是 h(x) = x % 5。

假设我们要查找 key == 11，我们会怎么做呢？遍历一遍哈希表？那就是 O(n)了，完全没有体现出哈希表的优越性。我们根据哈希映射 h(x) = x % 5，把 x == 11 代进去，一下子就知道了在 1 号格子里，相当于直接读取列表的一个 index，自然时间复杂度也就是 O(1)。但如果多个 key 映射到同一个格子怎么办？比如来了个 key == 16，按照哈希映射应该放到 1 号格子，但是这里已经有 key == 11 了，这就产生了**哈希冲突**，如果所有 key 都映射到一个各自，就达到了最坏的时间复杂度 O(n)，解决冲突的方式有：1. 开放定址法(线性探测，二次探测，伪随机探测)；2. 链地址法；3. 再散列（双重散列，多重散列）；4. 建立一个公共溢出区等。

在字典或集合中的“查找”可以分为四步：

1. 判断 key，根据 key 算出索引。
2. 根据索引获得索引位置所对应的键值对链表（产生冲突就会由多个节点）。
3. 遍历键值对链表，根据 key 找到对应的 Entry 键值对。
4. 拿到 value。

产生冲突的时候需要对链表进行查找，时间复杂度为 O(n)，但产生冲突的几率极小，所以 HashMap 的平均时间复杂度是 O(1)。

集合的话在交互模式里调用 pop()是弹出第一个元素，在 shell 中调用是随机弹出一个元素。**实际上 HashMap 也是按哈希值大小排序的**，如果一组递增的数，经过散列后 hash 值也是递增的，那你会看到 pop 的时候是有序的，但这只是特殊情况，字典和集合依然是无序的。

【HashMap, HashTable，HashSet,TreeMap 的时间复杂度 注意数组链表 增删改查的时间复杂度都不相同(阿里)】

【面试中关于 HashMap 的时间复杂度 O(1)的思考】

4.10 *args and **kwargs

第五章 算法基础

5.1 提示

1. 每章我都尽量附上了参考文章，一来方便大家查看原文深入了解，二来尊重原作者的付出。但是内容越整理越多，疏漏之处还请理解。

2. **千万不要看不起暴力解**，很多优化都是建立在暴力解之上，比如动态规划问题最困难的就是写出状态转移方程，即暴力解。优化方法无非是用备忘录或者 DP table。

3. 代码模板默认如下，**有时候会省略类，所以看到只有一个函数却有 self 参数不要奇怪**，实际中只有一个函数还套用 class 有点强行 OOP 的感觉，不过模板讲究通用性，而且刷题平台给的基本也是 class，我们重点看算法就行。

```python
class Solution:
    def my_function (self, nums):
        ...
        ...
        ...
        return nums


if __name__ == '__main__':
    S = Solution()
    test = [1, 2, 2]
    res = S.my_function(test)
    print(res)
```

5.2 斐波拉契、爬台阶

斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：1、1、2、3、5、8、13、21、34、……在数学上，斐波那契数列以如下被以递推的方法定义：F(1)=1，F(2)=1，F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）。有时候也从 0 开始，用 F(0)=0，F(1)=1，F(n)=F(n-1)+F(n-2)（n>=2，n∈N*）。

暴力递归，时间复杂度 O(2^n)，空间复杂度为树的高度 h，O(n)级别。时间复杂度为二叉树的节点个数：(2^h)-1=O(2^N) 。二叉树节点总数为指数级别，所以子问题个数为 O(2^n)。子问题只有 f(n - 1) + f(n - 2) 一个加法操作，时间为 O(1)。所以，这个算法的时间复杂度为 O(2^n)，指数级别：

```
# 暴力递归，时间复杂度为 O(2^n) ，空间复杂度为树的高度 O(n)。
def fibonacci(n):
    # global cnt
    # cnt += 1  # 统计函数执行次数

    if n == 1 or n == 2:
        return 1
    return self.fibonacci(n-1) + self.fibonacci(n-2)
```

备忘录递归，子问题只是 f(1)，f(2)，f(3) ... ，不必重复计算所以子问题个数为 O(n)。解决一个子问题的时间，没有什么循环也是 O(1)。总时间复杂度 O(n)。

```
# 记忆化递归，时间复杂度为 O(n) ，空间复杂度为 O(n)。
def fibonacci2(n):
    # global memo, cnt
    # cnt += 1  # 统计函数执行次数
    # print(n)

    if memo[n] != -1:
        return memo[n]

    if n == 1 or n == 2:
        memo[n] = 1
        return memo[n]

    memo[n] = self.fibonacci2(n-1) + self.fibonacci2(n-2)
    return memo[n]
```

迭代，时间复杂度 O(n) ，空间复杂度为 O(1)。

```
# 迭代
def fibonacci3(n):
    if n == 1 or n == 2:
        return 1
    a = b = 1
    for _ in range(n-2):
        a, b = b, a + b
    return b
```

矩阵快速幂的方法可以达到 O(log n)的复杂度，希望不要考。。。

5.3 汉诺塔（hannuota）

有三根杆子 A，B，C。A 杆上有 N 个(N>1)穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至 C 杆：

1. **每次只能移动一个圆盘**；
2. **大盘不能叠在小盘上面**。

提示：可将圆盘临时置于 B 杆，也可将从 A 杆移出的圆盘重新移回 A 杆。

设盘子个数为 n 时需要 T（n)步，那么把 A 柱子的 n-1 个盘子移到 B 柱子需要 T（n-1)步，把 A 柱子最后一个盘子移到 C 柱子需要一步，把 B 柱子上 n-1 个盘子移到 C 柱子需要 T（n-1)步。得到递推公式 T(n)=2T(n-1)+1+T(1)。所以汉诺塔问题的时间复杂度为 O(2^n)：

```
def move(A, C):
    print("move:" + A + "--->" + C)

def hannuota(n, A, B, C):
    if (n == 1):
        move(A, C)
    else:
        hannuota(n - 1, A, C, B)
        move(A, C)
        hannuota(n - 1, B, A, C)
if __name__ == '__main__':
    hannuota(3, 'a', 'b', 'c')
```

5.4 排序

Python 中的 sorted 排序用的是 TimSort 算法，TimSort 算法**本质上是对归并排序的优化，最好时间复杂度 O(n)，平均和最坏是 O(nlog n)**，Timsort 的核心过程如下：

TimSort 算法为了减少对升序部分的回溯和对降序部分的性能倒退，**将输入按其升序和降序特点进行了分区。排序的输入的单位不是一个个单独的数字，而是一个个的块-分区**。其中每一个分区叫一个 run。针对这些 run 序列，每次拿一个 run 出来按规则进行合并。每次合并会将两个 run 合并成一个 run。合并的结果保存到栈中。合并直到消耗掉所有的 run，将栈上剩余的 run 合并到只剩一个 run 为止。这时这个仅剩的 run 便是排好序的结果。

综上述过程，Timsort 算法的过程包括：

（0）如何数组长度小于某个值，直接用二分插入排序算法；

（1）找到各个 run，并入栈；（2）按规则合并 run

timesrot 是稳定的排序算法，最坏时间复杂度是 O(nlogn)。在最坏情况下，Timsort 算法需要的临时空间是 n/2，在最好情况下，它只需要一个很小的临时存储空间

https://blog.csdn.net/yangzhongblog/article/details/8184707

5.4.1 冒泡排序（Bubble Sort）

类似于水中冒泡，越小的元素会经由交换慢慢“浮”到数列的顶端。

```
# 优化版冒泡，平均和最坏 O(n^2)，最好 O(n)，空间 O(1)
def bubble_sort(self, nums):
    n = len(nums)
    for i in range(n - 1):
        flag = False    # 优化的关键，用于记录是否发生交换
        for j in range(n - i - 1):    # 注意这个范围
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
                flag = True
        if not flag:
            break
    return nums
```

比较相邻的元素，如果第一个比第二个大，就交换他们两个。

1. 外层循环：n 个数字最多需要比较 n - 1 次，所以外层循环是 n - 1 次。**每次循环结束最后一个数字都处于最终的位置**。

2. 内层循环：每次外层循环确定一个位置，所以内层循环会越来越小，比如第一轮循环确定了最后一位是最大值，下一轮循环遍历到倒数第二位就行。
3. 优化：如果数组本来就基本有序，那就不必循环 n - 1 次，所以定义一个 flag，每次交换都会改变 flag，如果某次循环结束后 flag 没变化，就说明没发生交换，直接结束。
4. 稳定排序：相邻的两个数一样大时候不必交换，所以冒泡是**稳定的排序**。
5. 适用于：数据量不大的时候。
6. 时间复杂度：**平均和最坏是 O(n^2)，基本有序的情况下，优化版可以达到 O(n)**。
7. 空间复杂度：O(1)，只需要一个常量级的临时空间存储变量 flag，是原地排序算法。

5.4.2 插入排序（Insertion Sort）

将一个值插入到已经排好序的有序表中。

```
# 标准插入排序，平均和最坏 O(n^2)，最好 O(n)，空间 O(1)
def insert_sort(self, nums):
    for i in range(1, len(nums)):
        j, tmp = i - 1, nums[i]
        while j >= 0 and tmp < nums[j]:
            nums[j+1] = nums[j]
            j -= 1
        nums[j+1] = tmp
    return nums
```

优化：

```
# 二分插入排序，平均和最坏 O(n^2)，最好 O(n)，空间 O(1)
def insert_sort_binary(self, nums):
    res = [nums[0]]
    for i in range(1, len(nums)):
        low, high = 0, len(res)
        while low < high:
            mid = low + (high-low) // 2
            if nums[i] >= res[mid]:
                low = mid + 1
            else:
                high = mid
        res.insert(low, nums[i])
    return res
```

插入排序把数组看成有序和无序两部分，从无序部分中取出 1 个数，在有序部分**从后向前**扫描，找到相应位置将其插入。在从后向前扫描过程中，需把插入位置后面的元素依次后移，**为即将插入的元素腾出位置**。

1. 外层循环：遍历无序部分的所有数字，初始只有第 1 个数属于有序部分。
2. 内层循环：寻找插入点，插入点之后的整体后移一位以腾出插入位置。
3. 优化：折半（二分）插入，**折半查找只是减少了比较次数**，但是元素的移动次数不变，所以时间复杂度和直接插入是一样的，为 O(n^2)。
4. 稳定排序：相等的数字不做处理，所以插入是**稳定的排序**。
5. 适用于：数据量不大且基本有序的时候。
6. 时间复杂度：**平均和最坏是 O(n^2)，基本有序的情况下可以达到 O(n)**。
7. 空间复杂度：O(1)，原地排序算法。

5.4.3 选择排序（Selection sort）

每次都找出一个最小的数字和当前数字交换，记录的是数字下标。

1. 外层循环：i 为遍历从 0 到 n-1 的数字。
2. 内层循环：找出 i + 1 到 n 中最小的数和 i 交换。
3. 优化：可以同时找出最大和最小值，理论上能减少一半的循环。
4. **不稳定**排序：比如 5 8 5 2 9，第一个 5 和 2 交换后，就改变了两个 5 原来的顺序。
5. 适用于：数据量不大且基本无序的时候。
6. 时间复杂度：平均和最坏是 O(n^2)，基本有序的情况下通过优化可以达到 O(n)。
7. 空间复杂度：O(1)，只需要一个常量级的临时空间存储变量 flag，是原地排序算法。

```
# 标准选择排序，  平均最坏和最好都是 O(n^2)，  空间 O(1)
def selection_sort(self, nums):
    for i in range(len(nums) - 1):
        min = i
        for j in range(i+1, len(nums)):
            if nums[j] < nums[min]:
                min = j
        nums[i], nums[min] = nums[min], nums[i]
    return nums
```

5.4.4 三种排序的比较

最好：顺序或数字全部重复的时候：冒泡 > 插入 > 选择。

最坏：逆序的时候：选择 > 插入 > 冒泡。

一般：非常小的数据情况下：插入 > 选择 > 冒泡。

一般：超过 50 条数据：选择 > 插入 > 冒泡。

我用 Python 测试是这样的，百千个数据以上基本选择排序稳定快于插入排序，应该是插入过程中大量的移位（赋值）造成，而选择排序至多只需要 n-1 次交换。

来看看最好的情况，顺序时候，三种排序都达到了最好的复杂度：

1. 冒泡排序的比较次数是 n(n-1)/2，但经过优化后（通过判断 flag 状态提前结束）比较 n-1 次即可。
2. 选择排序的比较次数是固定 n(n-1)/2，不受顺序逆序的影响，甚至会把顺序数组中的每个数字和自己交换一次。
3. 插入排序的比较次数是 2(n-1)，因为还需要判断是否越界。
4. 冒泡和插入都只需要比较就行了，但选择还要进行交换，所以在效率上冒泡 > 插入 > 选择。

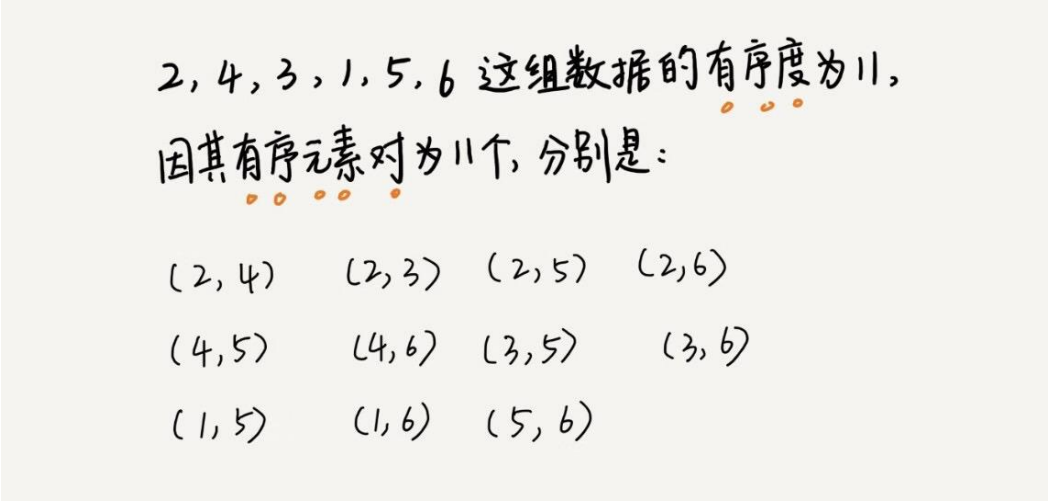
来看看最坏的情况，逆序时候，三种排序都达到了最坏的复杂度：

1. 冒泡排序的比较次数是 n(n-1)/2，相当于 1+2+3+…+n-1，一个等差数列的和。
2. 选择排序的比较次数是固定 n(n-1)/2，并且不受顺序逆序的影响。
3. 插入排序的比较次数是(n+2)(n-1)/2，插入比较的次数主要是两部分，一部分是根据比较判断插入点，单独看这部分其实也是 n(n-1)/2，第二部分是看是否越界，每次比较都要先看是否越界，所以一开始就要进行两次比较，相当于 2+3+4+…+n，也是等差数列的和。
4. 但除了比较次数还要看交换和移动，冒泡要进行 n(n-1)/2 次的比较和交换。插入排序要进行(n+2)(n-1)/2 次的比较和移动（赋值挪位），选择排序固定进行 n(n-1)/2 次的比较，但至多只需要 n-1 次的交换。因为赋值的效率是高于交换的，所以对于逆序数组，效率上选择 > 插入 > 冒泡。

插入排序一般优于冒泡排序，冒泡排序包含比较和交换两个操作，不管算法怎么改进，交换次数总是固定为逆序度。插入排序也包含比较和移动两个操作，移动的次数也是固定为逆序度（逆序度概念参考下面），虽然都是逆序度，但冒泡的交换比插入的移动（用赋值来实现）要耗时，交换一次需要赋值三次（tmp=a; a=b; b=tmp）。Python 中的交换常用 ”a, b = b, a” 来实现，看似没有使用临时变量，但其实在 2、3 个值分配的时候是直接运用栈，在 3 个以上值分配的时候是用了拆包的原理【详细看这里】。

对于完全顺序的数组，优化后的冒泡排序最快，对于基本有序或者几十个数字的数组，插入排序最快，对于随机百千级别的数组，选择排序最快。

顺便了解下“有序度”和“逆序度”这两个概念：



对于一个完全有序的数组，有序度就是 n*(n-1)/2，比如 [1，2，3，4，5，6]，也叫满有序度，逆序度的定义正好跟有序度相反。逆序度 = 满有序度 - 有序度。我们排序的过程就是一种增加有序度，减少逆序度的过程，最后达到满有序度，就说明排序完成了。

冒泡排序包含两个操作原子，比较和交换，不管算法怎么改进，交换次数总是固定的，即为逆序度。

插入排序包含两个操作原子，比较和移动。移动的次数也是固定的，即为逆序度。但是比较时候多了一个是否越界的判断。

【直接插入排序最坏情况（逆序）时的总比较次数和总移动次数是多少】

【11 | 排序（上）：为什么插入排序比冒泡排序更受欢迎？】

【为什么说平均情况下，插入排序比选择排序快?】

【为什么插入排序比冒泡排序更受欢迎】

5.4.5 快速排序（Quick Sort）

5.4.5.1 最简单版本

容易理解，但是额外空间消耗较大：

```
def quick_sort2(self, nums):
    # random.choice 遇到空列表会报错，不能省略
    if len(nums) <= 1: return nums
    left, right, p = [], [], []
    pivot = random.choice(nums)
    for n in nums:
        if n == pivot: p.append(n)
        elif n < pivot: left.append(n)
        else: right.append(n)
    return self.quick_sort2(left) + p + self.quick_sort2(right)
```

5.4.5.2 Hoare 版本

Hoare 版本就是最常见的首尾指针**双向扫描，最左边元素是 pivot**：

1. 尾指针 j 从右向左找到一个比 pivot 小的，首指针 i 从左向右找到一个比 pivot 大的，交换这两个数，交换是用相互赋值实现的。
2. 继续上面的操作直到 i 和 j 相遇，然后交换 pivot 和 i。
3. 第一轮扫描结束后，小于 pivot 的在其左边，大于等于 pivot 的在其右边。

```
# 平均和最好是 O(nlogn)，最坏是 O(n^2)，空间 O(nlogn)
# right=len(nums)-1 语法错误，找不到 nums，这里是提示一下传入参数的格式。
def quick_sort(self, nums, left=0, right=len(nums)-1):
    if left < right:
        pi = self.partition_hoare(nums, left, right)
        print(nums, left, right)
        self.quick_sort(nums, left, pi-1)
        self.quick_sort(nums, pi+1, right)
    return nums

# Hoare 版本，首尾指针双向扫描，最左边元素是 pivot
def partition_hoare(self, nums, start, end):
    pivot = nums[start]
    while start < end:
        while start < end and nums[end] >= pivot:
            end -= 1
        nums[start] = nums[end]
        while start < end and nums[start] < pivot:
            start += 1
        nums[end] = nums[start]
    nums[start] = pivot
    return start
```

一般 Hoare 是优于 Lomuto 版本的，因为 Hoare 不需要频繁交换。

5.4.5.3 N.Lomuto 版本

N.Lomuto 版本是双指针**单向扫描，最右边元素是 pivot**：

1. 初始 low 在 -1，high 在 0 位置，从左向右扫描，**low 指向已经排序的、小于 pivot 的元素的最右位置**（那么 low 的右侧下一个位置就是大于枢纽的了，low 走过的就属于已经排序的区间），high 指向待排序的数组（high 扫描待排区间，一定位于 low 之后），当 **high 找到了一个小于等于 pivot 的位置**，则将 low 加 1（此时 low 指向了大于枢纽的第一个位置），然后将 low 和 high 所指的位置元素交换；交换后 low 所指的元素小于 pivot（此

时，low 的右侧下一个位置就是大于枢纽的了），high 所指的值大于 pivot。

2. high 继续向右扫描。扫描到最后一位，也就是 pivot，把 pivot 和 low+1 交换，此时 pivot 左边都是比它小的，右边都是比它大的。

```
# N.Lomuto 版本，双指针单向扫描，最右边元素是 pivot
def partition_lomuto(self, nums, start, end):
    i, pivot = start - 1, nums[end]
    for j in range(start, end):
        if nums[j] <= pivot:
            i += 1
            nums[i], nums[j] = nums[j], nums[i]
    nums[i+1], nums[end] = nums[end], nums[i+1]
    return i + 1
```

Lomuto 的优点是实现起来简单，缺点是太多的交换。

5.4.5.4 三向切分法

三向切分仅**适用于重复值较多**的时候，在没有重复内容的时候，三向切分快排反而要多做很多交换。

思想是：最终 lt 左边的元素小于 pivot，gt 右边的元素大于 pivot，lt 和 gt 这个闭区间的元素等于 pivot。所以 i 从 lt 的下一位开始遍历，遇到小于 pivot 的就和 lt 交换，否则如果大于 gt 就和 gt 交换，建议手写一下：

```
# 三向切分法，适用于大量重复元素，3-Way QuickSort (荷兰国旗问题)
def quick_sort_3way(self, nums, start, end):
    if start >= end: return
    lt, lg = start, end
    i = start + 1
    while i <= lg:
        if nums[i] < nums[lt]:
            nums[lt], nums[i] = nums[i], nums[lt]
            i += 1
            lt += 1
        elif nums[i] > nums[lt]:
            nums[lg], nums[i] = nums[i], nums[lg]
            lg -= 1
        else:
            i += 1
    self.quick_sort_3way(nums, start, lt-1)
    self.quick_sort_3way(nums, lg+1, end)
    return nums
```

【快速排序（三向切分）】

5.4.5.5 优化

1. pivot 随机化：

在待排序列是部分有序时，固定选取 pivot 使快排效率低下，要缓解这种情况，就引入了随机选取 pivot，如果是 Hoare 算法就和第一个数交换，依然以第一个数为 pivot，如果是 N.Lomuto 算法就和最后一个数交换，依然以最后一个数为 pivot。

```
p_index = random.randint(start, end)
pivot = nums[p_index]
nums[end], nums[p_index] = nums[p_index], nums[end]
```

2. pivot 三数中值：

为了减少出现不好分割的几率，引入了三数取中当作 pivot。选取首中尾三个数，通过比较交换使第一个数小于中间数小于尾数，然后把中间数和尾数的前一个数交换。

```
mid = (start + end) // 2
if nums[mid] < nums[start]:
    nums[start], nums[mid] = nums[mid], nums[start]
if nums[end] < nums[start]:
    nums[start], nums[end] = nums[end], nums[start]
if nums[end] < nums[mid]:
    nums[end], nums[mid] = nums[mid], nums[end]
nums[end - 1], nums[mid] = nums[mid], nums[end - 1]
```

比如 4 5 7 8 1 2 3 6 ，三个数经过比较，交换变成 4 5 7 6 1 2 3 8
把中间数和倒数第二个数交换 4 5 7 3 1 2 6 8
以 6 为 pivot，现在的 4 和 8 已经分别处于 pivot 的两侧，
https://yq.aliyun.com/articles/590612

只要 pivot 在右侧就得先从左扫描。标准的 Hoare 算法 pivot 是在左侧第一位，所以先从右扫描。

为什么 pivot 在左侧就必须先从右开始扫描呢？因为从左向右找到一个大于 pivot 的数没问题，但是从右向左扫描时候 i 和 j 相遇会停止，此时 j 可能没有找到比 pivot 小的数，这时候 pivot 和 j 交换，可能换了个很大的数。考虑 2 1 4 9，如果 i 先从左扫描，找到比 2 大的数是 4，再 j 从右扫描找比 2 小的数，还没找到 1， i 和 j 就在 4 这里相遇了，这时候 2 和 4 交换结果是 4 2 1 9，明显是不对的，先从右边开始就算找不到比 2 小的数，顶多也是 2 和 2

交换，不会出错。当你从右边开始的时候，就一定是可以保证和 pivot 交换那个数一定是小于等于 pivot 的。

复杂度：

若第一层的话就是 n/2，n/2，若是第二层就是 n/4,n/4,n/4,n/4 这四部分，即 n 个元素理解上是一共有几层 2^x=n，x=logn，然后每层都是 n 的复杂度，那么平均就是 O(nlogn)的时间复杂度。但如果你是 ascending 的顺序，那么递归只存在右半部分了，左半部分都被淘汰了。(n-1)*(n-2)*....*1，这个复杂度就是 O(n^2)，这种情况还不如用插入排序。

5.4.6 几种排序的复杂度

冒泡排序是稳定的排序。平均和最坏 O(n^2)，最好 O(n)，空间 O(1)。

插入排序是稳定的排序。平均和最坏 O(n^2)，最好 O(n)，空间 O(1)。

选择排序是不稳定的排序。平均最好和最坏都是 O(n^2)，最好 O(n)，空间 O(1)。特点是耗时几乎不怎么受数据状态的影响，不管什么数据都要进行 O(n^2)次的循环和比较。

快速排序是不稳定的排序。平均和最好 O(nlogn)，最坏 O(n^2)，空间 O(nlogn)，绝大多数场合都用快速排序而不是其它的 O(nlogn)排序。

堆排序是不稳定的排序。平均最好和最坏都是 O(nlogn)，空间 O(1)。堆排序比较和交换的次数比快速排序多，所以平均而言比快速排序慢，也就是常数因子比快速排序大。

但有时候你要的不是“排序”，而是另外一些与排序相关的东西，比如最大/小的元素，top K 之类，这时候堆排序的优势就出来了。用堆排序可以在 N 个元素中找到 top K，时间复杂度是 O(N log K)，空间复杂的是 O(K)。

另外一个适合用 heap 的场合是优先队列，需要在一组不停更新的数据中不停地找最大/小元素，快速排序也不合适。

5.5 链表

链表是一种线性表，但并**不是按顺序存储**，而是在每一个节点里保存下一个节点的指针，来看一下插入，删除，查找（我认为用读取更合适，比如读取a[2]，因为查找容易产生歧义，比如用二分法在列表中查找一个元素需要0(logn)，但这里我们指的其实是读取一个特定编号的元素，而不是在列表或单链表中搜索一个元素）的时间复杂度。

1. **单链表的插入删除可以达到0(1)**的复杂度，因为不需要按顺序存储，但**查找指定元素却需要0(n)**，因为不是按顺序存储的。
2. **列表的插入删除可以达到0(n)**的复杂度，因为插入后需要移动元素，但**查找特定编号的元素只需要0(1)**。

实际中插入/删除和查找是一体的，需要查找完再进行插入或删除，所以整体来看单链表和顺序表的查找+插入/删除都是0(n)，区别是单链表中的查找是查询（读取）花费了0(n)，顺序表是写入（移动元素）花费了0(n)。

5.5.1 反转单链表

先看一下整体模板：

```
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def reverseList(self, head):
        .....
    def print_node(self, head):
        while head:
            print(head.val)
            head = head.next

if __name__ == '__main__':
    S = Solution()
    n1, n2 = ListNode(1), ListNode(2)
    n3, n4 = ListNode(3), ListNode(4)
    n5 = ListNode(5)
    n1.next, n2.next = n2, n3
    n3.next, n4.next = n4, n5
    res = S.reverseList(n1)
    S.print_node(res)
```

迭代：

```
def reverseList(self, head):
    prev = None
    while head:
        cur = head
        head = head.next
        cur.next = prev
        prev = cur
    return cur
```

递归：

```
def reverseList2(self, head):
    if not head or not head.next:
        return head
    cur = self.reverseList2(head.next)
    head.next.next = head
    head.next = None
    return cur
```

5.5.2 反转单链表指定区间

```
def reverseBetween(self, head, m, n):
    dummy = prev = ListNode(0)
    dummy.next = head
    cur = head
    for _ in range(m-1):
        cur = cur.next      # 定位到第一个要改变的点
        prev = prev.next    # 前置节点也需要记录

    for _ in range(n - m):
        tmp = cur.next
        cur.next = tmp.next    # 改变 cur.next 节点
        tmp.next = prev.next   # 改变 cur.next.next
        prev.next = tmp         # 改变 cur.prev.next
    return dummy.next
```

5.5.3 环形链表入口（LC142）

如果可以使用额外空间的话很简单,用哈希表，但这种题考点一般是快慢指针 Floy 算法，值相同的两个节点并不一定是同一个节点，但我们直接比较两个

节点，只要相同一定是同一节点，两个一样的节点具有相同的内存地址，所以比较两个节点最好用 is，别用 == 。时间复杂度：O(n)：

```
def detectCycle2(self, head):
    visited = set()
    while head:
        if head in visited:
            return True
        else:
            visited.add(head)
            head = head.next
    return False
```

找环形链表的入口分两个阶段来做：

1. 快慢指针相遇表示有环，快指针每次走 2 步，慢指针每次走 1 步，快指针总是走了慢指针两倍的路，所以有环的话它们会相遇。
2. 找到相遇点后，令 ptr1 指向链表的头， ptr2 指向相遇点不变，然后一步一步向后移动，它们再次相遇的点就是环的入口。

代码如下：

```
def detectCycle(self, head):
    fast = slow = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            fast = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow
```

快指针每次走 2 步，慢指针每次走一步，相对的可以理解为慢指针不动，快指针一步一步往前走，只要有环迟早能相遇。也可以用数学归纳法思考。首先，由于链表是个环，所以相遇的过程可以看作是快指针从后边追赶慢指针的过程。那么：

1. 快指针与慢指针之间差一步。此时继续往后走，慢指针前进一步，快指针前进两步，两者相遇。
2. 快指针与慢指针之间差两步。此时继续往后走，慢指针前进一步，快指针前进两步，两者之间相差一步，转化为第一种情况。
3. 快指针与慢指针之间差 N 步。此时继续往后走，慢指针前进一步，快指针前进两步，两者之间相差(N+1-2)-> N-1 步，转化为前一种情况。所以快指针必然与慢指针相遇。

【为什么用快慢指针找链表的环，快指针和慢指针一定会相遇？- 冯昱尧的回答】

注意点：

1. 首先快慢指针的“相遇”是指，**某一次行动完，快慢指针同时指停在一个点上，路过不算相遇**。
2. 其次**快慢指针初始化是从链表头节点开始的**，这是一个很重要的前提，我们经常会写几个例子看到底是否能相遇，追击问题就是不断的在环内循环，所以只看环内的情况，比如这个环有 0 到 5 共 6 个节点，假如当前快慢指针距离一步，步长分别为 fast=3，slow=1，好像永不相遇？但其实这个假设本身就是错误的，快慢指针同时从一个点出发的话，在六个节点的环内根本不存在快慢指针相差一步的情况，当然如果换成三个或五个节点的话是存在的，快慢指针可能的距离差还与环的节点数有关。

关于快慢指针判断是否有环的问题，有三点非常重要：

1. **有环链表中，从链表头节点同时出发的快慢指针一定会相遇吗（注意这里的快慢就是一快一慢，不是特指 2 步和 1 步）？**
   答： 一定会，只要同一起点出发。

2. **如上所说，如果快慢指针一定能相遇，那快慢指针为什么要取 2 步和 1 步，快指针可以取 3，4，5 步吗？**
   答：可以，1 步 2 步只是为了高效。

3. **为什么快指针相遇后，重置一个指针为链表头节点，然后两个指针一步一步走下去，再次相遇就是环的入口点？**
   答：设链表头节点到环入口点的距离为 s，环长度为 t，相遇点为 j，那从 j 再走 t - j 就会到达环入口点，**而 s = t - j，从链表头节点走 s 到达环入口点，从相遇点走 t - j 也到达环入口点，相遇。**

下面证明一下：

证明 1：为什么有环链表中，同时从头节点出发的快慢指针一定会相遇？

我们先假设快慢指针相遇在一点，再证明这个点是存在的。设链表头节点到环入口点的距离是 s，环入口点到快慢指针相遇点的距离是 j，环的长度是 t，快指针是慢指针的 k 倍，为了方便我们把环入口点标记为 0，j 既是相遇点，也是慢指针环内走过的长度。

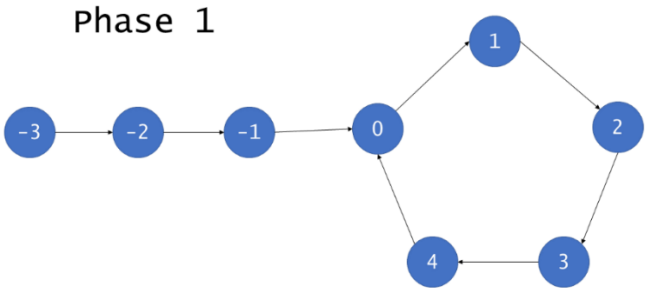
快慢指针在 j 点相遇可以得出公式 $j \bmod t = (ks + kj - s) \bmod t$，这个公式左右两边分别代表**快慢指针在环内走的长度**，慢指针总共走了 $s + j$，但只有 j 在环内，快指针总共走了 $k(s + j)$，其中 $k(s + j) - s$ 在环内，如果能证明这个公式有解，那么快慢指针就能相遇：

$j \bmod t = (ks + kj - s) \bmod t$

➔ $(j + s) \bmod t = k(s + j) \bmod t$

➔ $[(k - 1)(s + j)] \bmod t = 0$

只要快慢指针不相同，k - 1 就是正整数，可以看到存在 j = t - s 使上述公式变形为 t(k - 1) mod t = 0 ，肯定成立，实际上满足 j = (t - s) mod t 的都成立，因为是一个环。看如下例子，s = 3，t = 5，j = (t - s) mod t ➔ j = (5 - 3) mod 5 = 2。



	环内起点	下一步	下一步	…	…	…	…	…
slow = 1	0	1	2	3	4	0	1	2
fast = 2	3	0	2					
fast = 3	1	4	2					
fast = 4	4	3	2	1	0	4	3	2
fast = 5	2	2	2					

【为什么用快慢指针找链表的环，快指针和慢指针一定会相遇？ - 小周周的爸爸的回答】

证明 2：为什么要取慢指针步长为 1，快指针步长为 2，快指针取 3、4、5 等会怎么样？

s：链表头节点到环入口点的距离，
j：环入口点到快慢指针相遇点的距离，
t：环的长度，
k：快指针是慢指针的 k 倍，
m：快慢指针相遇时，快指针走了多少圈，

相遇时，慢指针走过的距离是 s + j，快指针走过的距离是 s + j + m * t，因为慢指针到环入口时，快指针已经在环中走了一段距离，它们相遇的的话快指针一定绕了环至少一圈，快指针是慢指针的 k 倍，可以得到：

k * (s + j) = s + j + m * t
➔ s + j = [m / (k - 1)] t

上面等式要成立的话，慢指针走过的长度 s + j 是一个整数，环的长度是一个整数，所以 m / (k - 1) 也是一个整数，那么慢指针走过的长度就是环长度的整数倍，慢指针走的长度越小效率越高，所以当 m / (k - 1) = 1 是最好的，也就是慢指针走过的长度就是环的长度，因此，m = k - 1 ➔ k = m +1，由于 m 是快指针绕环的次数，快指针想要和慢指针相遇至少要多跑一圈，因此 m >= 1，当 m = 1 时最高效， 因此 k = 2。如果我们取 k > 2 的值，则两个指

针将必须行进更多的距离。

综上，**慢指针步长为 1，快指针步长为 2 只是为了高效**，此时它们可以最快的相遇，此时慢指针走过的长度为 s + j，等于环的长度 t，快指针绕环一圈。

【Why increase pointer by two while finding loop in linked list, why not 3,4,5?】

证明 3：为什么快慢指针相遇后，令一个指针指向链表起点，然后两个指针一步一步走下去，再次相遇就是环入口点？

在证明 2 中我们得出，慢指针步长为 1，快指针步长为 2 时，它们可以最快的相遇，此时慢指针走过的长度为 s + j，等于环的长度 t，即 s + j = t ➔ s = t - j，在链表头节点的指针走过 s 到达环入口点，没问题，在 j 点的指针接着走 t - t 也到了环入口点，没问题，所以它们相遇了。

公式证明如下，利用证明 2 中的变量：

s：链表头节点到环入口点的距离，

j：环入口点到快慢指针相遇点的距离，

t：环的长度，

m：快慢指针相遇时，快指针走了多少圈，

这里就用快指针是慢指针的 2 倍，慢指针走过的长度乘以 2 等于快指针走过的距离：

2 * (s + j) = s + j + mt

➔ s + j = m t

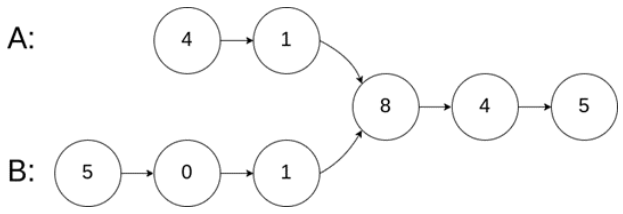
➔ s = m t - j

环是 360 度，所以不管 m 是几圈，从 j 出发走 t-j 步肯定是环入口点。

https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/huan-xing-lian-biao-ii-by-leetcode/

5.5.4 两个链表交点（LC160）

两个链表有交点，可以看成是一个链，有两个分支，而不是两个独立的链。交点就是一个点，或者说两个完全一样的点，而不仅仅是值相同的点，下图交点是 8 而不是 1，因为两个 1 的内存地址不同，是两个点。如果有交点，交点之后的节点都是重复的，形似躺倒的 Y，不可能是 X。



思路一：哈希表，把 A 链表节点放到集合，遍历 B 链表时候看是否在集合中出现过。

思路二（重要）：拼接两个链表，然后找到环的第一个结点，类似 142，首先使用快慢指针法，找到相遇点。然后创建两个指针，一个指针从入口出发，另一个指针从相遇点出发，两指针相遇点就是入口点。

```
def getIntersectionNode(self, headA, headB):
    if not headA or not headB: return None
    last = headA
    while last.next: last = last.next
    last.next = headB
    slow = fast = headA
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        # 有环，接着 slow 从入口出发，fast 从相遇点出发，两指针相遇点就是入口
        if slow is fast:
            fast = headA
            while fast is not slow:
                slow, fast = slow.next, fast.next
            last.next = None
            return slow
```

思路三：消除两个链表长度差，然后一起移动一定会同时到达交点。如果两个链长度相同的话，那么对应的一个个比下去就能找到交点，如果两个链表长度不同，只需要把长链表变短即可。

1. 方法一：先分别求出两个链表长度，再计算长度差，把较长的那个链表向后移动这个差值的个数，然后再一起向后移动，最终两链表同时走到交点。

2. **方法二（虽然上面的要掌握，但是推荐用这个算法）：双指针遍历消除长度差。**pA 和 pB 分别指向 A、B 两个链表的头节点，一起向后遍历，当 pB 到达结尾后就指向 A 链表头节点，当 pA 到达结尾后就指向 B 链表头节点。

假如 B 短，当 pB 到达结尾后就指向 A 链表头节点（此时 pA 和 pB 都在链表 A 上），继续遍历，当 pA 到达结尾后就指向 B 链表头节点，此时 pB 在 A 链表上走过的距离就是两个链表的长度差，pB 已经在 A 链表上走过了它们的长度差，继续一起往后遍历，即可同时走到相交点。

可能有点难理解，原理如下：设 A 的长度为 a + c，B 的长度为 b + c，其中 c 为尾部公共部分长度，可知 a + c + b = b + c + a，你从 A 链表的结尾跳到 B 链表的头节点遍历下去，就相当于遍历了两个链表，同样从 B 链表的结尾跳到 A 链表的头节点遍历下去，也相当于遍历了两个链表，那它们一定会

一起到达结尾，如果结尾不同，一定没交点，因为有交点的话，交点之后的节点都是重复的。

方法二：双指针遍历消除长度差。时间复杂度 O(m+n)。空间复杂度 O(1)。

```
def getIntersectionNode(self, headA, headB):
    if not headA or not headB: return None
    pa, pb = headA, headB
    # 或者 pa，pb 都为 None 退出，或者是同一结点退出。
    # is 比较的是 id，内存地址，== 单纯的比较两个对象是否相同。
    while pa is not pb:
        pa = pa.next if pa else headB
        pb = pb.next if pb else headA
    return pa
```

5.5.5 合并两个排序链表（LC206）

递归，规模太大可能会造成堆栈溢出：

```
def mergeTwoLists(self, l1, l2):
    if l1 is None: return l2
    if l2 is None: return l1
    if l1.val < l2.val:
        l1.next = self.mergeTwoLists(l1.next, l2)
        return l1
    else:
        l2.next = self.mergeTwoLists(l2.next, l1)
        return l2
```

迭代：

```
def mergeTwoLists(self, l1, l2):
    head = cur = ListNode(0)
    while l1 and l2:
        if l1.val < l2.val:
            cur.next = l1
            l1 = l1.next
        else:
            cur.next = l2
            l2 = l2.next
        cur = cur.next
    cur.next = l1 or l2
    return head.next
```

5.5.6 合并 K 个排序链表（LC23）

方法一：暴力求解，所有结点放到一个列表，再遍历排序。链表的题目**讨巧**的话基本都可以用列表来做，把链表的每个值保存到列表，然后对列表做处理就很简单了，对列表排序，翻转等都有对应的内置方法。

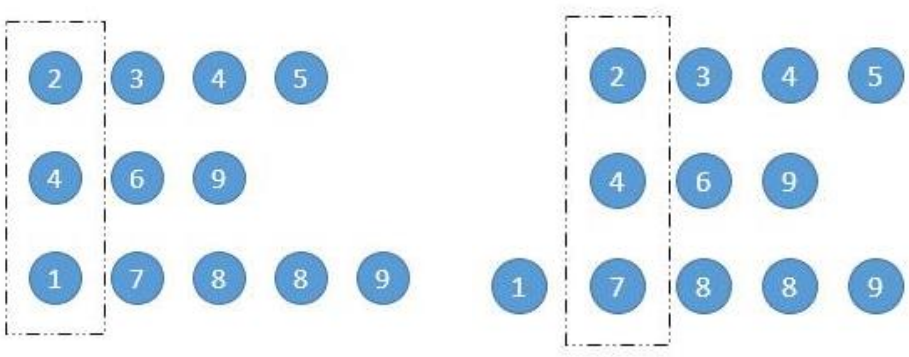
方法二：一列一列的比较。初始列表中保存的是每个链表的头节点，每次找出当前列表中最小的节点 minnode，然后下一次就是其它链表的头节点和 minnode 的下一个节点比较，直到所有链表都比较完，因此还需要一个标记位 isBook，如果 isBook 没发生变化就说明链表都遍历完了：

```
def mergeKLists(self, lists):
    minidx = 0
    head = cur = ListNode(0)
    while True:
        isBook, minnode = True, float('inf')
        for i in range(len(lists)):
            if lists[i]:
                if lists[i].val < minnode:
                    minidx = i
                    minnode = lists[i].val
                isBook = False
        if isBook: break
        cur.next = lists[minidx]
        cur = cur.next
        lists[minidx] = lists[minidx].next
    return head.next
```

博主总结：【23. Merge k Sorted Lists】

时间复杂度 O（kn），假设最长的链表长度是 n ，那么 while 循环将循环 n 次。假设链表列表里有 k 个链表，for 循环执行 k 次。空间复杂度：N 表示最终链表的长度，则为 O（N）。

过程如下图所示：



方法三：和方法二差不多，运用了优先队列：

```
from queue import PriorityQueue
def mergeKLists(self, lists):
    head = cur = ListNode(0)
    q = PriorityQueue()
    # 在 python3 中，优先级一样的话会依次比较，node 之间比较会报错，故加入 idx
    for idx, node in enumerate(lists):
        if node: q.put((node.val, idx, node))

    while not q.empty():
        val, idx, cur.next = q.get()
        cur = cur.next
        # 如果这条链表还有元素，就加入队列继续比较
        if cur.next:
            q.put((cur.next.val, idx, cur.next))
    return head.next
```

【10-line python solution with priority queue】

时间复杂度：O(Nlogk) ，假如总共有 N 个节点，k 是链表的数目，每个节点入队出队都需要 logk，所有时间复杂度是 O(N logk)。弹出操作时，比较操作会被优化到 O(logk) 。同时，找到最小值节点的时间开销仅仅为 O(1)。

空间复杂度： O(n) ，创造一个新的链表需要 O(n) 的开销。同时优先队列（通常用堆实现）需要 O(k) 的空间。

方法四：两两合并，第 0 个和第 1 个合并，新生成的再和第 2 个合并。实际上就是一直调用“合并两个链表”（mergeTwoLists()函数）：

```
def mergeKLists(self, lists):
    if len(lists) == 0:
        return None
    if len(lists) == 1:
        return lists[0]
    head = lists[0]
    for i in range(1, len(lists)):
        head = self.mergeTwoLists(head, lists[i])
    return head
```

时间复杂度：O(kN) ，其中 k 是链表的数目。空间复杂度：O(1)

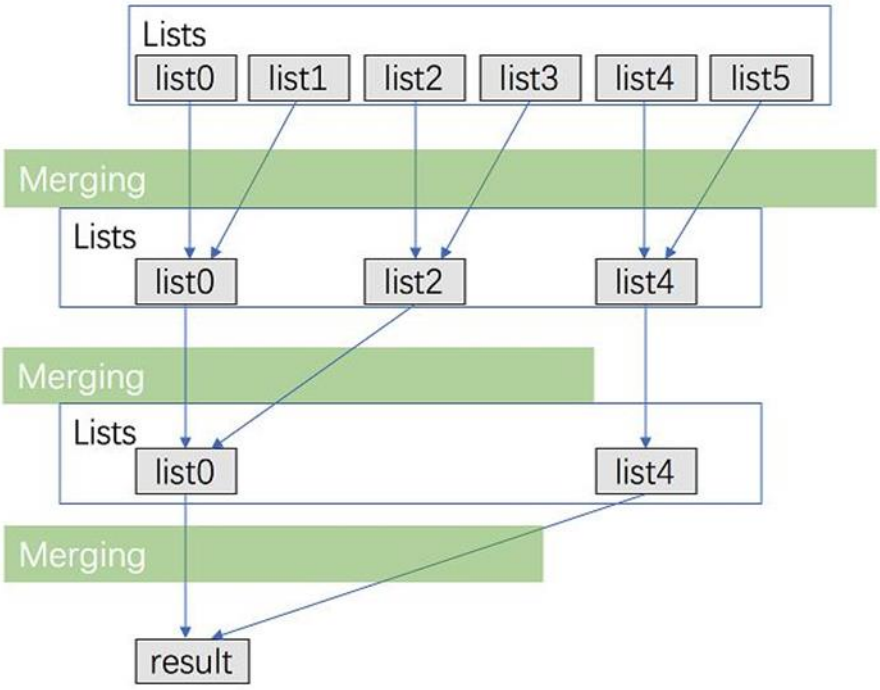
方法五：分治，这个方法沿用了上面的解法，但是进行了较大的优化。我们不需要对大部分节点重复遍历多次。在方法四的基础上，第 0 个和第 1 个合并到 0，第 2 个和第 3 个合并到 2，第 4 个和第 5 个合并到 4，然后第 0

个和第 2 个合并到 0，第 0 个和第 4 个合并到 0。也是一直调用“合并两个链表”（mergeTwoLists()函数）：

```
def mergeKLists(self, lists):
    interv = 1
    while interv < len(lists):
        for i in range(0, len(lists) - interv, interv*2):
            lists[i] = self.mergeTwoLists(lists[i], lists[i+interv])
        interv *= 2
    return lists[0] if len(lists) > 0 else None
```

时间复杂度：O(Nlogk) ，两个链表聚合发生了 K-1 次。但是注意，题解中把 K 个链表两两聚合，生成 K/2 个链表的过程叫一次 Merging。然后这样的 Merging 总共发生 log(K)次。每一次 Merging 需要比较的次数是 N。 所以总的时间复杂度是 O(N*log(K))。

空间复杂度：O(1)



官方文档【23. Merge k Sorted List】

5.6 树的基本概念

了解这些概念主要是为了引出树的遍历和堆。

5.6.1 二叉树

二叉树的特点：

1. 每个结点最多有两棵子树，所以二叉树中不存在度大于 2 的结点。
2. 左子树和右子树是有顺序的，次序不能任意颠倒。
3. 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

二叉树性质

1. 在二叉树的第 i 层上最多有 **2^(i-1)** 个节点（i>=1）。
2. 二叉树中如果深度为 k,那么最多有 2^k - 1 个节点(k>=1）。当有 2^k-1 个节点时，称为**满二叉树**。当除最后一层外，若其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干节点，则为**完全二叉树**。
3. **n0 = n2 + 1**，n0 表示度数为 0 的节点数，n2 表示度数为 2 的节点数。因为 n 个节点的二叉树，节点数满足 n = n0 + n1 + n2，总共会有 n-1 条边，度为 2 的结点有两条边，度为 1 的结点有一条边，所以边数满足 n-1 = 2*n2 + 1*n1，合并可以得出 n0 = n2 + 1 。
4. 在**完全二叉树**中，具有 n 个节点的完全二叉树的**深度**为(log n)+1，其中(log n)是向下取整。
5. 二叉树的**深度是从上到下**计算，**高度是从下到上**计算，树的整体高度和深度是一样的，但是对于某个节点来说不一定相同，但是同一层的节点具有相同的高度和相同的深度。

5.6.2 满二叉树（Full Binary Tree）

在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

满二叉树的特点有：

1. 叶子只能出现在最下一层。
2. 非叶子结点的度一定是 2。
3. 在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多，深度为 k 的满二叉树有 2^k - 1 个节点。
4. **有 n 个节点的满二叉树的深度是 log(n+1)。**

5.6.3 完全二叉树（Complete Binary Tree）

对一颗具有 n 个结点的二叉树按层编号，如果编号为 i(1<=i<=n)的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。

完全二叉数的特点有：

1. 满二叉树一定是完全二叉树，完全二叉树不一定是满二叉树。
2. 最下层的叶子结点集中在树的左部。
3. 倒数第二层若存在叶子结点，一定在右部连续位置。
4. 深度为 k 的完全二叉树,最多有 2^k - 1 个节点（满二叉树），最少有 2^(k-1) 个节点(k>=1）。
5. **n 个节点的完全二叉树的深度是(log n)+1，(log n)向下取整。**其实根据第 4 点可以知道，深度为 k 的完全二叉树，节点数处于一个区间，2^(k-1) <= n <=2^k - 1，可得出 log (n+1) <= k <= (log n)+1，取(log n)+1，(log n)向下取整，如果忘了怎么算就带入几个数字，在线计算一下。

5.6.4 二叉搜索树（Binary Search Tree）

二叉搜索树的特点是：

1. **没有键值相等的节点**。
2. 左子树上所有节点的值均小于它的根节点的值。
3. 右子树上所有节点的值均大于它的根节点的值。
4. 二叉搜索树查找次数不会超过树的深度。即最好的情况下，**二叉查找树的查找效率为 O(log n)，当二叉查找树退化为单链表时，查找效率只为 O(n)**。

对于 BST，由于节点 left < root < right，所以只通过前/后/层次遍历就能重建 BST，所以不需插入额外的字符。

但是对于二叉树，我们知道前+中，后+中才能确定唯一的二叉树（元素不能相同，前+后不行），也可以可以用前/后/层次遍历 + 额外插入"#" 或 "null"来实现，所以 BT 中必须插入额外字符。

5.6.5 平衡二叉树（Balanced Binary Tree）

为了解决二叉搜索树退化为单链表时查找效率低下的问题，引入了平衡二叉树(AVL，这个简写是人名)。平衡二叉树保证查找、插入、删除的时间复杂度稳定在 O(log n)下。

平衡二叉树的性质如下：

1. 父节点的左右两棵子树的深度之差的绝对值不超过 1。
2. 左右子树都是平衡二叉树。

局限性：

AVL 一般是靠旋转维持平衡，维护这种高度平衡所付出的代价并不小，所以实际用的多的是红黑树。当然，如果应用场景中对**插入删除不频繁，只是对查找要求较高，那么 AVL 还是较优于红黑树**。

5.6.6 红黑树（Red Black Tree）

红黑树是一种自平衡二叉查找树，但在每个节点增加一个存储位表示节点的颜色，非红即黑。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，**红黑树确保没有一条路径会比其它路径长出两倍**，因此，红黑树是一种**弱平衡二叉树**（由于是弱平衡，可以看到，在相同的节点情况下，AVL 树的高度低于红黑树），相对于要求严格的 AVL 树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树。

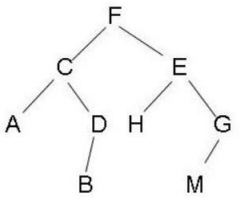
红黑树的性质：

1. 每个节点非红即黑。
2. 根节点是黑的。
3. 每个叶节点都是黑的。
4. 如果一个节点是红的，那么它的两儿子都是黑的。
5. 对于任意节点而言，其到叶节点的每条路径都包含相同数目的黑节点。

【深入学习二叉树(一) 二叉树基础】
【红黑树和 AVL 树（平衡二叉树）区别】

5.7 树的遍历

1. 中序遍历：先左子树，再根节点，最后右子树。
2. 前序遍历：先访问根节点，再访问左子树，最后访问右子树。
3. 后序遍历：先左子树，再右子树，最后根节点。
4. 层序遍历：每一层从左到右访问每一个节点。



无重复元素的二叉树可以用前/后/层次 + 中序确定，但是前序+后序不行，因为前序和后序在本质上都是将父节点与子结点进行分离，但并没有指明左子树和右子树的能力。

只用前/后/层次一种遍历就想确定一棵二叉树的话，就需要额外插入特殊符号，用特殊符号标记叶节点，这样也可以确定一棵二叉树，这也是 297，428 题目中要插入特殊符号的原因。

但是由于 BST 的性质，单用前/后/层次遍历就可以确定一棵 BST：1. 如果给定中序，那严格递增就是 BST；2. 如果给定前序，那一定是先根节点后子节点，根据大小可以确定左右孩子，也就能确定 BST。或者排下序就得到中序，知道前+中序也可以确定一棵树；3. 如果给定后序，同前序，从后往前遍历，一定是先根后孩子。同同样可以排序得到中序。

不管是前序，中序还是后序遍历，不管是递归还是迭代，都需要遍历每个节点一次，都需要利用一个辅助保存每个节点，所以时空复杂度都为 O(n)，n 为结点数。

5.7.1 中序遍历（InOrder Traversal）

递归，如果传入一个全局变量 res，则只需 append，不需要 += ：

```
def in_order(self, root):
    res = []
    if root:
        res = self.in_order(root.left)
        res.append(root.val)
        res += self.in_order(root.right)
    return res
    # return self.in_order(root.left) + [root.val] + \
    #       self.in_order(root.right) if root else []
```

迭代：

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Traversal:
    def in_order2(self, root):
        res, stack = [], []
        cur = root
        while cur or stack:
            if cur:
                stack.append(cur)
                cur = cur.left
            else:
                cur = stack.pop()
                res.append(cur.val)
                cur = cur.right
        return res

if __name__ == '__main__':
    td1, td2 = TreeNode(1), TreeNode(2)
    td3, td4 = TreeNode(3), TreeNode(4)
    td5, td6 = TreeNode(5), TreeNode(6)
    td1.left, td1.right = td2, td3
    td2.right = td4
    td3.left, td3.right = td5, td6
    T = Traversal()
    res1 = T.in_order(td1)
    print(res1)
```

1. 初始化 cur = root（其实直接用 root 也行），只要 cur 不为空，就入栈，然后继续遍历左节点。
2. 如果 cur 为空，就弹出栈顶节点，然后 cur 指向这个节点的右孩子。
3. **弹出顺序就是中序遍历的顺序。入栈顺序其实就是先序遍历。**

总结为：向左边走边入栈，为空出栈向右走。向左走，左节点就是下一个中节点，所以入栈顺序就是先中再左，就是左节点先出，走不通了就指向中节点的右节点（左中右，中序遍历）。

5.7.2 前序遍历（PreOrder Traversal）

递归

```
def pre_order(self, root):
    res = []
    if root:
        res.append(root.val)
        res += self.pre_order(root.left)
        res += self.pre_order(root.right)
    return res
    # return [root.val] + self.pre_order(root.left) + \
    #        self.pre_order(root.right) if root else []
```

迭代：

```
def pre_order2(self, root):
    stack, res = [], []
    cur = root
    while cur or stack:
        if cur:
            res.append(cur.val)
            stack.append(cur.right)
            cur = cur.left
        else:
            cur = stack.pop()
    return res
```

1. 初始化 cur = root（其实直接用 root 也行），只要 cur 不为空，就输出 cur.val，把 cur 右节点入栈，然后继续遍历左节点。
2. 如果 cur 为空，就弹出栈顶节点。
3. **中序遍历的 stack 入栈顺序其实就是先序遍历的顺序**。

总结为：向左走，右入栈，为空就出栈。因为左节点就是下一个中节点，所以遍历顺序就是先中再左，走不通了弹出右。当然也可以用旁边的方法，边遍历边入栈，左节点没了，就弹出栈中节点，然后指向其右节点。

5.7.3 后序遍历（PostOrder Traversal）

后序遍历的顺序是左右中，其实就是中右左的逆序，但中右左可以通过交换先序遍历的 left，right 顺序得到，所以对先序遍历稍作调整就行，先交换左右子树的遍历顺序，再把结果逆序。

递归：

```
def post_order(self, root):
    res = []
    if root:
        res += self.post_order(root.left)
        res += self.post_order(root.right)
        res.append(root.val)
    return res
    # return self.post_order(root.left) + \
    #        self.post_order(root.right) + \
    #        [root.val] if root else []
```

迭代：

```
def post_order2(self, root):
    stack, res = [], []
    cur = root
    while cur or stack:
        if cur:
            res.append(cur.val)
            stack.append(cur.left)
            cur = cur.right
        else:
            cur = stack.pop()
    return res[::-1]
```

5.7.4 层次遍历（Level0rder Traversal）

递归：

```
def level_order(self, root):
    res = []
    if not root: return res
    self.helper(root, 0, res)
    return res

def helper(self, node, level, res):
    if len(res) == level:
        res.append([])
    res[level].append(node.val)
    if node.left:
        self.helper(node.left, level + 1, res)
    if node.right:
        self.helper(node.right, level + 1, res)
```

迭代一：

```
def level_order2(self, root):
    res, queue = [], [root]
    while queue:
        cur = queue.pop(0)
        res.append(cur.val)
        if cur.left:
            queue.append(cur.left)
        if cur.right:
            queue.append(cur.right)
    return res
```

这种方法容易理解，但是有两个缺点，一个是只能按照层次遍历的顺序输出，不能体现哪一层有哪些节点，另一个是 pop(0)复杂度较高。

双向队列 deque.popleft()要比 list.pop(0)更快，因为 deque 已被优化以在 O(1)范围内，而 pop(0)是 O(n)。

迭代二：

```
def level_order2(self, root):
    res, level = [], 0
    if not root: return res
    queue = deque([root])
    while queue:
        res.append([])
        for _ in range(len(queue)):
            node = queue.popleft()
            res[level].append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        level += 1
    return res
```

为什么是弹出一个节点，就把它加入 res？如果弹出一个节点，就把它的两个孩子加入 res 不是更快吗。看看 res.append([])，只要 queue 不为空就加入一个空列表，如果检查一个节点就把其孩子加入 res，会造成节点过早的进入 queue，queue 还没空 res 已经有了全部节点，那在弹出剩下节点的时候，会加入多余的空列表，需要额外处理这些空列表，反而让代码逻辑更复杂。

如果仍然按层遍历，但是每层从右往左遍历怎么办呢？将上面的代码 left 和 right 互换即可。

如果仍然按层遍历，但是我要第一层从左往右，第二层从右往左，第三从左往右...这种 zigzag 遍历方式如何实现？将 res[level].append(node.val) 进行一个层数奇偶的判断，一个用 append()，一个用 insert()，如下：

```
    if level % 2 == 1:
        res[level-1].append(node.val)
    else:
            res[level-1].insert(0, node.val)
```

树有前、中、后、层次几种遍历方式，图是一种特殊的树，其实 **DAG 的拓扑排序，就是逆后序遍历**，因为后序遍历逆序之后就是先父节点后子节点，这符合拓扑排序的性质，当然拓扑排序可能有多个结果，逆后序遍历是其中一个结果。

5.7.5 二叉树深度

递归：

```
def bTreeDepth(self, root):
    if root is None:
        return 0
    ldepth = self.bTreeDepth(root.left)
    rdepth = self.bTreeDepth(root.right)
    return max(ldepth, rdepth) + 1
```

迭代的话其实就是层次遍历，只不过返回的是层数。

【二叉树(前序，中序，后序，层序)遍历递归与循环的 python 实现】

5.8 堆排序（Heapsort）

堆总是满足下列性质：

1. 堆中某个节点的值总是不大于或不小于其父节点的值；
2. 堆总是一棵**完全二叉树**。

Python 中的 heapq 是小根堆。

5.8.1 大根堆（Max Heap）

大根堆：根节点的大于左右子节点。时间复杂度建堆用 O(n)，但这个 O(n) 并不是简单的遍历 n 个节点得出的，具体看下面的复杂度分析，调整堆的时间复杂度是 O(nlog n)，共 O(nlog n)，空间复杂度是 O(1)。

```
#  test = [50, 16, 30, 10, 60, 90, 2, 80, 70]
def heap_sort(self, nums):
    n = len(nums)
    # 创建大根堆，从最后一个父节点开始向上调整
    for idx in range(n//2 - 1, -1, -1):
        self.sift_down(nums, idx, n)
    print("build maxh:", nums) #  [2, 10, 30, 16, 60, 90, 50, 80, 70]

    # 如果要得到一个递增的序列，需要用大根堆，因为父节点是最大值，
    # 首尾交换，得到一个最大值，然后把新的堆调整为大根堆，反复执行。
    # 交换后根节点是一个小值，所以从第一个父节点开始向下调整。
    for idx in range(n - 1, 0, -1):
        nums[idx], nums[0] = nums[0], nums[idx]
        self.sift_down(nums, 0, idx)
    print("sort:", nums) #  [90, 80, 70, 60, 50, 30, 16, 10, 2]

def sift_down(self, nums, start, end):
    """最大堆，向下调整"""
    largest = start
    left, right = 2 * start + 1, 2 * start + 2
    if left < end and nums[largest] < nums[left]:
        largest = left
    if right < end and nums[largest] < nums[right]:
        largest = right
    if largest != start:
        nums[start], nums[largest] = nums[largest], nums[start]
        self.sift_down(nums, largest, end)
```

堆排序过程中有以下几种操作：

1. 首先将待排序的数组构造出一个大根堆。
2. 取出这个大根堆的堆顶节点(最大值)，与堆的最下最右的元素进行交换，然后把剩下的元素再构造出一个大根堆。
3. 重复第二步，直到这个大根堆的长度为 1，此时完成排序。

每个结点的值都大于等于其左右孩子结点的值叫做大根堆，每个结点的值都小于等于其左右孩子结点的值叫小根堆。**堆常被用于优先队列和解决 top K 问题**。

迭代版：

```
def sift_down2(self, nums, start, end):
    largest = start
    left = 2*start+1
    while left < end:
        # 要确保右子节点不越界，让 left 指向左右中大的节点
        if left + 1 < end and nums[left] < nums[left+1]:
            left += 1
        if nums[largest] < nums[left]:
            nums[largest], nums[left] = nums[left], nums[largest]
            # 要谨慎使用连续赋值，这里就必须分开写，这里必须 largest 先变，
            # 连续赋值的话使用的是变化之前的 largest。
            largest = left
            left = 2 * largest + 1
        else:
            break
```

要点是建堆时候从最后一个父节点到根节点的顺序，调整堆是从根节点开始，结束的点是变动的，因为每交换一次首尾节点，就排好一个最大值。

5.8.2 小根堆（Min Heap）

大根堆转换为小根堆很简单，我们知道一组数字中，给最大的数加一个负号就会变成最小的数，所以建堆时候给每个数加负号，输出时候再把负号去掉即可。295. Find Median from Data Stream 就用到了这种方式。

5.8.3 时间复杂度

建堆用 O(n)，但这并不是因为遍历了 n 个节点：

首先要分清二叉树的深度和高度和层数，虽然对于整棵树来说，深度和高度是相同的，但对于某一个节点来说，不一定，深度是从上到下，根节点深度为 1，高度是从下到上，叶节点高度是 1。层数始终是从上往下数的。

具有 n 个元素的完全二叉树，树的高度为 k。那么总的时间计算为：s = 2^(i - 1) * (k - i)，其中 i 表示第几层，2^(i - 1) 表示该层上最多有多少个元素，(k - i) 表示子树上要比较的次数，第 i 层只需要和 i 层以下的比较就行了，i 层以下有多少层呢？有(k - i)层。果在最差的条件下，就是比较次数后还要交换；因为这个是常数，所以提出来后可以忽略。

叶子层不用交换，所以 i 从 k-1 开始（也就是倒数第二层）到 1，只需比较一层，所以 S = 2^(k-2) * 1 + 2^(k-3)*2+...+2*(k-2)+2^(0)*(k-1)。

等式左右乘上 2，然后和原来的等式相减，就变成了：
S = 2^(k - 1) + 2^(k - 2) + 2^(k - 3) ..... + 2 - (k-1)

除最后一项外，就是一个等比数列了，直接用求和公式：

S = $a_1 \cdot \frac{1-q^n}{1-q}$

S = 2^k -k -1；又因为完全二叉树的高度 k = log(n) + 1，综上所述得到：S = 2n - logn -2，所以时间复杂度为：O(n)。

调整堆用 O(nlogn)

每次重建意味着有一个节点出堆，所以需要将堆的容量减 1，n 个节点排序需要循环 n - 1 次，每次都是从根节点往下循环重建，这需要 logn（树的高度），所以总时间是(n-1)logn = nlogn - logn，堆排序的时间复杂度为：O(nlogn)。

【排序算法之 堆排序 及其时间复杂度和空间复杂度】

5.9 二分查找

二分查找的**前提是非递减序列**，本质上是不断折半缩小区间，**难点**在于**边界的控制**，避免陷入**死循环**和**漏查**的情况。一般查找指定的值，right 可以用 len(nums)或 len(nums)-1，但是查找左右边界的话，用 len(nums)更简单。

二分查找应用的场景有很多，比如查找非递减序列中是否存在 val、返回非递减序列中第一个等于 val 的元素位置、返回非递减序列中第一个大于 val 的元素位置等，代码上稍有不同。

C++中三个二分查找相关的函数，我们用 python 改写一下。

5.9.1 binary_search

binary_search(first, last, val)，在**非递减序列**的左闭右开区间[first, last)中查找**是否存在** val ，不存在则返回 -1 。

```
# last 参数语法错误，这里只是给出传入参数的格式。
def search_val1(self, array, first=0, last=len(array), value):
    while first < last:
        # mid = (first + last)//2 是为了防溢出，虽然 python 不用考虑溢出
        mid = first + (last - first) // 2
        if array[mid] < value:
            first = mid + 1
        elif array[mid] > value:
            last = mid
        else:
            return mid
    return -1
```

还有一种写法

```
# last 参数语法错误，这里只是给出传入参数的格式。
# while 用左闭右闭的区间，last 就要用 len(array)-1，如果不减 1 会越界。
def search_val8(self, array, first=0, last=len(array)-1, value):
    while first <= last:
        mid = first + (last - first) // 2
        if array[mid] < value:
            first = mid + 1
        elif array[mid] > value:
            last = mid - 1
        else:
            return mid
    return -1
```

5.9.2 lower_bound

lower_bound(first, last, val)，在**非递减序列**的**左闭右开**区间 [first, last) 中进行二分查找，返回**大于等于 val 的第一个元素位置**。如果所有元素都小于 val，则返回 last 的位置，此时 last 的位置是越界的。

```
# last 参数语法错误，这里只是给出传入参数的格式，
def lower_bound(array, first=0, last=len(arrar), value):
    while first < last:
        mid = first + (last - first)//2  # 防溢出
        if array[mid] < value:
            first = mid + 1
        else:
            last = mid
    return first  # last 也行，因为[first, last)为空的时候它们重合

if __name__ == '__main__':
    test = [1,3,4,4,4,5,6]
    val = 4
    res1 = lower_bound(test, 0, len(test), val)
    print(res1)
```

如果想找**小于等于 val 的最后一个元素位置**，可以使用 upper_bound() - 1 得到。比如“0 1 2 3 3 4 4 4 8”，可以用 upper_bound()找到下标 8，减一找到小于等于 4 的最后一个元素（即第三个 4）。

5.9.3 upper_bound

upper_bound(first, last, val)，在**非递减序列**的**左闭右开**区间 [first, last) 中进行二分查找，返回**大于 val 的第一个元素位置**。如果 val 大于数组中全部元素，则返回 last 的位置，此时 last 的位置是越界的。跟上面函数的区别仅仅是 if array[mid] <= value。

```
# last 参数语法错误，这里只是给出传入参数的格式，
def upper_bound(self, array, first=0, last=len(arrar), value):
    while first < last:
        mid = first + (last - first)//2  # 防溢出
        if array[mid] <= value:
            first = mid + 1
        else:
            last = mid
    return first # last 也行，因为[first, last)为空的时候它们重合
```

如果想找 小于 val 的最后一个元素位置，可以使用 lower_bound() -1 得到。比如“0 1 2 3 3 4 4 4 8”，可以用 lower_bound()找到下标 5，减一找到小于 4 的最后一个元素（即第二个 3）。

5.9.4 复杂度和边界问题

时间复杂度：如果有 n 个元素，那么二分后每次查找的区间大小就是 n，n/2，n/4，…，n/2^k，其中 k 就是循环的次数，最坏的情况是 k 次二分之后，每个区间的大小为 1,找到想要的元素，令 n/2^k=1，可得 k=logn,（是以 2 为底，n 的对数），所以时间复杂度为 O(logn)。空间复杂度：递归的话是 O(logn)，迭代的话是 O(1)。

二分查找的原理是利用区间内数值有序的特点，不断让可行区间减半，最终可行区间长度减到 1 得到答案，要保证二分查找能得到正确答案，并且不会漏找和陷入死循环，有几点要注意:

1. **传参时候区间整体大小用 len(arrar)还是 len(arrar)-1?**
   答：看使用左闭右开还是左闭右闭区间，和后面的 while 对应即可。

2. **用 while first < last 还是 while first <= last?**
   答：其实就是上面说的左闭右开还是左闭右闭，如果整体区间用 len(arrar)-1，为了不漏找，那就对应的使用 while first <= last，如果整体区间用 len(arrar)，那就对应的使用 while first < last，取到最后一个值反而会越界。

3. **比较大小时候用 array[mid] <= value 还是 array[mid] < value?**
   答：再看一看 lower_bound()、upper_bound()和 binary_search()的区别就知道了。

4. **每次判断后要缩小区间，那是用 first = mid+1，last = mid 还是 first = mid + 1，last = mid-1?**
   答：这取决于 while 怎么写，若 while first < last，是左闭右开，
   a. 如果下一次取右侧区间，那我们 first 右移即可，因为 mid 已经比较过，所以是 first = mid+1，
   b. 如果下一次取左侧区间，那我们 last 左移即可，要 last = mid，因为左闭右开的区间取不到最右侧，实际上只取到了 mid-1。
   c. 若 while first <= last，是左闭右闭，那就必须 first = mid + 1，last = mid-1。

【细说二分查找（Binary Search）】
【LeetCode Binary Search Summary 二分搜索法小结】
【二分查找细节详解】

5.10 图的基本概念

5.10.1 连通图和非连通图

1. 连通图：**所有的点都有路径相连**，这需要**至少 n-1** 条边。路径是从一个顶点到另一顶点途径的所有顶点组成的序列，**若路径中 各顶点都不重复则此路径又被称为“简单路径”**。
2. 非连通图：存在某两个点没有路径相连。
3. 有向图中的连通图称为强连通图。

5.10.2 完全图

每个顶点都与除自身之外的其他顶点恰有一条边相连就是完全图：

1. 若 G 是有向图，则边数范围在 0 ≤ e ≤ n(n-1),**恰有 n(n-1)条弧的有向图称为有向完全图**(Directed Complete Graph)，因为每个顶点到其它 n-1 个点都有一条弧。
2. 若 G 是无向图，则边数范围是 0 ≤ e ≤ n(n-1)/2，**恰有 n(n-1)/2 条边的无向图称无向完全图**(Undireet-ed Complete Graph)，注意边和路径的区别。

5.10.3 稠密图、稀疏图

边数很少的图称为稀疏图（Sparse graph），反之称为稠密图（Dense graph）。稀疏图适合用邻接表存储，稠密图适合用邻接矩阵存储。稀疏图和稠密图的具体定义是比较模糊的，一般有两种衡量方法，图 G 有 n 个顶点时：

1. 若边数满足 e > nlogn 则为稠密图，反之为稀疏图。
2. 从量级上看，若边数 e 接近 O(n^2)为稠密图，接近 O(n)为稀疏图。

5.10.4 极大极小连通子图

极大极小是**针对边数**：

1. 极大连通子图，包含**最多边**的连通子图。**对于连通图，极大连通子图其实就是它本身**。
2. 极小连通子图，包含**最少边**的连通子图。**对于连通图，极小连通子图其实就是它的生成树**，加一条边就会产生环，减一条边就无法构成完整的生成树。

5.10.5 连通分量

连通分量（Connected Component）指的就是无向图中的**极大连通子图**（注意不是最大，最大子图只有一个，连通分量可以是多个）。任何连通图的连通分量只有其自身一个，非连通的无向图有多个连通分量，如下 G3 是非连通图，有三个连通分量。有向图中的极大强连通子图称为有向图的强连通分量。

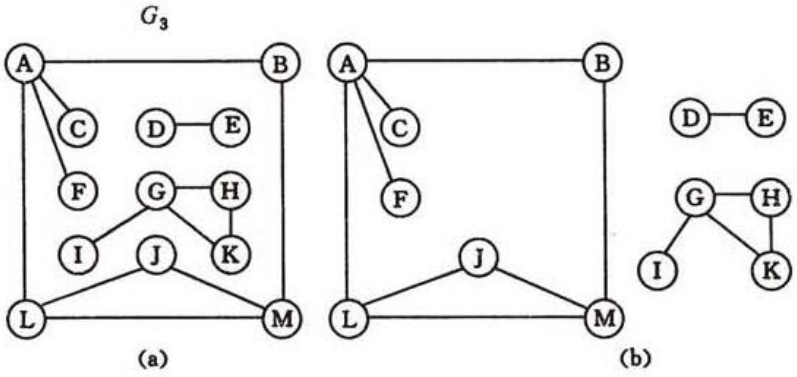


图 7.3　无向图及其连通分量

（a）无向图 G_3；（b）G_3 的 3 个连通分量

5.10.6 最小生成树

一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的 n - 1 条边，一个连通图可能有多个生成树，权值和最少的就是最小生成树。

5.11 图的遍历

图的遍历问题可以分为四类：

1. 遍历完所有**边**而**没有重复**，即所谓“欧拉路径问题”（又名一笔画问题）。
2. 遍历完**所有顶点**而**没有重复**，即所谓 “哈密顿路径问题”。
3. 遍历完**所有边**而**可以重复**，即所谓“中国邮递员问题”。
4. 遍历完**所有顶点**而**可以重复**，即所谓“旅行推销员问题”。

对于第一和第三类问题已经得到了完满的解决，而第二和第四类问题则只得到了部分解决。第一类问题就是研究所谓的欧拉图的性质，而第二类问题则是研究所谓的哈密顿图的性质。

图的遍历（Traversing Graph）中基础算法是 DFS 和 BFS，这是求解图的连通性问题、拓扑排序和求关键路径等算法的基础，**DFS 和 BFS 是以遍历所有顶点**为目标。

不管 BFS 还是 DFS，时间复杂度上采用邻接矩阵存储则为 O(n^2)；采用邻接表存储则为 O(n+e)，因为每个点和边都要访问一次。空间上都是 O(n)。

5.11.1 BFS

时间：邻接矩阵和邻接表分别是 O(n^2)和 O(n+e)，空间都是 O(n)。注意白框中的转换，**转换后的字典要体现叶子节点**，比如 [[0, 1], [0, 2], [1, 3]] 要转换成{0: [1, 2], 1: [3], 2: [], 3: []}，**2 和 3 为空但不能省**。

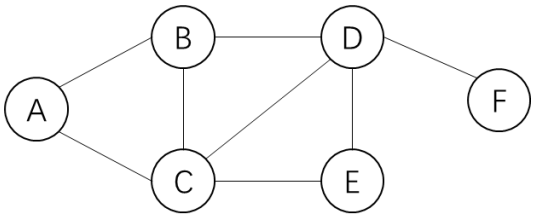
```
from collections import deque
class Solution:
def bfs_iterate(self, graph, start):
        visited, q = {start}, deque([start])  # visited 保存访问过的点
        while q:
            vertex = q.popleft()
            for w in graph[vertex]:
                if w not in visited:
                    visited.add(w)
                    q.append(w)
            print(vertex)

if __name__ == '__main__':
    S = Solution()
    graph = {'A': ['B', 'C'], 'B': ['A', 'E', 'D'],
             'C': ['A', 'F'], 'D': ['B'],
             'E': ['B', 'F'],  'F': ['C', 'E']}
    res = S.bfs_iterate(graph, "A")
```

5.11.2 DFS

时间：邻接矩阵存储是 O(n^2)，邻接表存储是 O(n+e)。空间都是 O(n)。

```
def dfs_iterate(self, graph, start):
    visited, stack = {start}, [start]
    while stack:
        vertex = stack.pop()
        for w in graph[vertex]:
            if w not in visited:
                visited.add(w)
                stack.append(w)
        print(vertex)
```

只需要把 BFS 中的队列换成栈即可，但从遍历顺序上看，这不是一个完美的 DFS，因为特定的条件下这个算法的遍历顺序并不是严格的 DFS，看下面的例子：



如果走的是 “ABDE” 路线，我们的算法接下来结果就是 “ABDEFC” ，而正确的答案应该是 “ABDECF”。

出问题的原因是：当前顶点未被访问过，则把它的相邻顶点标记访问并入栈，简单说就是只判断父节点就处理了子节点（当然树和图不一样，此处为了便于理解）。但这种写法可以和 BFS 用一个模板，对顺序要求没那么严格的话还是很好用的，想要严格的 DFS 遍历顺序可以如下处理：

```
# 时间：邻接矩阵存储是 O(n^2)，邻接表存储是 O(n+e)。空间：都是 O(n)
def dfs_iterate(self, graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            for w in graph[vertex]:
                if w not in visited:
                    stack.append(w)
            print(vertex) # 按顺序输出访问节点
    return visited # 集合无序，只代表这些节点访问过。
```

递归：

```
# 时间：邻接矩阵存储是 O(n^2)，邻接表存储是 O(n+e)。空间：都是 O(n)
def dfs_recursion(self, graph, start, visited=None):
# visited 只用于确认节点是否访问过，不需要按顺序保存的话可以换成集合。
    if not visited: visited = []
    visited.append(start)
    for v in graph[start]:
        if v not in visited:
            self.dfs_recursion (graph, v, visited)
    return visited
```

【数据结构：图之 DFS 与 BFS 的复杂度分析】

5.11.3 欧拉路径（一笔画问题）

一笔画问题即：是否存在着一个恰好包含了所有的边，并且没有重复的路径？用图论的术语来说，就是判断这个图是否是一个能够遍历完所有的边而没有重复,这样的图称为欧拉图，遍历的路径称作欧拉路径（Eulerian Tour），如果路径闭合，则称为欧拉回路（Eulerian circuit）。

连通的**无向图**是**欧拉路径**的充要条件是：G 中**奇顶点**（连接的边数量为奇数的顶点）的**数目等于 0 或者 2**。或者说图中奇顶点数目不多于 2 个，这是因为奇顶点数目不可能是 1 个。

连通的**无向图**是**欧拉环**（存在欧拉回路）的充要条件是： G 中**每个顶点**的度都是**偶数**。

连通的**有向图**是**欧拉路径**的充要条件是：从顶点 u 到 v ，**u 的出度比入度多 1，v 的出度比入度少 1**，而**其它顶点的出度和入度都相等**。

连通的**有向图**是**欧拉环**（存在欧拉回路）的充要条件是：**所有顶点**的**入度等于出度**。

欧拉路径的算法是 Hierholzer 算法（逐步插入回路法）：

```
1.判断有无解,确定起点
2.开始递归函数 Hierholzer(x):
    循环寻找与 x 相连的边(x,u):
        删除(x,u),删除(u,x)
        Hierholzer(u);
    将 x 插入 res 之中
3.逆序 res 就是结果
```

对每个点的所有邻接点做 DFS，每个访问过的边都删除（通过 pop 对应的节点实现），把**没有邻接点**的节点依次都加入 res 数组，然后**逆序**就是结果。

假设一定存在欧拉路径，那如果 DFS 走到最后一个点，却是死路，那一定有其它路径连到这个点，这个点就是最后一个点，所以最后还需要逆序。

每个顶点的度都是偶数，满足无向图的欧拉环，从 1 开始，
删边 1-2 递归到 2，删边 2-3 递归到 3，删边 3-7 递归到 7，
删边 7-1 递归到 1，
1 无边加入队列，返回到 7，7 无边加入队列，返回到 3，
删边 3-4 递归到 4，删边 4-5 递归到 5，删边 5-6 递归到 6，
删边 6-3 递归到 3，
3 无边加入队列，返回到 6，6 无边加入队列，返回到 5，
5 无边加入队列，返回到 4，4 无边加入队列，返回到 3，
3 无边加入队列，返回到 2，2 无边加入队列，返回到 1，
1 无边加入队列，结束。
答案队列为：1 7 3 6 5 4 3 2 1。反向输出即为答案。

比如 leetcode 332 就是一个欧拉路径的问题，给定起点"JFK"，找出一条路径遍历每个行程，tickets = [["JFK", "SFO"], ["JFK", "ATL"], ["SFO", "ATL"], ["ATL", "JFK"], ["ATL", "SFO"]]：

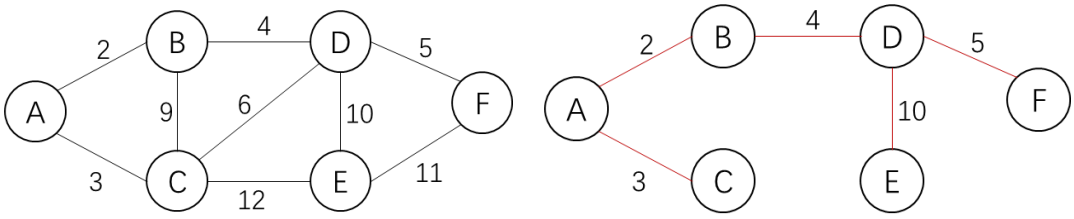
```
def findItinerary(self, tickets):
    #targets = collections.defaultdict(list)
    targets = {i: [] for i, j in tickets}
    for i, j in sorted(tickets)[::-1]:
        targets[i].append(j)
    route = []
    self.visit(targets, route, "JFK")
    return route[::-1]

def visit(self, targets, route, airport):
    # 注意 for 和 while 的区别，比如 for i in range(n)，i 的改变不影响 for 循环
    while targets[airport]:
        self.visit(targets, route, targets[airport].pop())
    route.append(airport)
    return
```

时间复杂度应该是 O（（V + E）^ 2），因为在访问每个新节点时，我们最多会有 V + E 分支，每次迭代中可能执行 E + V 工作。

【欧拉回路与欧拉路径浅析】

5.11.4 最小生成树

最小生成树（Minimum Spanning Trees，MST），包含图中**所有顶点**且**没有环**的**子图**就是**生成树**，一个连通图可能有多个生成树，**权值和最少的就是最小生成树**。对于非连通无向图来说，它的每一连通分量同样有最小生成树，它们的并被称为最小生成森林。本章的代码经常会以下图为例，下图以字母为顶点，代码中以数字为顶点，顶点“ABCDEF”对应“012345”。



最小生成树的算法有 Prim 和 Kruskal，它们的**核心思想都是贪心算法**。**Prim 算法一般用于求解稠密图**，稠密图常用邻接矩阵存储，**Kruskal 算法一般用于求解稀疏图**，稀疏图常用邻接表存储。Prim 算法时间复杂度是 O（n^2），Kruskal 算法的时间复杂度 O（ElogE）。但 Prim 算法可以用**邻接表**加**堆优化**把复杂度降低至 O（ElogV）。

为什么要强调稠密图，稀疏图，邻接矩阵，邻接表？因为这将决定使用 Prim 还是 Kruskal，一般 Prim 算法适合求解稠密图，稠密图适合用邻接矩阵存储；Kruskal 算法适合求解稀疏图，稀疏图适合用邻接表存储。

5.11.4.1 Prim 算法

Prim（普里姆算法）算法**每次选择一个与当前顶点集最近的一个顶点，并将两顶点之间的边加入到树中，直到所有顶点都被访问**。该树最开始只有一个顶点，然后会添加 n-1 个边，时间复杂度为 O(n^2)，Prim 算法不依赖于边。

对于 n 个顶点的图，最外层的 n - 1 次循环就能把所有顶点都访问到，里面第一个循环找距离当前顶点最近的顶点 minnode，第二个循环把 dist 更新为从 minnode 出发到其他还未访问过的顶点，的最短距离。最多执行 n-1 次操作，而内部两个循环的时间复杂度都是 O(n)，总时间复杂度 O(n^2)。

数组 visited，用来保存已确认的顶点，每次从未访问过的点中找下一个点，所以不会成环。

字典 disit，key 是顶点，value 是该点到邻接点的最小权值，初始只有起点 dist[start] = 0，其它都还未访问，所以设无穷大，如果给定的顶点用数字表示，一维数组就足够了，但是如果给定顶点用字母表示，就需要字典了：

```python
class Solution:
    # 初始起点到自己的距离是 0，到其它点距离是 inf
    def prim1(self, graph, n, start):
        visited = []
        dist = {i: float('inf') for i in range(n)}
        dist[start] = 0
        # parent 保存每个顶点的父节点，可用于打印路径，totalweight 是总权值
        parent, totalweight = {}, 0
        for _ in range(n):
            minnode, weight = None, float('inf')
            for i in range(n):
                if i not in visited and dist[i] < weight:
                    minnode, weight = i, dist[i]

            totalweight += weight
            visited.append(minnode)
            for j in range(n):
                if j not in visited and graph[minnode][j] < dist[j]:
                    dist[j] = graph[minnode][j]
                    parent[j] = minnode
        print(parent, totalweight)

if __name__ == '__main__':
    S = Solution()
    inf = float('inf')
    vertex_num, start = 6, 0
    test = [[0,   2  , 3  , inf, inf, inf],
            [2,   0  , 9  , 4  , inf, inf],
            [3,   9  , 0  , 6  , 12 , inf],
            [inf, 4  , 6  , 0  , 10 , 5  ],
            [inf, inf, 12 , 10 , 0  , 11 ],
            [inf, inf, inf, 5  , 11 , 0  ]]
    S.prim1(test , vertex_num, start)
```

用堆和邻接表可以降低 Prim 的复杂度，其实邻接表的话直接用 Kruskal 算法就是 O（ElogE），不过这里还是写一下用小根堆和邻接表优化 Prim。

我们把(e，v1，v2)保存到堆 q 中，就会按边排序，q 每次 pop 都是最小边，然后判断 v2 是否访问过，没访问过就是这次要选取的顶点，接下来把 v2 相邻的，未被访问过的点和边加入队列，直到所有的点都访问完。

如果 V 个顶点 E 条边，外层 while 循环需要 V-1 次选定 V-1 条边。因为是无向图，for 循环总共（算上 while）会对 2E 条边进行判断（但只有 E 条边会加入堆中），堆排序是 O(log E)所以总共 O（ElogE）。堆 heapq.heappop()弹

出最小值后需要调整，复杂度是 O（log E）所以总共 O（VlogE）。O(ElogE + VlogE) = O(E logE)，网上一般都是 O(E logV)，因为 E 基本等于 V^2，O(logE) = log(V^2) = 2*log(V) = O(log(V)。

```
import heapq
class Solution:
    def prim2(self, graph, n, start):
        q = [(start, start, 0)]
        heapq.heapify(q)
        visited, dic = [], {}
        # mstpath 保存每组边
        mstpath, totalweight = [], 0
        """转换成字典格式如下，注意如何不漏掉 5: {}
        {0: {1: 2, 2: 3}, 1: {3: 4, 2: 9}, 2: {3: 6, 4: 12},
         3: {4: 8, 5: 5},  4: {5: 11}, 5: {}}
        """
        for v1, v2, e in graph:
            if not dic.get(v2): dic[v2] = {}
            if dic.get(v1): dic[v1][v2] = e
            else: dic[v1] = {v2: e}
        print(dic)
        while q and n > 0:
            e, v1, v2 = heapq.heappop(q)
            if v2 not in visited:
                visited.append(v2)
                mstpath.append((v1, v2, e))
                totalweight += e
                n -= 1
                for v in dic[v2]:
                    if v not in visited:
                        heapq.heappush(q, (dic[v2][v], v2, v))
        print(mstpath, totalweight)

if __name__ == '__main__':
    S = Solution()
    vertex_num = 6
    test2 = [[0, 1, 2], [0, 2, 3], [1, 3, 4],
            [1, 2, 9], [2, 3, 6], [2, 4, 12],
            [3, 4, 10], [3, 5, 5], [4, 5, 11]]
    res2 = S.prim2(test2, vertex_num)
```

如果想在邻接表时候用 Prim，需要把数组处理成字典，转字典的难点在于容易漏掉顶点，如上列中的 5: {}，其实对于无向图来说，这个点有没有或者是否为空都不影响表达，但有这一个 key 会简化我们求解过程。

【Time Complexity of Prims Algorithm?】

【How Prim's algorithm time complexity is ElogV using Priority Q?】

【求无向连通图的最小生成树算法——Prim 与 Kruskal 及相关优化】

5.11.4.2 Kruskal 算法

Kruskal（克鲁斯卡尔）算法是**先按照边的权重（从小到大）进行排序，然后再依次加入生成树中，若加入某条边会成环则不加入该边。直到树中含有 V-1 条边为止**。一般用**并查集**实现（并查集的详解和优化看后面章节）。

Kruskal 算法的时间复杂度为 O(Elog E)，排序所有边需要 O(ELogE)的时间，排序后还需要对所有边应用 Union-find 算法，Union-find 操作最多需要 O(logV)时间，再乘以边数 E，所以就是 O(ElogV)。O(ELogE + ELogV)。

```
class Solution:
    def kruskal(self, connections, n):
        # 首先初始化 parent，每个顶点的父节点都是自己，然后还要按权重排序。
        parent = {i: i for i in range(n)}
        connections.sort(key=lambda item: item[2])
        print(parent, connections)
        mstpath, k = [], 0
        # N 个顶点需要 N-1 条边，所以用 n - 1 > 0，k 用来遍历 connectios。
        while n - 1 > 0:
            v1, v2, weight = connections[k]
            k += 1
            v1_root = self.find_root(v1, parent)
            v2_root = self.find_root(v2, parent)
            if v1_root != v2_root:
                mstpath.append((v1, v2, weight))
                self.merge_node(v1_root, v2_root, parent)
                n -= 1
        print(mstpath)

    def find_root(self, node, parent):
        while node != parent[node]:
            node = parent[node]
        return node

    def merge_node(self, node1, node2, parent):
        parent[node2] = node1
```

```
if __name__ == '__main__':
    S = Solution()
    vertex_num = 6
    test = [[0, 1, 2], [0, 2, 3], [1, 3, 4],
            [1, 2, 9], [2, 3, 6], [2, 4, 12],
            [3, 4, 10], [3, 5, 5], [4, 5, 11]]
    S.kruskal(test, vertex_num)
```

如果想用字母表示顶点怎么办？很简单，加一个字典表示数字和字母的对应关系即可，dic = {0: 'A', 1: 'B', 2: 'C', 3:'D', 4: 'E', 5: 'F'}。

5.11.5 最短路径

1. **确定起点的最短路径问题** - 即已知起始结点，适合用 Dijkstra 算法，但 Dijkstra 是贪心算法，所以不适用于有负权值的情况。
2. **确定终点的最短路径问题** - 与确定起点的问题相反，相当于把所有路径方向反转的确定起点的问题。
3. **确定起点和终点的最短路径问题** - 已知起点终点求两点之间的最短路径。
4. **任意两点最短路径问题** - 求任意两点的最短路径，适合用 Floyd 算法,但有负权回路的时候不能用 Floyd 算法，负权回路每走一圈都会更小。

最小生成树能够保证这个生成树的路径总和是最小的，但不能保证任意两点之间是最小的。最短路径是从一点出发到达终点的路径最小。而且一个**有效的最短路径，默认是没有负权环的**，因为负权环会随着循环无限减小，无解。

5.11.5.1 Dijkstra 算法

Dijkstra（迪杰斯特拉）算法是典型的**单源最短路径**算法（SSSP：Single-Source Shortest Path），用于计算**从一个顶点到其余各顶点的最短路径**，主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止，时间复杂度和 Prim 一样，邻接矩阵时为 O(n^2)，也可以用邻接表+堆优化把复杂度降低至 O（ElogV）。

Dijkstra 算法**不适用于有负权值**的情况，因为 Dijkstra 是贪心算法，贪心算法成功的前提是局部最优解即全局最优解，但**存在负权的话不能保证局部最优解就是全局最优解**，负权值可以使用 Bellman-Ford 算法。（注：也有人认为 Dijkstra 是 DP，维基百科也把 Dijkstra 列为 DP 的一个案例，但我觉得 Dijkstra 严格说不属于任何一种，它利用贪心算法，在每一步做出局部最优解，以此希望去寻找总体的最小值去解决问题。）

思想是：把图中顶点分成两组，用 S 表示已求出最短路径的顶点集合，用 U 表示其余未确定最短路径的顶点集合。初始时 S 中只有一个源点 v，以后每求得一条最短路径 ，就将对应的顶点加入到集合 S 中，直到全部顶点都加入

到 S 中就结束了。每加入 S 一个点，就要比较 U 中顶点经过这个点到达源点是否距离更近，是的话更新 U 中距离。

说这么多其实和 Prim 算法几乎一样，如果也用邻接矩阵存储，只有两处不同：1. 最后一个 for 循环对边做松弛的两行代码稍有改变，代码如下；2. dist 含义不同，dist 的 key 仍代表顶点，但 value 代表起点到该顶点的路径长度：

```
    if j not in visited and graph[minnode][j] + weight < dist[j]:
    dist[j] = graph[minnode][j] + weight
```

因为 Prim 是求最小生成树，只比较两点之间的边做松弛，Dijkstra 是最短路径，比较两点之间的路径做松弛。

和 Prim 算法一样，Dijkstra 也可以用邻接表+堆优化进行版本，代码也和 Prim 十分相似。所以复杂度也是 O(E logE)或者 O(E logV)，因为 E 基本等于 V^2，O(logE) = log(V^2) = 2*log(V) = O(log(V)。

```
class Solution:
    def dijkstra(self, graph, n, start):
        q = [(start, start, 0)]
        heapq.heapify(q)
        visited, dic = [], {}
        # spath 保存起点，终点和路径长度，parent 保存每个顶点的父节点
        spath, parent = [], {}
        """转换成字典格式如下，注意如何不漏掉 5: {}
        {0: {1: 2, 2: 3}, 1: {3: 4, 2: 9}, 2: {3: 6, 4: 12},
         3: {4: 8, 5: 5},  4: {5: 11}, 5: {}}
        """
        for v1, v2, e in graph:
            if not dic.get(v2): dic[v2] = {}
            if dic.get(v1): dic[v1][v2] = e
            else: dic[v1] = {v2: e}
        print(dic)
        while q and n > 0:
            e, v1, v2 = heapq.heappop(q)
            if v2 not in visited:
                visited.append(v2)
                spath.append((start, v2, e))
                parent[v2] = v1
                n -= 1
                for v in dic[v2]:
                    if v not in visited:
                        heapq.heappush(q, (dic[v2][v]+e, v2, v))
        print("spath:%s \nparent:%s" % (spath, parent))

if __name__ == '__main__':
```

```
    S = Solution()
    test2 = [[0, 1, 2], [0, 2, 3], [1, 3, 4],
            [1, 2, 9], [2, 3, 6], [2, 4, 12],
            [3, 4, 10], [3, 5, 5], [4, 5, 11]]
    res = S.dijkstra(test2, 6, 0)
```

为什么有负权（注意区别负权和负权环）的情况下 Dijkstra 算法会失效？如下左侧的图是一个经典的反面案例，但实际上在某些情况下这个图可以用 Dijkstra 算法得到正确解：

【Negative weights using Dijkstra's Algorithm】

5.11.5.2 Floyd 算法

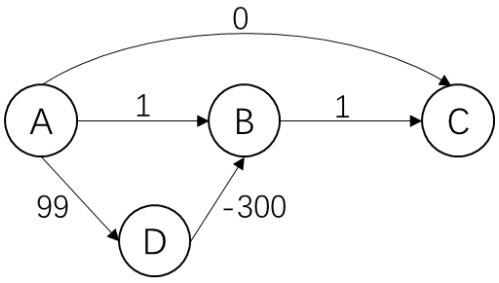
Floyd-Warshall（弗洛伊德）算法是典型的**多源最短路径算法**，用于计算**任意两个顶点之间**的最短路径，能求解**有负权边但无负权环的最短路径**问题，同时也被用于计算有向图的传递闭包。无向图实际上是双向的，只要有负权边就有负权环（从负权边出去再沿着负权边原路进来），所以 Floyd 实际上是针对**无负权边的无向图**，和**无负权环的有向图**。

其实对每个顶点做 Dijkstra 也可以求得任意两顶点的最短路径，时间复杂度是 O(n^3)，但 Floyd-Warshall 算法代码算法简洁，容易理解，时间复杂度也是 O(n^3)，空间复杂度为 O(n^2)，Floyd 其实就是暴力求解：

```
class Solution:
    def floyd(self, graph):
        # path 矩阵保存终点的父节点，初始化 -1 代表起点终点之间没其它点。
        n = len(graph)
        path = [[-1] * n for _ in range(n)]
        print(path)
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if graph[i][k] + graph[k][j] < graph[i][j]:
                        graph[i][j] = graph[i][k] + graph[k][j]
                        path[i][j] = k
                        # print(i, k, j, graph[i][k] + graph[k][j])
        print(path)
        self.getPath(0, 2, path)

    def getPath(self, i, j, path):
        # 按照“终点->起点”的顺序保存，最后逆序输出
        res = [j]
        if i == j: return [i, i, j]
        while path[i][j] != -1:
```

```
            res.append(path[i][j])
            j = path[i][j]
        res.append(i)
        print(res[::-1])

if __name__ == '__main__':
    S = Solution()
    test = [[0,   1,    0,   99],
            [inf, 0,    1,   inf],
            [inf, inf,  0 ,  inf],
            [inf, -300, inf, inf]]
    res = S.floyd(test)
```

Floyd 算法运用了动态规划的思想。用通俗的语言来描述的话，从顶点 i 到顶点 j 的最短路径不外乎两种可能，直接从 i 到 j ，或者从 i 经过若干个节点 k 到 j。所以，我们假设 dis(i, j) 为顶点 v 到顶点 u 的最短路径距离，对于每一个顶点 k，我们检查 dis(i, k) + dis(k, j) < dis(i, j) 是否成立，如果成立，证明从 i 到 k 再到 j 的路径比 i 直接到 j 的路径短，于是设置 dis(i, j) = dis(i, k) + dis(k, j)，这样一来，当我们遍历完所有节点 k，dis(i, j) 中记录的便是 i 到 j 的最短路径的距离。



【算法 6：只有五行的 Floyd 最短路算法】
【Floyd 算法(一)之 C 语言详解】

5.11.5.3 Bellman-Ford 算法

Bellman - ford（贝尔曼-福特）算法也是**单源最短路径**算法，可以判断**是否有负权环**，并求解**有负权边但无负权环**的单源最短路径问题，但效率较低。其原理为连续进行松弛，在每次松弛时把每条边都更新一下，若在 n-1 次松弛后还能更新，则说明图中有负权环，因此无法得出结果，否则就完成，有动态规划的思想。

Bellman-Ford 算法有点找最小生成树的影子，算法思想是循环|V-1|次，每次都对所有的边进行 relax 操作，最后再一次 relax 操作，如果还能被 relax，就说明存在负环。**下面代码加了 flag 进行优化**，一旦扫描所有边后没有进行 relax，可以**提前退出**，不必循环完 V-1 次：

```python
class Solution:
    def bellman_ford(self, graph, start):
        # path 矩阵保存终点的父节点，初始化 -1 代表起点终点之间没其它点。
        n = len(graph)
        dist = {i: float('inf') for i in graph}
        cnt = {i: 0 for i in graph}
        dist[start], cnt[start] = 0, 1
        parent = {}
        for _ in range(n - 1):
            flag = 0
            for node in graph:
                for v in graph[node]:
                    if dist[v] > dist[node] + graph[node][v]:
                        dist[v] = dist[node] + graph[node][v]
                        parent[v] = node
                        flag = 1
            if flag == 0: break
        print(dist, parent)

        for node in graph:
            for v in graph[node]:
                if dist[v] > dist[node] + graph[node][v]:
                    return -1

if __name__ == '__main__':
    S = Solution()
    test = {'a': {'b': 24, 'c': 8, 'd': 15},
            'b': {'e': 6},
            'c': {'f': 3, 'e': 7},
            'd': {'g': 4},
            'e': {'g': 9},
            'f': {'d': 5, 'e': 2, 'g': 3},
            'g': {'b': 3}}
    res = S.bellman_ford(test)
```

Bellman-Ford 算法的时间复杂度为 O(VE)。为什么要循环 n-1 次？ 因为最短路径肯定是个简单路径，不可能包含回路， 如果包含回路且回路的权值和为正的，那么去掉这个回路就可以得到更短的路径。如果循环 n-1 次后，还有权值不断缩小，说明存在负权环，正权环的总权值每次循环只会不断增加。

5.11.5.4 SPFA 算法

先说重点，**不要把 SPFA 作为求解最短路径的模板**，如果你对最短路径的几种算法充分理解，能准确分析给定的场景，那可以用用。许多国外文章中都不存在 SPFA 算法，一般叫做队列优化的 Bellman-Ford 算法，也不承认 SPFA 以及它的复杂度证明，那为什么 SPFA 在以前的比赛中那么火？因为求解单源路径效率最高的就是堆优化的 Dijkstra，复杂度为 O(ElogV)，但 Dijkstra 不能处理负权边且手写堆也很麻烦，有负权边只好用 Bellman-Ford，可是 Bellman-Ford 的复杂度较高达到了 O(VE)，所以就有了 SPFA，写起来简单，又可以处理负权边（不能有负权环），还可以判断负权环，在随机稀疏图中复杂度为 O(kE)胜过 Bellman-Ford，不过现在很多题目都会卡 SPFA 算法了，因为 SPFA 效率高的前提是 k 这个常数比较小，但实际被证明，这个 k 最坏可以达到 V 的量级，比如随机网格图（在网格图中走错一次路可能导致很高的额外开销），最坏复杂度和 Bellman-Ford 算法一样，只有在随机稀疏图才能重复体现出其优势。SPFA 的实现：

1. 用 q 存放进行过松弛操作的点，初始队列只有一个起点；
2. 用一个 dist 数组表示起点到其它点的距离，初始只有起点到自己是 0，到其它是无穷大。
3. 然后开始循环，只要队列不为空就出队列，每弹出一个点就看目前的路径是否比之前的小（dist[node] + graph[node][v] < dist[v]），是的话就进行松弛，松弛完看这个点是否在队列中，不在就加入队列。
4. 直到队列为空时算法结束。

```
from collections import deque
class Solution:
    def spfa(self, graph, start):
        q = deque([start])
        dist = {i: float('inf') for i in graph}
        cnt = {i: 0 for i in graph}
        dist[start], cnt[start] = 0, 1
        while q:
            node = q.popleft()
            for v in graph[node]:
                if dist[v] > dist[node] + graph[node][v]:
                    dist[v] = dist[node] + graph[node][v]
                    if v not in q:
                        q.append(v)
                        cnt[v] += 1
                        # 判断是否有负环，避免死循环
                        if cnt[v] > len(graph): return -1
            print(q, dist)
```

```
if __name__ == '__main__':
    S = Solution()
    test = {'a': {'b': 24, 'c': 8, 'd': 15},
            'b': {'e': 6},
            'c': {'f': 3, 'e': 7},
            'd': {'g': 4},
            'e': {'g': 9},
            'f': {'d': 5, 'e': 2, 'g': 3},
            'g': {'b': 3}}
    res = S.spfa(test, 'a')
```

总结就是：**每次仅对最短路估计值发生变化的顶点的所有出边进行松弛操作**。其实就是 BFS，区别是 BFS 中一个点出了队列就不能再重新进入队列，但 SPFA 中只要队列里没这个点就可以把它加入队列，不管它之前进没进过队列。

判断有无负环：**如果某个点进入队列的次数超过 V 次则存在负环**，因为起点到终点的松弛路径的长度最多可以有 V-1 种变化，所以起点最多入队 V-1 次。SPFA 只能判断是否有负权环但无法处理带负环的图，因为负权环上的点可以一直被松弛，一直能被松弛就代表着队列会不断反复让负权环上的点入队出队，程序就会死循环。

【SPFA 算法详解（强大图解，不会都难！）】

【Dijkstra、Bellman Ford、SPFA、Floyd 算法复杂度比较】

【最短路径 | 优化 Bellman-Ford（SPFA）】

【如何看待 SPFA 算法已死这种说法？ - 知乎】

5.11.5.5 四种算法总结

概念：

1. Dijkstra 算法是典型的**单源最短路径**算法，用于计算**从一个顶点到其余各顶点的最短路径**，图中**不能存在负权边**。

2. Floyd 算法是典型的**多源最短路径算法**，用于计算 **任意两个顶点之间的最短路径。可以正确处理有向图或负权边（但不能存在负权回路）的最短路径问题**。

3. Bellman-Ford 算法是**单源最短路径**算法，还可以**判断是否存在负权环**。它和 Dijkstra 的区别是，**可以求解有负权边但无负权环的最短路劲**问题，效率较低。

4. SPFA 算法就是 Bellman-Ford 算法经过队列优化而已，是**单源最短路径**算法，还可以**判断是否存在负权环**。相比 Dijkstra 的优点是可以求解负权边

而且写起来简单，相比 Bellman-Ford 的优点是处理随机稀疏图时间复杂度低。缺点是处理网格图时间复杂度很高，不建议作为最短理解的模板。

时间复杂度：

1. Dijkstra 算法时间复杂度和 Prim 一样，用邻接矩阵时间和空间复杂度为 O(V^2)，用**邻接表+堆优化**可以把时间复杂度降低至 O（ElogV），时间复杂度为 O（V + E）。

2. Floyd 算法时间复杂度是 O(V^3)，空间复杂度为 O(V^2)。一般绑定邻接矩阵，因为要把每个顶点当做中间点计算一次，就算给定邻接表也会转换成邻接矩阵，这样方便计算。

3. Bellman-Ford 算法时间复杂度事 O(VE)，V 代表顶点数，E 代表边数。

4. SPFA 算法时间复杂度是 O(kE)，稀疏图中 k 是个很小的常数，最坏是 O(VE)，所以并没有提升 Bellman-Ford 算法最坏情况下的复杂度。

适用情况

1. Dijkstra 算法适用于**无负权边**的**单源最短路径**问题。因为 Dijkstra 是贪心算法，贪心算法成功的前提是局部最优解即全局最优解，但**存在负权的话不能保证局部最优解就是全局最优解**。

2. Floyd 算法适用于**有负权边（但不能存在负权环）**的**多源最短路径**问题。因为 Floyd 用了动态规划的思想，有点类似暴力求解。不能存在负权环是因为，负权环随着循环会无限变小。

3. Bellman-Ford 算法适用于**有负权边（但不能存在负权环）**的**单源最短路径**问题。无向图实际上是双向的，只要有负权边就有负权环（从负权边出去再沿着负权边原路进来），所以 Floyd 实际上是针对**无负权边的无向图**，和**无负权环的有向图**。

4. SPFA 算法适用于**稀疏图**中**有负权边（但不能存在负权环）**的**单源最短路径**问题。无向图实际上是双向的

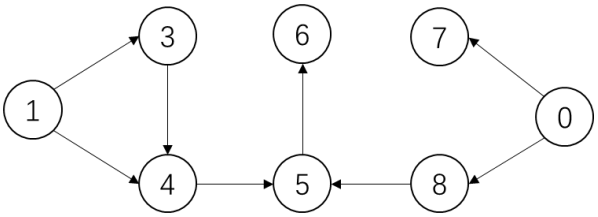
5.11.6 拓扑排序

AOV-网：**AOV 就是用顶点表示活动，弧表示活动之间存在的某种制约关系**。在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系的有向图称为“顶点表示活动的网”，简称 AOV-网（Activity On Vertex Network）。

拓扑排序（Topological sorting）：由一个**有向无环图**（Directed Acyclic Graph， DAG）的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：

1. **每个顶点出现且只出现一次；**
2. **若 A 在序列中排在 B 的前面，则在图中不存在从 B 到 A 的路径。**

每个有向无环图都至少存在一种拓扑排序。拓扑排序主要用来解决有向图中的依赖解析（Dependency Resolution）问题，举例来说，如果我们将一系列需要运行的任务构成一个有向图，图中的有向边则代表某一任务必须在另一个任务之前完成这一限制。那么运用拓扑排序，我们就能得到满足执行顺序限制条件的一系列任务所需执行的**先后顺序**。当然也有可能图中并不存在这样一个拓扑顺序，这时候我们无法根据给定要求完成这一系列任务，这种情况称为循环依赖（circular dependency）。



如上图，我们可以直观的看出节点的先后顺序，但是一般题目中给出的列表不是直接的邻接表，比如[3, 1]意思是想要访问 3 必须先访问 1，给定的是[[3, 1], [4, 1] ,[4, 1], [5, 4], [6, 5], [5, 8], [8, 0], [7, 0]]，所以我们转换字典和求入度的时候要注意。算法步骤如下：

1. 首先我们要把列表转换成字典 adja，然后计算出度表 inde。
2. 将所有入度为 0 的节点入栈。入度为 0 的点就是起点。
3. 当 stack 非空时，依次把 node 出栈，在图中删除节点 node，并不是真正从邻接表中删除此节点，主要是为了将此节点对应所有邻接节点的入度减一，当邻接节点的入度减一后为 0，说明这个邻接节点的所有前置节点已被访问，它已经可以访问了，此时将它入栈。
4. 若课程安排图中存在环，一定有节点的入度始终不为 0。此时得到的是其中符合拓扑排序的一部分节点。

每个顶点进一次栈，出一次栈，所需时间 O(V)。每个顶点入度减 1 的运算共执行了 E 次。所以总的时间复杂为 O(V+E)。

```
class Solution:
    def topologica(self, graph):
        adja, tp = {}, []
        # 把数组转换成字典格式，虽然稍微复杂但通用。题目中数组是逆序邻接表，
        # 所以我们做如下转换后，字典中 key 是 value 的前置节点。
        for i, j in graph:
            if i not in adja: adja[i] = set()
            if j not in adja: adja[j] = {i}
```

```
            else: adja[j].add(i)
        # 初始化入度列表，graph 中下标 0 是终点，下标 1 才是起点。
        inde = {v: 0 for v in adja}
        for i, j in graph: inde[i] += 1
        # 初始化一个栈存放入度为 0 的点
        stack = [v for v in inde if inde[v] == 0]
        print(adja, inde, stack)
        while stack:
            node = stack.pop()
            tp.append(node)
            for v in adja[node]:
                inde[v] -= 1
                # 入度为 0 时表示没有前置节点，入栈
                if inde[v] == 0: stack.append(v)
        print(tp)

if __name__ == '__main__':
    S = Solution()
    test = [[3, 1], [4, 1], [5, 4], [6, 5], [5, 8], [8, 0], [7, 0]]
    res = S.topologica(test)
```

https://leetcode-cn.com/problems/course-schedule/solution/course-schedule-tuo-bu-pai-xu-bfsdfsliang-chong-fa/

如果给定了顶点数，而且顶点是从 0 开始的连续数字，那转换字典，初始化入度列表就很简单了，207 题就是这样：

```
inde = [0 for _ in range(n)]
adja = {[] for i in range(n)}
# 初始化字典和入度表
for cur, pre in graph:
    inde[cur] += 1
    adja[pre].append(cur)
# 初始化入读为 0 的列表
stack = [i for i in range(n) if inde[i] == 0]
print(adja, inde, stack)
```

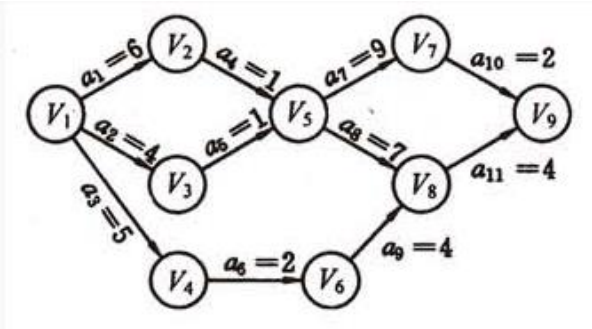
5.11.7 关键路径

拓扑排序是解决一个工程能否顺序进行的问题，但有时还需解决工程完成需要的最短时间。与 AOV-网对应的是 AOE-网（Activity On Edge Network），即“边表示活动的网”，**AOE-网是一个带权的有向无环图**，用顶点表示事件，弧表示活动，权值表示活动的持续时间 。通常 **AOE-网可用来估算工程完成的**

时间，AOV-网用来描述活动之间的制约关系，和 AOV-网不同，对 AOE-网有待研究的问题是：

1. 完成整项工程至少需要多少时间？
2. 哪些活动是影响工程进度的关键？

如下图有 9 个事件 11 项活动，v1 表示整个工程开始，v9 表示整个工程结束。只有活动 a4 和 a5 完成后事件 v5 才能发生，事件 v5 发生后活动 a7 和 a8 才可以开始。正常情况下网中只有一个**源点**（入度为零的点）和一个**汇点**（出度为零的点）。



路径上各个活动所持续的时间之和称为路径长度，从**源点到汇点具有最大长度的路径就叫关键路径**（Critical Path），为什么是长度最大呢？只有最慢的活动完成后，整个工期才算完成，所以只有缩短关键路径上的活动才能缩短整个工程时间，一个工程可能有多条关键路径。图中关键路径是 (v1.v2.v5,v8,v9)，路径长度是 18。先看几个定义：

1. **事件最早开始时间 ve(i)**：假设 v1 是起点，从 v1 到 vi 的长度就叫**事件** vi 的最早开始时间，这个时间决定了所有以 vi 为尾的弧所表示的**活动**的最早开始时间，图中事件 v6 的最早开始时间是 7，这也是以它为尾的活动（即 a9 ）的最早开始时间。
2. **事件最迟开始时间 vl(i)**：事件 vi 最迟开始的时间，也是以它为尾的**活动**的最早开始时间，超出则会延误整个工期。
3. **活动最早开始时间 e(i)**：活动 ai 最早开始的时间。图中活动 a6 最早开始时间是 5。
4. **活动最迟开始时间 l(i)**：活动 ai 最迟开始的时间。在不推迟整个工程完成的前提下，活动最迟必须开始的时间，图中 a6 最迟开始时间是 18 - 4 - 4 - 2 = 8，也就是 a6 推迟 3 天开始或者延迟 3 天完成都不会导致整个工程延期。
5. **活动持续时间 dut(<j,k>)**：如果活动 ai 由弧 <j,k> 表示，则其持续时间记为 dut(<j,k>)。

找关键路径的过程就是辨别哪些是关键活动的过程，辨别关键活动就是要找 e(i) = l(i) 的活动，也就是我们要找的这个活动，它的最早开始时间和最迟开始时间是相同的。如果活动 ai 由弧 <j,k> 表示，则有如下关系：

1. e(i) = ve(j)，即活动 ai 的最早开始时间和事件 vj 的最早开始时间相同，如上图 a6 的最早开始时间和事件 v4 的最早开始时间相同。

2. l(i) = vl(k) - dut(<j, k>)，即活动 ai 的最迟开始时间 = 事件 vk 的最迟开始时间 - 活动 ai 的持续时间（弧 <j,k> 的长度），如上图中活动 a6 的最迟开始时间就是事件 v6 的最迟开始时间减去 a6 的持续时间。

一个**活动的最早和最迟开始时间**，其实就是这个**活动起始事件的最早和最迟开始时间**，为了求得 AOE-网中活动 ai 的 e(i)（活动最早开始时间）和 l(i)（活动最迟开始时间），首先应求得事件 vj 的 ve(j)（事件最早开始事件）和 vl(j)（事件最迟开始时间），**ve(j) 可以利用拓扑排序求得，vl(j) 可以利用拓扑逆序求得**。

```
class Solution:
    def topologica(self, graph):
        adja, tp = {}, []
        # 把数组转换成字典格式，和拓扑排序中稍有不同。
        for i, j, e in graph:
            if not adja.get(j): adja[j] = {}
            if adja.get(i): adja[i][j] = e
            else: adja[i] = {j: e}
        # 初始化入度列表
        inde = {v: 0 for v in adja}
        ve = {v: 0 for v in adja}
        for i, j, e in graph:
            inde[j] += 1
        # 初始化一个栈存放入度为 0 的点
        stack = [v for v in inde if not inde[v]]
        print(adja, inde, stack)
        while stack:
            node = stack.pop()
            tp.append(node)
            for v in adja[node]:
                inde[v] -= 1
                # 入度为 0 时表示没有前置节点，入栈
                if inde[v] == 0: stack.append(v)
                # 计算事件最早开始时间，
                if ve[v] < ve[node] + adja[node][v]:
                    ve[v] = ve[node] + adja[node][v]
        # 返回字典，拓扑排序的顶点，事件最早开始时间
        return adja, tp, ve

    def criticalpath(self, graph):
        dut, tp, ve = self.topologica(graph)
        cpath = []
```

```
        # 初始化事件最迟开始时间，
        vl = {v: ve[tp[-1]] for v in dut}
        # 关键路径的起点是确定的，所以不需要处理
        for node in tp[:0:-1]:
            for v in dut[node]:
                # 事件最迟开始时间 - 活动的持续时间 = 活动最迟开始时间，
                # 需要从汇点往前计算，所以用到拓扑逆序
                if vl[node] > vl[v] - dut[node][v]:
                    vl[node] = vl[v] - dut[node][v]
                # 判断关键活动：活动最早开始时间 = 活动最迟开始时间，
                # 而事件最早开始时间，就是以该事件为起点的活动的最早开始时间
                if ve[node] == vl[node]:
                    cpath.append((node, v, vl[node]))
        # 把起点加入 cpath，
        cpath.append((tp[0], cpath[-1][0]))
        return cpath

if __name__ == '__main__':
    S = Solution()
    test = [['v1', 'v2', 6], ['v1', 'v3', 4], ['v1', 'v4', 5],
            ['v2', 'v5', 1], ['v3', 'v5', 1], ['v4', 'v6', 2],
            ['v5', 'v7', 9], ['v5', 'v8', 7], ['v6', 'v8', 4],
            ['v7', 'v9', 2], ['v8', 'v9', 4]]
    res = S.criticalpath(test)
    print(res)
```

5.11.8 并查集

并查集（Disjoint Sets）是一种树型的数据结构，用于处理一些不交集的合并及查询问题。有一个 Union-Find 算法（union-find algorithm）定义了两个用于此数据结构的操作：

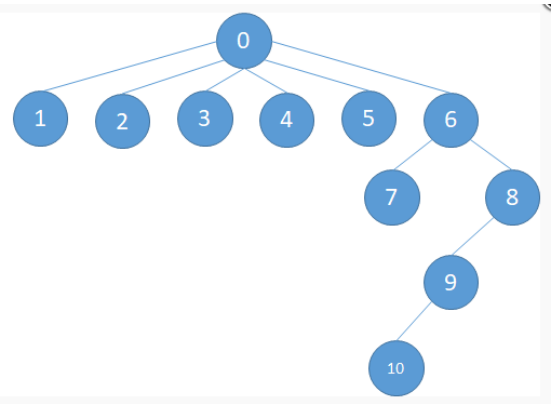
1. MAKE-SET：初始化构建树，一个点的根节点就是自己本身。
2. Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。下面代码中我们定义为 root()。
3. Union：将两个子集合并成同一个集合。下面代码中我们定义为 merge_node()。

并查集主要就两步，第一步查找两个元素是否属于一个集合（在图中的表现即，它们是否有共同的根节点），第二步如果它们不属于一个集合，就把它们合并成到同一集合（在图中的表现即，让一个元素的根节点成为另一个元素根节点的子节点）。

5.11.8.1 基础版

定义一个列表 parent（也可以是字典），下标是顶点，值是该顶点的根节点，比如 parent = [0, 0, 0, 2] 代表 0, 1, 2 的根节点是 0，3 的根节点是 2，find(self, x, parent) 用于查询 x 的根节点，union(self, x, y, parent) 用于合并 x 和 y（x 和 y 本身已经是根节点）。最终一般会**对 union 做基于 rank 或 size 的优化，对 find 做路径压缩优化**，后面细说。

```
class Solution:
def grouping(self, edges, nums):
        # 注意这里 parents 的初始化用了 range，只适用于给定点是连续的数字。
        parent = [i for i in range(nums)]
        # size = [1 for _ in range(nums)] # 基于 size 的优化
        # rank = [1 for _ in range(nums)] # 基于 rank 的优化
        for v1, v2 in edges:
            v1_root = self.find(v1, parent)
            v2_root = self.find(v2, parent)
            if v1_root != v2_root:
                self.union(v1_root, v2_root, parent)
        print(parent)

    def find(self, x, parent):
        while x != parent[x]:
            x = parent[x]
        return x

    def union(self, x, y, parent):
        parent[y] = x

if __name__ == '__main__':
    S = Solution()
    nums = 11
    test = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5],
            [6, 7], [9, 10], [8, 10], [7, 9], [2, 8]]
    S.grouping(test, 11)
    # parent: [0, 0, 0, 0, 0, 0, 0, 6, 6, 8, 9]
```



5.11.8.2 基于 Union 的优化

基于 size 的优化：

树的结构直接影响 find 效率，线性的树就退化成了链表结构，查询复杂度是 O(n)，所以需要优化。我们可以引入一个列表 size，专门用来记录每一个集合都有多少个元素，然后在进行合并时候，先查询 size 中两个集合的元素个数，把元素个数小的集合并入元素个数大的集合中。这样，就能大大的减少因为合并造成树的层数过高的现象，提高 find 效率。 size 初始化每个元素各自为一个集合，个数均为 1 ，代码如下：

```
def union_size(self, x, y, parent, size):
    if size[x] < size[y]:
        parent[x] = y
        size[y] += 1
    else:
        parent[y] = x
        size[x] += 1
# 得到 parent: [0, 0, 0, 0, 0, 0, 9, 6, 9, 0, 9]
```

基于 size 的优化，先判断两个集合元素的数量，**元素数量小的集合并入元素数量大的集合**，从而降低了树高，使得 find 操作的效率更高。但这中优化似乎也有局限性，如果一个集合 A 元素很多却只有两层，而另一个集合 B 元素很少却有三层，这时候明显把 A 并到 B 更优，总层数就是 B 的层数，但是把 B 并入 A 的话总层数会比 B 的层数还高 1 。

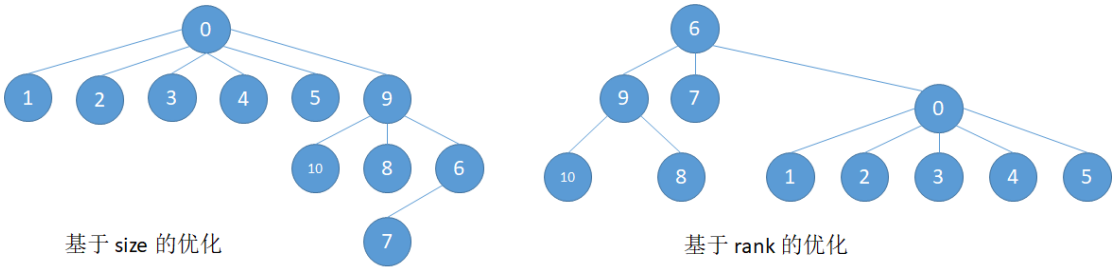
在集合中，层数越少，对于每一个节点平均来说，找到根节点所需要查找的次数就会越小，所以接下来我们看看对层数的优化。

基于 rank 的优化：

我们可以引入一个列表 rank，专门用来记录两个集合的层数，然后在进行合并时候，先查询 rank 中两个集合的层数，**层数小的集合并入层数大的集合中**。这样，就能更准确的降低合并过程中树的层数，提高 find 效率。 rank 初始每个元素各自是一个集合，层数均为 1 ，代码如下：

```
def union_rank(self, x, y, parent, rank):
    if rank[x] < rank[y]:
        parent[x] = y
    elif rank[x] > rank[y]:
        parent[y] = x
    else:
        parent[y] = x
        rank[x] += 1
# 得到 parent: [6, 0, 0, 0, 0, 0, 6, 6, 9, 6, 9]
```

合并后集合层数唯一会变的情况，就是两个集合层数一模一样的时候。层数相同时，谁是谁的根节点都无所谓了。这两种优化方式其在非极端情况下其实差不多，不过基于 rank 的优化更能应对极端情况。结果如下图右。



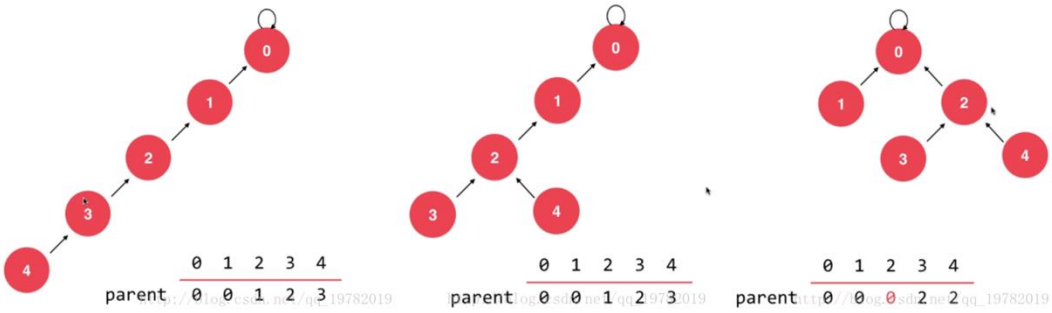
5.11.8.3 基于 Find 的优化

上面都是对合并过程的优化，现在我们对查询过程进行优化，也就是**路劲压缩优化**。可不可以在 find 的同时，做一些操作使得树的层数尽量变得更少呢？

对于一个集合树来说，它的根节点下面可以依附着许多的节点，因此，我们可以尝试在 find 的过程中，从底向上，如果此时访问的节点不是根节点的话，那么我们可以把这个节点尽量的往上挪一挪，减少集合的层数，这个过程就叫做路径压缩。

基于循环的路径压缩优化：

假设我们起始的并查集如下图所示，现在我们要 find[4] ，首先我们根据 parent[4] 可以得出 4 并不是一个根节点，因此我们可以在向上继续查询之前，先把这个节点往上面挪一挪（路径压缩），我们让节点 4 连接到其父节点的父节点上，让节点 2 作为其新的父亲节点，如果节点 2 正好是根节点，那么 2 的父节点指向自己，所以这种变动并不会因为越界而出错。



在进行 find 操作的同时，我们不仅把需要查找的根节点给找到了，与此同时我们还对树进行了压缩操作，这便是路径压缩的意思。通过路径压缩，我们在下一次执行 find 操作的时候，层数变得尽可能少了，那么效率将会大大的提高。

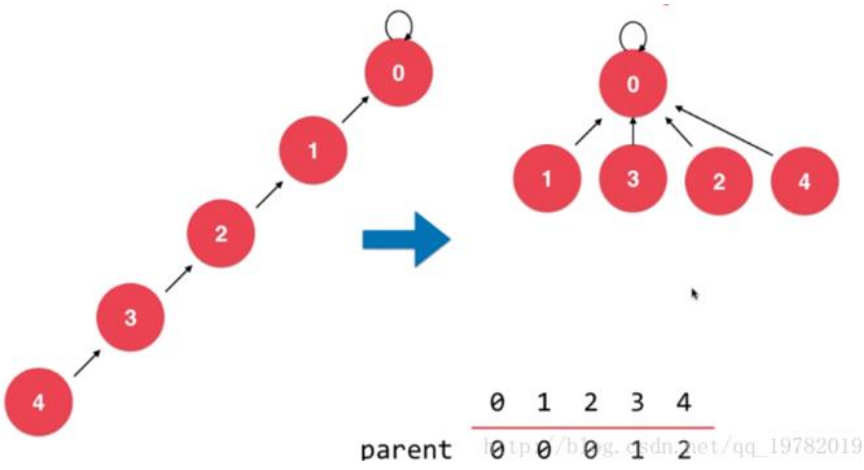
代码如下：

```
def find_iteration (self, x, parent):
    while x != parent[x]:
        parent[x] = parent[parent[x]]
        x = parent[x]
    return x
```

查询节点时候，顺便让该节点指向了其父节点的父节点，也就是大于等于 3 层的结构中才能体现出其优势。其实这种路径压缩方法并没有把层数压缩到最小，最优情况下应该压缩到两层，所有的节点都指向根节点，这就需要使用基于递归的路径压缩。

基于递归的路径压缩优化：

最优情况下应该压缩到两层，如下图：



因为 find 操作返回的是集合的根节点，因此我们只需要集合中所有的非根节点的父亲指针都指向这个根节点就好了，我们完全可以用递归的方法去实现，递归是典型的用空间换时间的方法。

代码如下：

```
def find_recursion(self, x, parent):
    if x == parent[x]:
        return x
    else:
        parent[x] = self.find_recursion(parent[x], parent)
        return parent[x]
```

用一开始的测试用例，这两种情况效果一样，我们需要用图中结构的测试用例才行，代码逻辑很好理解，不再测试。

标准并查集的时间复杂度是 O(n^2)。经过 size 或者 rank 优化后的时间复杂度是 O(mlog n)，因为将 m 个操作（Union 或 Find）应用于 n 个元素。同时使用路径压缩和基于秩（rank）合并的优化后，时间复杂度是 O(α(n))，α 是阿克曼函数的某个反函数，这个 n 十分巨大时候也还是小于 5。

优化版，union 基于 rank 优化，find 用路径压缩优化：

```
class Solution:
    def grouping(self, edges, nums):
        parent = [i for i in range(nums)]
        rank = [1 for _ in range(nums)]
        for v1, v2 in edges:
            v1_root = self.find_pcompression(v1, parent)
            v2_root = self.find_pcompression(v2, parent)
            if v1_root != v2_root:
                self.union_rank(v1_root, v2_root, parent, rank)
        print(parent)

 # 路径压缩优化 Path compression
    def find_pcompression (self, x, parent):
        while x != parent[x]:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x
    # 基于 rank 的优化
    def union_rank(self, x, y, parent, rank):
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
```

```
        else:
            parent[y] = x
            rank[x] += 1
        print("合并节点:", parent)

if __name__ == '__main__':
    S = Solution()
    nums = 11
    test = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5],
            [6, 7], [9, 10], [8, 10], [7, 9], [2, 8]]
    S.grouping(test, 11)
```

5.12 动态规划

记住，要符合「最优子结构」，子问题间必须互相独立。啥叫相互独立？你肯定不想看数学证明，我用一个直观的例子来讲解。

比如说，你的原问题是考出最高的总成绩，那么你的子问题就是要把语文考到最高，数学考到最高..... 为了每门课考到最高，你要把每门课相应的选择题分数拿到最高，填空题分数拿到最高..... 当然，最终就是你每门课都是满分，这就是最高的总成绩。

得到了正确的结果：最高的总成绩就是总分。因为这个过程符合最优子结构，“每门科目考到最高”这些子问题是互相独立，互不干扰的。

但是，如果加一个条件：你的语文成绩和数学成绩会互相制约，此消彼长。这样的话，显然你能考到的最高总成绩就达不到总分了，按刚才那个思路就会得到错误的结果。因为子问题并不独立，语文数学成绩无法同时最优，所以最优子结构被破坏。

回到凑零钱问题，显然子问题之间没有相互制约，而是互相独立的。所以这个状态转移方程是可以得到正确答案的。

第六章 思考类题目

6.1 两人取球，如何保证甲取到最后一个球

有 100 个球，甲乙两个人轮流取，每人每次能取 1 到 5 个球，如果甲先取，怎样才能保证甲取到第 100 个球？

如果现在轮到乙取，还剩下 6 个球，因为只能取 1 到 5 个，那么无论乙怎么取，接下来都能保证甲取到第 100 个。那此前的甲要怎么取才能剩下 6 个球呢？为什么要剩 6 个呢？6 有什么特别吗？

其实这道题是考倍数的问题。每次最多取 5 个最少 1 个，只要保证剩下的个数是 6 的倍数即可，100÷6=16 余 4，甲先取 4 个，然后不管乙取几个，都和他凑够 6 个就行，这样剩下的始终是 6 的倍数，最后一定剩 6 个，轮乙取。

推广一下，也可以是每次取 3 到 5 个，那就维持 8 的倍数。

https://blog.csdn.net/less_cold/article/details/78298660

6.2 有 25 匹马 5 个跑道，最少多少次能比出前 3

有 25 匹马，速度都不同，但每匹马的速度都是定值。现在只有 5 条赛道，无法计时，即每赛一场最多只能知道 5 匹马的相对快慢。问最少赛几场可以找出 25 匹马中速度最快的前 3 名？找出前五名呢？

求前 3 名最少要 7 次：

1. 前 5 次，将马分成 A、B、C、D、E 五组,各组分别进行比赛，决出各组名次，各组排名依次如下：
   第一组：A1、A2、A3、A4、A5，
   第二组：B1、B2、B3、B4、B5，
   第三组：C1、C2、C3、C4、C5，
   第四组：D1、D2、D3、D4、D5，
   第五组：E1、E2、E3、E4、E5。

2. 第 6 次比赛，各组第一名进行一次比赛，决出名次，排名如下：
   A1、B1、C1、D1、E1。

第四组和第五组是没机会进前三了，因为他们的第一名都没进前三，现在前三名的候选者是：

第一组：A1、A2、A3，
第二组：B1、B2，
第三组：C1。

即，可能的前三名是：
A1、A2、A3，
A1、A2、B1，
A1、B1、B2，
A1、B1、C1，

3. 第 7 次比赛，因为 A1 是第一名已经确定了，所以这五个再比一次：
   A2、A3、B1、B2、C1，比赛求出第 2，第 3 名即可。

如果求前 5 名最少要 8 次：

1. 前六次比赛，同上，将马分成 A、B、C、D、E 五组，假如第 6 次比赛结果是：A1、B1、C1、D1、E1，确定了 A1。现在前五名的候选者是：
   第一组：A1、A2、A3、A4、A5，
   第二组：B1、B2、B3、B4，

第三组：C1、C2、C3，
第四组：D1、D2，
第五组：E1

2. 第 7 次比赛，将上一次**第一名所在组的下一号**和**其他四个第一名**比，即 A2、B1、C1、D1、E1 比一次，第一名已经确定，只剩四个名额，所以垫底的那一名以及他后面的马都会被淘汰，假如结果是：A2、B1、C1、D1、E1，那 E 组会被排除，剩下的候选者是：
   第一组：A1、A2、A3、A4、A5，
   第二组：B1、B2、B3，
   第三组：C1、C2，
   第四组：D1，

3. 第 8 次比赛，将上次**第一名所在组的下一号，第二名所在组的下一号，以及剩余的三个**比，即 A3、B2、C1、C2、D1 比一次，：
   a. 假如结果是：C1、C2、D1、B2、A3，因为 B1 比 C1 快，所以此时可以直接确定前五名为：A1、A2、B1、C1、C2。最好的情况需 8 次。

   b. 假如结果是：A3、B2、C1、C2、D1，已知 B1 比 B2 快，但 B1 和 A3 不确定，所以需要再比一次 A3、A4、A5、B1、B2，最坏 9 次一定能出结果。

https://blog.csdn.net/ldw662523/article/details/79567685

这类型题目有个特点，不管选手和跑道多少，只要求的前几名越少，就越简单，比如 64 匹马 8 个跑道找前四，那还是分 8 组各跑一次，第九次让这 8 组的第一名比，这次下来，只有前四名所在的组能继续比赛，**要找前几名，就留前几组**，而且依次第一名所在的组有 3 名候选者（第一名已确定），第二名所在的组有 3 名候选者，第三名所在的组有 2 名候选者，第一名所在的组有 1 名候选者，也就是有 9 个候选者，那第十次就是其中 8 人跑一次，第十一次就是前三名和剩下的那一个跑一次，总共需要 11 次。

https://blog.csdn.net/ZYDX18984003806/article/details/100103658

6.3 位运算实现正整数加法

注意是正整数，加法主要的操作是“加”和“进位”，要用位运算代替加法，就要找到位运算跟“加”、“进位”有什么关联。

异或运算：相同为 0，不同为 1。与加法的比较如下：

```
0 ^ 0 = 0       0 + 0 = 0
1 ^ 0 = 1       1 + 0 = 1
0 ^ 1 = 1       0 + 1 = 1
1 ^ 1 = 0       1 + 1 = 0
```

与运算：都为 1, 才为 1，否则为 0。与乘法的比较如下：

```
0 & 0 = 0       0 & 0 = 不进位
1 & 0 = 0       1 & 0 = 不进位
0 & 1 = 0       0 & 1 = 不进位
1 & 1 = 1       1 & 1 = 进位
```

从上面我们可以发现规律：

在不考虑进位的情况时，相加正好是个异或运算，x + y = x ^ y。因为异或是同一个位置上数字不一样时（1 ^ 0 或者 0 ^ 1）为 1，一样时（0 ^ 0 或者 1 ^ 1）为 0，这就是无视进位的加法。

进位正好可以用与运算表示，(x & y)，因为与运算是同一位置上数字都为 1 时（1 & 1）为 1，否则（1 & 0 或者 0 & 1 或者 0 & 0）为 0，这就是二进制进位的规则，但是进位的数字是要加到下一位上，所以需要左移。

得到加法到位运算的对应公式：**x + y = (x^y) + ((x&y)<<1)**

不过还没有完，因为 (x^y) + ((x&y)<<1) 本身就是两个数相加，它们有没有产生新的进位无法确定，所以需要一个循环。

例如 5 + 7：

```
5 = 0 1 0 1
7 = 0 1 1 1

x ^ y = 0 0 1 0       = c
x & y = 0 1 0 1
(x&y)<<1 = 1 0 1 0    = d

c ^ d = 1 0 0 0       = e
```

c & d = 0 0 1 0
(c&d)<<1 = 0 1 0 0 = f

e ^ f = 1 1 0 0
e & f = 0 0 0 0 为 0，不再有进位，结束
(e&f)<<1 = 0 0 0 0

通过与运算可以知道第二和第四位产生了进位，它们需要加到第一位和第三位（左边高位右边低位，溢出的值要进位到高位），所以左移，接下来应该执行 (x^y) + ((x&y)<<1)，但是这个过程有没有产生进位呢？所以还要判断 c & d 是否为 0，不为 0 的话说明还有进位，需要继续进行。

代码如下：

```python
# 位运算实现正整数
def sum(a, b):
    if b == 0:
        return a
    return sum(a^b, (a&b)<<1)
```

【python 使用位运算实现整数加法运算】

第七章 编译原理

编译原理涉及内容很多，也很复杂，本章涉及的主要是编译器和解释器相关的内容。

7.1 编译过程

计算机能执行的是二进制机器指令，我们写的源代码无论高级语言还是低级语言，计算机都无法直接执行，这就需要把源码翻译成机器码再执行，这个翻译执行的过程，有两种方法可以选择，编译器编译 + 操作系统执行，或者解释器直接解释并执行。

高级语言从开始到执行，基本都会经历预处理 → 词法分析（产生单词流）→ 语法分析（产生语法树）→ 代码翻译（产生中间代码）→ 执行（如果中间代码是字节码，一般由解释器执行，如果中间代码是可执行的机器码，一般由操作系统直接执行），不同的代码可能还会进行不同的代码优化。当然低级语言如汇编和二进制机器码是不需要这些步骤的。

以 C 语言为例，我们用 gcc 的命令 "gcc main.c - o main_direc" 将 main.c 源文件编译生成可执行文件 main_direct，但这实际上经历了四个步骤，分别是**预处理（Prepressing）、编译（Compilation）、汇编（Assembly）和链接（Linking）**：

1. 预处理，生成预编译文件（.i 文件）：gcc - E main.c - o main.i
2. 编译，生成汇编代码（.s 文件）：gcc - S main.i - o main.s
3. 汇编，生成目标文件（.o 文件）：gcc - c main.s - o main.o
4. 链接，生成可执行文件（executable 文件）：gcc main.o - o main

7.1.1 预处理（Prepressing）

预处理部分由 CPP 预处理器（C Pre-Processor）处理，是一个与 C 编译器独立的小程序。预处理又叫预编译，预处理阶段将根据已放置在文件中的预处理指令来修改源文件的内容，预编译器并不理解 C 语言语法，它仅是在程序源文件被编译之前，实现文本替换的功能。简单来说，**预处理就是将源代码中的预处理指令根据语义预先处理，并且进行一下清理、标记工作，然后将这些代码输出到一个 .i 文件**中等待进一步操作。

一般地，C/C++ 程序的源代码中包含以 # 开头的各种编译指令，被称为预处理指令，其不属于 C/C++ 语言的语法，但在一定意义上可以说预处理扩展了 C/C++。根据 ANSI C 定义，预处理指令主要包括：文件包含、宏定义、条件编译和特殊控制等 4 大类。

预处理阶段主要做以下几个方面的工作：

1. 文件包含：#include 是 C 程序设计中最常用的预处理指令，格式有尖括号 #include <xxx.h> 和双引号 #include "xxx.h" 之分，分别表示从系统目录下查找和优先在当前目录查找，例如常用的 #include <stdio.h> 指令，就表示使用 stdio.h 文件中的全部内容，替换该行指令。

2. 添加行号和文件名标识： 比如在文件 main.i 中就有类似 # 2 "main.c" 2 的内容，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号。

3. 宏定义展开及处理： 预处理阶段会将使用 #define A 100 定义的常量符号进行等价替换，文中所有的宏定义符号 A 都会被替换成 100，还会将一些内置的宏展开，比如用于显示文件全路径的__FILE__，另外还可以使用 #undef 删除已经存在的宏，比如 #undef A 就是删除之前定义的宏符号 A。

4. 条件编译处理: 如 #ifdef、#ifndef、#else、#elif、#endif 等，这些条件编译指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理，将那些不必要的代码过滤掉，防止文件重复包含等。

5. 清理注释内容： // xxx 和 /*xxx*/ 所产生的的注释内容在预处理阶段都会被删除，因为这些注释对于编写程序的人来说是用来记录和梳理逻辑代码的，但是对编译程序来说几乎没有任何用处，所以会被删除，观察 main.i 文件也会发现之前的注释都被删掉了。

6. 特殊控制处理: 保留编译器需要使用 #pragma 编译器指令，另外还有用于输出指定的错误信息，通常来调试程序的 #error 指令。

7.1.2 编译（Compilation）

编译部分由编译器 cc 或者 ccl 处理。编译过程是整个程序构建的核心部分，也是最复杂的部分之一，其工作就是**把预处理完生成的 .i 文件进行一系列的词法分析、语法分析、语义分析以及优化后产生相应的汇编代码文件，也就是 .s 文件**。汇编代码和机器码是一一对应的，但汇编代码相比机器码可读性更好，方便调试和优化。

编译主要做以下几方面工作：

1. 词法分析（Lexical analysis）或扫描（Scanning）：主要是使用基于有线状态机的 Scanner 分析出 token（标记符），这是为了能准确理解并处理字符串，token 具有类型和值属性。可以通过一个叫做 lex 的可执行程序来完成词法扫描，按照描述好的词法规则将预处理后的源代码分割成一个个记号，同时完成将标识符存放到符号表中，将数字、字符串常量存放到文字表等工作，以备后面的步骤使用。**扫描器的输入一般是文本，经过词法分析，输出是将文本切割为单词的流**。

2. 语法分析（Syntax analysis）或解析（Parsing）：对有词法分析产生的 token 采用上下文无关文法进行分析，从而产生语法树，此过程可以通过一个叫做 yacc 的可执行程序完成，它可以根据用户给定的语法规则对输入的记号序列进行解析，从而构建一棵语法树，如果在解析过程中出现了表达式不合法，比如括号不匹配，表达式中缺少操作符、操作数等情况，编译器就会报出语法分析阶段的错误。**输入是单词的流，经过语法分析，输出是语法树（syntax tree）或抽象语法树（abstract syntax tree，AST）**。

3. 语义分析(semantic analyzer)：此过程由语义分析器完成，**语义分析的一个重要部分是类型检查**，编译器 cc 所能分析的语义都是静态语义，是指在编译期间可以确定的语义，通常包括声明和类型的匹配，类型的转换等。比如将一个浮点型的表达式赋值给一个整型的表达式时，语义分析程序会发现这个类型不匹配，编译器将会报错。而动态语义一般指在运行期出现的语义相关问题，比如将 0 作为除数是一个运行期语义错误。语义分析过程会将所有表达式标注类型，对于需要隐式转换的语法添加转换节点，同时对符号表里的符号类型做相应的更新。

4. 代码优化： 此过程会**通过源代码优化器会在源代码级别进行优化**，针对于编译期间就可以确定的表达式（例如：100+1）给出确定的值，以达到优化的目的，此外还包括根据机器硬件执行指令的特点对指令进行一些调整使目标代码比较短，执行效率更高等操作。

7.1.3 汇编（Assembly）

汇编部分由汇编器 as 处理。**汇编就是将上一步产生的汇编代码转变成机器可识别的指令，最终生成.obj 或者.o 的目标文件**。这一步比较简单，汇编和机器指令是一一对应的，只需根据汇编代码语法和机器指令的对照表翻译过来就可以了。

目标文件中所存放的也就是与源程序等效的机器语言代码，通常至少包含代码段和数据段两个段，还要包含未解决符号表，导出符号表和地址重定向表等 3 个表。汇编过程会将 extern 声明的变量置入未解决符号表，将 static 声

明的全局变量不置入未解决符号表，也不置入导出符号表，无法被其他目标文件使用，然后将普通变量及函数置入导出符号表，供其他目标文件使用：

1. 代码段： 包含主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

2. 数据段： 主要存放程序中要用到的各种全局变量或静态的数据，一般数据段都是可读，可写，可执行的。

3. 未解决符号表： 列出了在本目标文件里有引用但不存在定义的符号及其出现的地址。

4. 导出符号表： 列出了本目标文件里具有定义，并且可以提供给其他目标文件使用的符号及其在出现的地址。

5. 地址重定向表： 列出了本目标文件里所有对自身地址的引用记录。

7.1.4 链接（Linking）

为什么需要链接？上一步产生的目标文件已经是机器码了，那可以执行吗？还不行，因为现在只是将我们自己写的代码变成了二进制形式，它还需要和系统组件（比如标准库、动态链接库等）结合起来才能运行，这些组件都是程序运行所必须的，链接就是做这件事的。

链接部分由连接器 ld 来处理。链接过程是程序构建过程的最后一步，可以简单的理解为将目标文件和库文件打包组装成可执行文件（.exe）的过程，其主要内容就是把各个模块之间相互引用的部分都处理好，将一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得各个模块之间能够正确的衔接，成为一个能够被操作系统装入执行的统一整体。

比如源代码 main.c 中的 printf 这个函数名就无法解析，需要链接过程与对应的库文件对接，完成的重定位，将函数符号对应的地址替换成正确的地址。前面提到的库文件其实就是一组目标文件的包，它们是一些最常用的代码编译成目标文件后打成的包。比如 printf 的头文件是 stdio.h，而它的实现代码是放在动态库 libc.so.6 中的，链接的时候就要引用到这个库文件。

从原理上讲，连接的的工作就是把一些指令对其他符号地址的引用加以修正，主要包括了地址和空间分配、符号决议和重定位等步骤，根据开发人员指定的链接库函数的方式不同，链接过程可分为静态链接和动态链接两种，链接静态的库，需要拷贝到一起，链接动态的库需要登记一下库的信息。

1. 静态链接： 函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时，代码将被装入到该进程的虚拟地址空间中，静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码，最终生成的可执行文件较大。

2. 动态链接： 函数的代码被放到动态链接库或共享对象的某个目标文件中。链接处理时只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在这样该程序在被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间，根据可执行程序中记录的信息找到相应的函数代码。这种连接方法能节约一定的内存，也可以减小生成的可执行文件体积。

7.1.5 总结

gcc 编译器的工作过程：源文件 --> 预处理 --> 编译 --> 汇编 --> 链接 --> 可执行文件。

gcc 编译过程文件变化：main.c -> main.i -> mian.s -> main.o -> main 。

通过上面分阶段的介绍，我们也明白了 gcc 其实只是一个后台程序的包装，它会根据不同阶段来调用 cpp、cc、as、ld 等命令。

可以看到**编译这一步就是把源码转换成汇编码，但其实上面所有步骤都可以理解为是编译器做的，因为我们通常说的编译器其实是包含了编译器、汇编器和链接器的**，在 IDE 中这个过程也不再分的那么细，比如在 VS 里点"编译"就直接生成一个可执行文件(.exe)。我们常说的 GCC 编译器，内部也是靠驱动 cc1、as 和 ld 三个部件完成编译、汇编和连接的工作。cc1 将 C 语言源文件编译为汇编文件（.s）。AS 将汇编代码转换为二进制的目标文件（.o）；生成的这些目标文件再由 AR 程序打包成静态库（.a），或者由 LD 程序连接成可执行程序（elf、.so 或其他格式），而 LD 就是所谓的连接器。

【gcc 编译生成可执行文件的过程中发生了什么】

7.2 编译器/解释器

编译器：**针对特定的操作系统**把源码**整体**“翻译”成机器码（目标文件），并通过链接器**封装成该系统可执行程序的格式**（注意这个可执行文件虽然是二进制文件但 CPU 是不能直接运行的，这是给操作系统运行的，如果 CPU 可以直接执行的话，那 Windows 上编译后也可以在 Linux 上执行，就不存在跨平台的问题了）。

解释器：解释器并**不会**把源代码预编译成机器码和可执行文件，而是**对源码逐行解释执行**。编译器自己不会执行代码，最终靠的是操作系统运行可执行文件得到结果，但解释器可以自己解析并执行源码，这也是解释型语言跨平台的原因，解释器通常会用以下几种方式来执行程序代码：

1. 分析源代码，并且直接执行（早期 Bisic）。
2. 把源代码翻译成相对更加高效的中间码然后立即执行它（Python）。
3. 执行由解释器内部的编译器预编译后保存的代码（Java）。

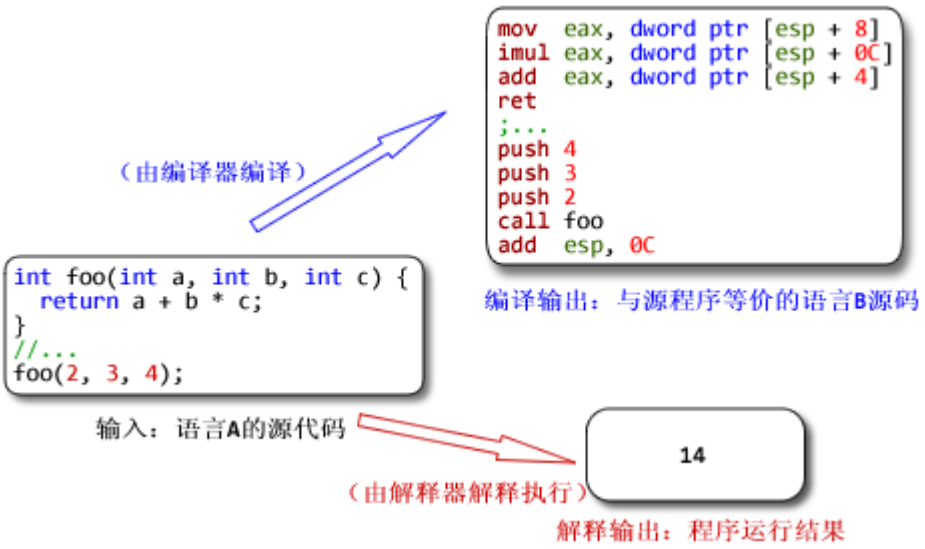
简单说**源码经过编译器产生机器码（表现为一个可执行文件），**操作系统再执行可执行文件输出结果，而解释器呢，**源码经过解释器直接输出结果，**解释器翻译执行一步到位。不是说解释器就不用分析处理源码，而是这个过程被封装了起来，我们看到的，就是解释器直接执行了源码得到结果。

7.3 编译型/解释型语言

编译型语言和解释型语言，这两个概念我觉得是从表象划分的，因为本质上任何高级语言想要执行都必须经过解析（词法分析、语法分析等）和翻译（源码转换成二进制机器码）才能执行，但在我们看来：

编译型语言（Compiled language）：**输入代码 -> 编译得到可执行文件（二进制机器码） -> 操作系统运行可执行文件输出结果**。编译器要做的就是**把整个源码编译成一个可执行文件**，运行的时候不再依赖编译器。

解释型语言（Interpreted language）：**输入代码 -> 解释器解释执行输出结果**。直接就得到结果，计算机只认二进制机器码，所以解释器内部肯定有个复杂的解释执行过程，但我们看到的就是这么简单，解释器要做的就是**对源码逐行解释执行**，意味着每**次执行都要依赖于解释器，执行一次解释一次**。



编译型和解释型的区别到底是什么呢？我觉得主要是三点：

1. 执行流程上不同。**编译型语言有一个单独的编译过程，把程序的执行分离成翻译和执行两步**，第一步把源码经过编译链接生成可执行文件，第二步操作系统执行可执行文件；而**解释型语言只有一步，解释器对源码解释并执行（虽然这个解释的过程也很复杂，但解释和执行是一个整体）**，其实很多解释型语言也产生中间代码，比如 Python 的 pyc 文件，但这也是由解释器做的。

2. 产生的中间码不同。**编译型语言会产生一个本地平台能运行的可执行文件，然后由当前操作系统运行。解释型语言为了提高效率一般也会产生一个中间码，叫做字节码，然后由解释器执行字节码**。解释器相当于在操作系统上又封装了一层，当然底层肯定是硬件 CPU 在执行，不过低于操作系统的底层环节这里不讨论。

3. 跨平台。编译型语言**针对不同型号的 CPU 和操作系统，需要编译成不同的可执行文件**，因为是操作系统直接执行，所以 Windows 下编译的可执行文件在 Linux 是不能执行的，所以**编译型语言无法跨平台**。但解释型语言靠解释器翻译和执行，不管什么系统**只要安装了相应的解释器（或虚拟机）就能执行**，这也是解释型语言跨平台的原因。

编译型语言和解释型语言的优缺点？

1. 编译型语言的优点是“静态”，代码必须作为整个工程来编译，这样**便于类型检查**，降低运行时错误率，编译之后可以在相应的平台上独立运行，不再依赖编译器，**运行时效率高（运行前的编译并不快）**。缺点是**不能跨平台**，不同的平台需要编译成不同的可执行文件，一旦修改源码就需要重新编译。编译型语言包括：C、C++等。

2. 解释型语言的优点是“动态”，**代码的每一行可独立解释执行（代码块除外），跨平台性好、可以灵活修改代码**。缺点是**执行速度慢**、效率低（因为每次执行都要解释一次）。解释型语言包括：JS、Python 等。

我们常说 C 语言是编译型语言，Python 是解释型语言，指的是它们主流或者默认的执行方式，C 主流是用 GCC 编译器，Python 主流是用 CPython 解释器，但并不绝对，你可以给 C 写一个解释器来解释执行，也可以给 Python 写一个编译器来编译执行，那为什么 Python 选择解释执行而不选择编译执行呢？想想 Python 的特性是什么，动态语言，灵活修改，如果编译执行的话每次修改都要重新编译，虽然编译之后的可执行文件运行起来快，但编译过程并不快，所以解释执行更符合动态语言的特性。**解释和编译都只是程序从源码到运行时的一种动作，跟语言本身无关**。Java 就可以归为编译+解释的混合型语言。

虽然解释器看起来是直接执行了源码，但实际上，**大部分解释器还是会先把源码编译成一种字节码，然后再执行**，这样能提高运行效率。比如 Python 会先把源码编译成字节码，然后解释器再执行字节码。解释器产生的字节码和编译器产生的机器码是不同的，编译器产生的机器码一般可以用操作系统直接运行，但字节码需要相应的解释器运行。

解释型语言到底有没有编译的过程？我认为是有的，我理解的编译是对源码的预处理和解析，而不是看有没有产生可执行文件，那其实解释型语言也得分析源码，任何**高级语言**想要执行都要经过预处理和解析，至少得解析出语法树来吧。所以其实**编译型语言和解释型语言的概念是比较模糊的**，早期 Java 明明是靠虚拟机解释执行的，但现在却引入了即时编译（JIT）。

7.4 Python/Java 的执行过程

7.4.1 Python 执行过程

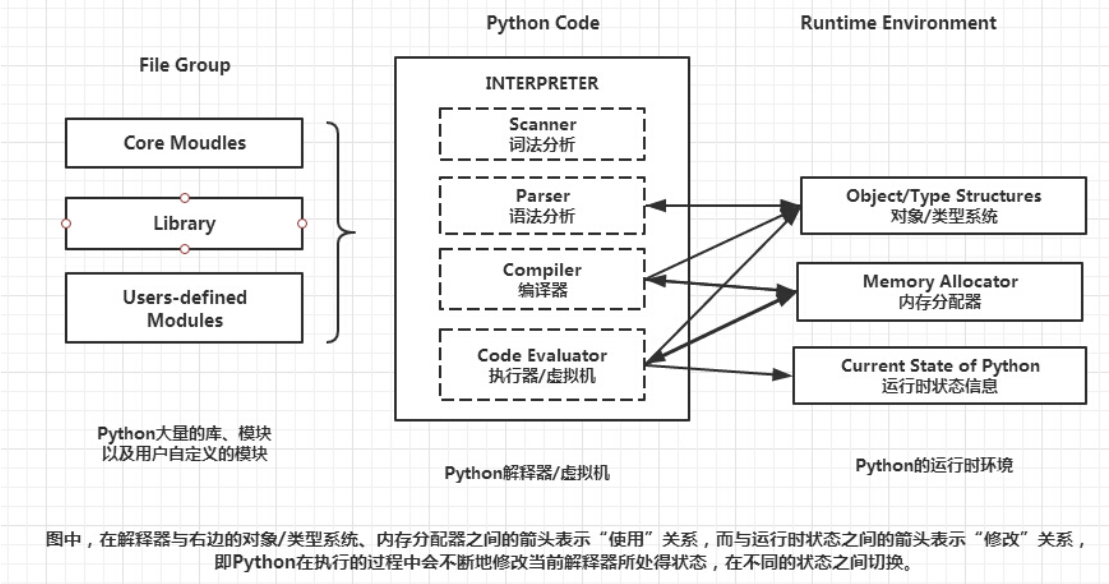
Python 在执行程序时，为了提高效率，CPython 解释器并不会直接对源码进行解释执行，而是先将源码编译成字节码（注意字节码不同于编译器产生的可执行机器码，编译器产生的可执行机器码只能由本地操作系统运行，而字节码和操作系统无关，需要由相应的解释器运行），然后再由解释器逐行执行这些字节码，程序每执行一次，就重复一次这个过程，这其中有大量的重复性的工作，那能不能进行优化呢？想想编译型语言的优点是什么，执行时候无需重复编译，那是不是 Python 也可以把一部分编译过的字节码保存下来，这样以后执行这部分代码就无需重复编译，那保存哪部分代码呢？保存那些**重用性很高，几乎不变的代码**。CPython 解释器认为：import 进来的模块，就是重用性很高的代码。

当运行 test.py 时，如果遇到 import abc，会先去设定好的 path 中寻找 abc.pyc 文件，如果找到并且 abc.py 文件的时间戳早于 abc.pyc 文件（说明生成 abc.pyc 文件后 abc.py 文件就没再变过），那就直接载入运行，否则会先将 abc.py 编译成 PyCodeObject 对象（也就是字节码）并将其写入内存当中，因为是 import 的模块，所以 PyCodeObject 对象还会被保存到硬盘上，形成我们看到的 pyc 文件，最后才会对 pyc 进行 import 操作，将 pyc 中的字节码重新复制到内存，并运行。

我们之所以要生成.pyc 文件，是为了下一次再运行程序时，不需要重新编译那些重复使用的模块，直接加载.pyc 文件就可以了，这无疑大大提高了程序运行速度。对于仅仅运行一次的程序，保存其 pyc 文件是没有必要的。注意 **Python 程序在内存中执行并不是依靠 pyc 文件来完成，pyc 文件只是为了加速，Python 的执行靠的是内存中的 PyCodeObject 对象**。

CPython 使用的是基于栈的虚拟机，下面是 CPython 解释器的运行机制：

1. Scanner 对应词法分析器，将从文件输入的代码切分为 token。
2. Parser 对应语法分析器，在 Scanner 的分析结果上进行语法分析，建立 AST。
3. Compiler 根据建立的 AST 生成指令集合 Python 字节码（这里的 .pyc 文件类似于 Java 的 .class 文件）。
4. 最后由 Code Evaluator 执行代码。
5. 解释器中的 Code Evaluator 就是你口中的 PVM（Python 虚拟机）。



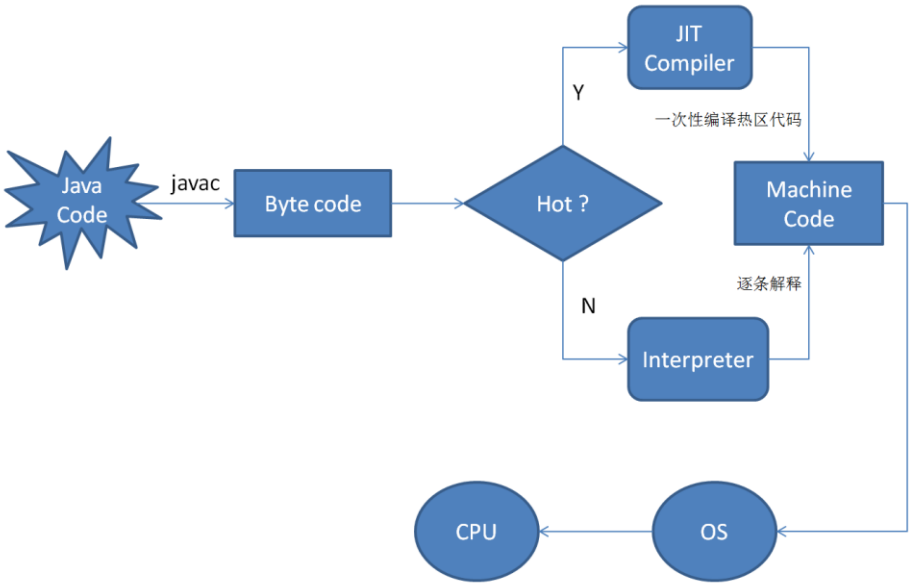
图中，在解释器与右边的对象/类型系统、内存分配器之间的箭头表示“使用”关系，而与运行时状态之间的箭头表示“修改”关系，即Python在执行的过程中会不断地修改当前解释器所处得状态，在不同的状态之间切换。

【编译器与解释器】
【什么是解释型语言？ - 李广胜的回答 - 知乎】
【编译器与解释器的区别和工作原理】
【知乎解释器和虚拟机的区别有哪些？ - 无与童比的回答 - 知乎】
【Python 编译：code 对象 与 pyc 文件】
【Python 什么情况下会生成 pyc 文件？ - 刘凯的回答 - 知乎】

7.4.2 Java 执行过程

首先我比较倾向于 Java 是解释 + 编译的混合型语言，早期的 Java 是先通过 Javac 将源码编译成.class 字节码，然后 JVM 对字节码逐行解释执行，标准的解释型语言。但是当某个方法或代码块运行很频繁时，重复的解释执行效率就很低，为了提高执行速度，引入了即时编译（Just In Time Compiler，JIT），**JVM 中的 JIT 编译器会把热点代码（经常执行的代码）编译成与本地平台相关的机器码**，以后执行到这部分代码，操作系统直接执行编译好的机器码就行了，不必重复解释执行，**其它代码由 JVM 中的解释器逐行解释执行**。这里也能看出 Python 和 Java 的区别了，Python 的 pyc 文件终究还是靠解释器解释执行的，而 Java 的 class 文件是有可能被 JIT 编译成本地机器码的。

Java 的执行过程看下图：



Javac 编译器有时候也被叫做前端编译器，因为其发生在整个编译的前期。**Javac 编译器的处理过程可以分为下面四个阶段：**

1. 词法、语法分析。在这个阶段，JVM 会对源代码的字符进行一次扫描，最终生成一个抽象的语法树。简单地说，在这个阶段 JVM 会搞懂我们的代码到底想要干嘛。就像我们分析一个句子一样，我们会对句子划分主谓宾，弄清楚这个句子要表达的意思一样。

2. 填充符号表。我们知道类之间是会互相引用的，但在编译阶段，我们无法确定其具体的地址，所以我们会使用一个符号来替代。在这个阶段做的就是类似的事情，即对抽象的类或接口进行符号填充。等到类加载阶段，JVM 会将符号替换成具体的内存地址。

3. 注解处理。我们知道 Java 是支持注解的，因此在这个阶段会对注解进行分析，根据注解的作用将其还原成具体的指令集。

4. 分析与字节码生成。到了这个阶段，JVM 便会根据上面几个阶段分析出来的结果，进行字节码的生成，最终输出为 class 文件。

JIT 编译：

现在主流的商用虚拟机（如 Sun HotSpot、IBM J9）中几乎都同时包含解释器和编译器（三大商用虚拟机之一的 JRockit 是个例外，它内部没有解释器，因此会有启动相应时间长之类的缺点，但它主要是面向服务端的应用，这类应用一般不会重点关注启动时间）。二者各有优势：当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即执行；当程序运行后，随着时间的推移，编译器逐渐会返回作用，把越来越多的代码编译成本地代码后，可以获取更高的执行效率。解释执行可以节约内存，而编译执行可以提升效率。

HotSpot 虚拟机中内置了两个 JIT 编译器：Client Complier 和 Server Complier，分别用在客户端和服务端，目前主流的 HotSpot 虚拟机中默认是采用解释器与其中一个编译器直接配合的方式工作。

运行过程中会被即时编译器编译的“热点代码”有两类：被多次调用的方法和被多次调用的循环体。两种情况，编译器都是以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。要知道一段代码或方法是不是热点代码，是不是需要触发即时编译，需要进行 Hot Spot Detection（热点探测）。目前主要的热点　判定方式有以下两种：

1. 基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这段方法代码就是“热点代码”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

2. 基于计数器的热点探测：采用这种方法的虚拟机会为每个方法，甚至是代码块建立计数器，统计方法的执行次数，如果执行次数超过一定的阀值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

在 HotSpot 虚拟机中使用的是基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器和回边计数器。

方法调用计数器用来统计方法调用的次数，在默认设置下，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间内方法被调用的次数。

回边计数器用于统计一个方法中循环体代码执行的次数（准确地说，应该是回边的次数，因为并非所有的循环都是回边），在字节码中遇到控制流向后跳转的指令就称为“回边”。

【【深入 Java 虚拟机】之七：Javac 编译与 JIT 编译】
【深入浅出 JIT 编译器】

【虚拟机随谈（一）：解释器，树遍历解释器，基于栈与基于寄存器大杂烩】
【JVM 基础系列第 4 讲：从源代码到机器码，发生了什么？】
【为什么大多数解释器都将 AST 转化成字节码再用虚拟机执行，而不是直接解释 AST？ - RednaxelaFX 的回答 - 知乎】

7.5 动态类型/静态类型语言

动态类型语言（Dynamically Typed Language），动态类型语言是指**在运行时才去做数据类型的检查**。在用动态语言编程时，**不用给变量指定数据类型**，第一次赋值给变量时，在内部将数据类型记录下来。动态类型语言有 JS、Ruby、Python 等。

动态类型语言的特点是：1. 变量在使用前不用先声明类型。2. 不同类型的变量之间界限模糊，如果需要在任何时候都可以直接把它转变为另一种类型。比如 Python 的变量也有类型，但是它们的类型不是由程序员在程序中显式给出的，事实上『Python 的变量类型是根据右值（赋值号右边的值）由解释器判断出来的』

静态类型语言（Statically Typed Language），静态类型语言与动态类型语言刚好相反，它的数据类型检查发生在在编译阶段，也就是说在**写程序时要声明变量的数据类型**。静态类型语言有 C/C++、C#、Java 等。

静态类型语言的特点是：1.变量在使用前要先声明类型。2. 不同类型的变量之间界限分明，必须显式的使用强制类型转换以改变变量的原本类型。

大家觉得 C 语言要写 int a, int b 之类的，Python 不用写(可以直接写 a, b)，所以 C 是静态类型，Python 是动态类型，这样理解是不够准确的。譬如 Ocaml 是静态类型的，但是也可以不用明确地写出来。Ocaml 是静态隐式类型。

无类型： 汇编
弱类型、静态类型 ： C/C++
弱类型、动态类型检查： Perl/PHP
强类型、静态类型检查 ： Java/C#
强类型、动态类型检查 ： Python, Scheme
静态显式类型 ： Java/C
静态隐式类型 ： Ocaml, Haskell

【动态语言 vs 动态类型语言】
【弱类型、强类型、动态类型、静态类型语言的区别是什么？】

7.6 动态/静态语言

动态语言（Dynamic Programming Language），动态语言是指可以在运行时改变其结构的语言，如增加或者删除函数、对象、属性、方法等。动态语言有 JS、Objective-C、Ruby、Python 等。

Python 在运行时添加 sex 属性：

```
>>> class Person(object):
    def __init__(self, name = None, age = None):
        self.name = name
        self.age = age

>>> P = Person("小明", "24")
>>> P.sex = "male"
>>> P.sex
'male'
```

静态语言（StaticProgramming Language），静态语言与动态语言相反，在运行时不能改变其结构，受到的约束性更强，相应的也更稳定，因为你无法随意修改它的方法属性等。其实静态语言也可以通过复杂的手段实现动态语言的特性。静态语言有 C/C++、Java 等。

7.7 强类型/弱类型

首先强类型、弱类型和语言的静态或动态是没有关系的。

强类型（weakly typed）：当你定义一个变量是某个类型，如果**不经过代码显式转换（强制转化）过，它就永远都是这个类型**，如果把它当做其他类型来用，就会报错。Java 和 Python 是强制类型定义的。如果不进行转换，你不可以把一个整数和字符串相加。

弱类型（strongly typed）：你想把这个变量当做什么类型来用，就当做什么类型来用，**语言的解析器会自动（隐式）转换**。弱类型语言有 VBS、PHP、C/C++等。

第八章 操作系统

先介绍操作系统中的几个常见概念，再介绍 Python 中的一些特点。比如先介绍什么是多线程，再说明为什么 Python 中的多线程是伪多线程。

8.1 并发和并行

我觉得并发（concurrency）和并行（parallelism）其实并不算是一个维度的概念，并发是指程序的设计结构，并行是指程序的运行状态。如果某个系统支持至少两个动作（Action）**同时存在**，那么这个系统就是一个**并发系统**。如果某个系统支持至少两个动作**同时执行**，那么这个系统就是一个**并行系统**。并发系统与并行系统这两个定义之间的关键差异在于“存在”这个词。

在并发程序中可以同时拥有两个或者多个线程。这意味着，如果程序在单核处理器上运行，那么这两个线程将交替地换入或者换出内存。这些线程是同时“存在”的——每个线程都处于执行过程中的某个状态。如果程序要并行执行，那么就一定是运行在多核处理器上。此时，程序中的每个线程都将分配到一个独立的处理器核上，因此可以同时运行。

也就是说，你可以编写一个拥有多个线程或者多进程的并发程序，但如果没有多核处理器来执行这个程序，那么就不能以并行方式来运行。因此，凡是在求解单个问题时涉及多个执行流程的编程模式或者执行行为，都属于并发编程的范畴。

【并发与并行的区别？ - Limbo 的回答 - 知乎】

另一个回答很形象，答主和跟帖的回答都有道理，角度不同，我略做修改，其中吃饭和电话是两个任务，嘴是 CPU：

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你说完一句吃一口，咽下一口说一句，这说明你支持并发。并发可以理解为**同一时间段内多个任务交替执行**。

你吃饭吃到一半，电话来了，你有两张嘴，边吃边说，这说明你支持并行。并行可以理解为**同一时刻多个任务同时执行**，所以一定要多核。

【并发与并行的区别？ - 「已注销」的回答 - 知乎】

8.2 分布式和高并发

这也是两个经常听到的术语，简单介绍一下：

分布式是为了解决单个物理服务器的容量和性能的瓶颈问题而采用的优化手段。该领域需要解决的问题很多，在不同的技术层面上，又包含分布式文件系统、分布式数据库、分布式缓存、分布式计算等等，Hadoop、zookeeper、MQ等也都跟分布式有关。从理念上讲，分布式的实现有两种方式：**水平扩展， 当一台机器的容量抗不住的时候，需要增加机器的方式**，将流量平均到所有服务器上，所有机器都可以提供相当的服务；**垂直扩展，比如前端有多种需求查询时，一台机器扛不住，可以将不同的需求分发到不同的机器上**，比如 A 机器处理余票查询的时候，B 机器可以处理支付请求等。

高并发反映的是同时有多少流量进来。比如：在线直播，秒杀等，同时有上万人观看，参抢。**高并发可以用分布式来解决**，将并发的流量分到不同的物理服务器上。但除此外，**还可以使用缓存、将所有的静态内容放到 CDN、使用多线程技术将一台服务的服务能力最大化等**。

总结一下：

分布式是从**物理资源的角度**去将不同的机器组成一个整体对外提供服务的，技术范围非常广且难度非常大，有了这个基础，高并发，高吞吐等系统很容易实现。

高并发是从以**业务角度去描述系统的能力**，实现高并发的手段可以采用分布式，也可以采用诸如：缓存、CND、多线程等。

【分布式，高并发，多线程之间有啥区别】

8.3 线程和进程

进程（process）：**进程是操作系统分配资源的最小单元**，进程是 OS 层面的东西。一个应用程序至少包括一个进程，**每个进程都有自己独立的内存空间**，不同进程之间的内存空间不共享，进程之间的通信由操作系统传递，**通讯效率较低，切换开销较大**。

线程（thread）：**线程是 CPU 调度的最小单元**，CPU 能看到的是线程不是进程。一个进程至少包括一个线程，**一个进程内所有线程共享进程的内存空间**，当多个线程同时读写同一份共享资源的时候，可能会引起冲突，这就需要引入**线程同步和互斥**机制。线程**通讯效率较高，切换开销较小**。

做个简单的比喻：进程 = 火车，线程 = 车厢：

- 线程在进程下行进（单纯的车厢无法运行）。
- 一个进程可以包含多个线程（一辆火车可以有多个车厢）。
- 不同进程之间的数据共享比较难（一辆火车上的乘客转移到另外一辆火车上比较难，可能要到换乘点才行）。
- 同一进程下不同线程之间的数据共享容易（同一火车的 A 车厢换到 B 车厢很容易）。
- 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）。
- 进程间不会相互影响，但一个线程挂掉一般会导致整个进程挂掉，好像 JAVA 是个例外？（一列火车不会影响到另外一列火车，但是如果一列火车上的某一节车厢着火了，一般会影响整列火车）。
- 进程可以拓展到多机，进程最适合多核，线程是包含在进程内的（不同火车可以开在多个轨道上）。
- 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）—"互斥锁"。
- 进程使用的内存地址可以限定使用量（比如火车上的餐厅，最多只允许多少人进入，如果满了需要在门口等，等有人出来了才能进去）—“信号量”。

那该用线程还是该用进程呢？

	多进程	多线程
进程/线程同步	简单	复杂
数据共享	复杂	简单
CPU/内存占用	多	少
创建/销毁/切换	慢	快
可靠性	多进程互不影响	一个挂掉影响整体

综上，多线程和多进程都能充分利用多核心（除了 Python），但是：

需要频繁创建销毁的（I/O 密集型场景）优先用多线程。涉及到网络、磁盘读写的任务都是 I/O 密集型任务，这类任务的特点是 CPU 消耗很少，大部分时间在等待 I/O 操作，进程切换消耗大量时间，所以用多线程合适。

需要进行大量计算的（CPU 密集型场景）优先使用多线程（Python 由于 GIL 反而要用多进程）。比如大量计算、高清解码这种主要消耗 CPU 资源，多线程和多进程都可以充分利用多核心的优事，但**多进程的切换比多线程要慢，**所以选择多线程。注意在 Python 中 CPU 密集型反而要用多进程，这主要是因为多进程能突破 GIL 的限制，Python 的多线程是伪多线程。

强相关的处理用多线程，弱相关的处理用多进程。

可能要扩展到多机分布的用多进程，单机多核分布式场景用多线程。

【线程和进程的区别是什么？ - biaodianfu 的回答 - 知乎】

8.4 协程

8.4.1 什么是协程

协程（Coroutine），又称微线程，我们把一个线程中的一个个函数叫做子程序，子程序的调用是通过栈实现的，调用顺序是明确的，协程和子程序的调用差不多，但在子程序内部是可中断的，**可以在执行一个子程序的时候中断转而执行别的子程序，在适当的时候再返回来接着执行，这就是协程**。

在一个子程序中中断去执行其他子程序，**不是函数调用，有点类似 CPU 的中断。协程是程序级别的，由程序员根据需要自己调度**。

8.4.2 协程和线/进程

线程进程和协程的区别：

线程进程是调用的系统 API，调度是由 CPU 来决定调度的，而**协程是由程序员实现的**，协程不同于线程的是，**线程是抢占式的调度，而协程是协同式的调度**，也就是说，**协程需要自己做调度**。在其他语言中，其实协程的意义不是特别大，多线程就可以解决 I/O 的问题，但因为 Python 主流解释器 CPython 有 GIL，导致多线程是伪多线程，所以如果一个线程里面 I/O 操作特别多，用协程是最合适的，因为 Python 的多线程是伪多线程，多进程虽然能充分利用多核，但 I/O 操作一般单核就足够了，多进程在进程切换上还要浪费很多时间。

协程的优点：

不管是进程还是线程，每次阻塞、切换都需要陷入系统调用(system call)，先让 CPU 跑操作系统的调度程序，然后再由调度程序决定该跑哪一个进程(线程)，这就避免不了调度切换的开销。

协程的特点在于是一个线程执行，最大的优势就是协程极高的执行效率。因为**子程序切换不是线程切换，而是由程序自身控制，因此没有线程切换的开销**，和多线程比，线程数量越多，协程的性能优势就越明显。但也因此，程序员必须自己承担调度的责任，同时协程也失去了使用多 CPU 的能力。

另外由于抢占式调度执行顺序无法确定的特点，使用线程时需要非常小心地处理同步问题，而**协程不需要多线程的锁机制**，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

协程的缺点：

无法利用多核资源：协程的本质是个单线程,它不能同时将单个 CPU 的多个核用上,协程需要和进程配合才能运行在多 CPU 上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

进行阻塞（Blocking）操作（如 IO 时）会阻塞掉整个程序

【python 协程是什么？ - 侠三十六的回答 - 知乎】

8.5 Python 多进程/多线程/协程

8.6 Python 多线程和 GIL

8.6.1 GIL（Global Interpreter Lock）

全局解释器锁（Global Interpreter Lock，GIL），GIL 其实是一个互斥锁，**用于同步线程的一种机制，使用 GIL 的解释器在任何时刻仅有一个线程在执行，即便在多核心处理器上**。每一个进程中只有一个 GIL 锁，当我们使用多线程时，**哪个线程拿到 GIL 锁，哪个线程就可以使用 CPU**，所以当 Python 用 CPython 作为解释器的时候，其实是伪多线程。常见的使用 GIL 的解释器有 CPython 与 Ruby MRI。

那 GIL 怎么释放呢？

1. CPU 闲置时会释放 GIL，比如 I/O 操作。

2. 会有一个专门 ticks 进行计数 一旦 ticks 数值达到一个阈值就会释放 GIL，线程之间开始竞争 GIL。

Python 中多线程和多进程的选择

I/O 密集型操作（网络 I/O，磁盘 I/O，数据库 I/O）适合用多线程，因为 I/O 密集的 CPU 占用率很低，单个 CPU 核心基本就足够用了，多线程正好充分利用 CPU 资源，用多进程给它多几个核心并不能显著提升性能，而且进程切换的时间比线程慢很多。

CPU 密集型操作时候用多进程。这是 CPython 解释器的特点，因为受 GIL 限制，Python 同一时刻只能有一个线程在执行，但是每个进程都有独立的 GIL，所以多进程能突破 GIL 的限制。

8.6.2 GIL 是怎么来的

在多核处理器普及后，为了有效的利用多核处理器的性能，就出现了多线程的编程方式，我们知道一个进程内的多线程是共享内存空间的，多个线程如果操作同一共享资源很容易产生冲突，Python 要支持多线程就得解决这个问题，而解决多线程之间数据完整性和状态同步的最简单方法就是加锁，于是就有了 GIL 这把超级大锁。

8.6.3 GIL 优缺点

GIL 的优点：GIL 可以保证我们在多线程编程时，无需考虑多线程之间数据完整性和状态同步的问题。

GIL 的缺点：不是真正意义上的的多线程，实际上是多个线程轮流被解释器执行，只不过切换的很快，给人一种多线程“同时”在执行的错觉。不过多进程可以突破 GIL 的限制，因为每个进程有自己的独立的 GIL。

综上可以总结出：

1. 因为 GIL 的存在，在 I/O 密集型操作场景下的多线程性能才会比较好。
2. 如果对并行计算性能较高的程序可以考虑把核心部分也成 C 模块，或者索性用其他语言实现
3. 在 Python 编程中，如果想利用计算机的多核提高程序执行效率，用多进程代替多线程
4. 即使有 GIL 存在，由于 GIL 只保护 Python 解释器的状态，所以对于非原子操作，在 Python 进行多线程编程时也需要使用互斥锁（如 thread 中的 lock）保证线程安全。
5. GIL 在较长一段时间内将会继续存在，但是会不断对其进行改进。

8.6.4 为什么不去掉 GIL

首先要知道 GIL 并不是 Python 的特性，确切的说应该是 CPython 解释器的特性，Python 也可以用其它解释器，比如 Jython 解释器就没有 GIL。其次 Python 社区为什么不解决 GIL 的问题。

一方面是因为 **Python（指 CPython 解释器）性能低的问题，GIL 只是其中之一，动态特性和解释执行才是主要原因**，关于动态语言、静态语言、静态语言、编译执行的选择，就像鱼和熊掌，你不可能既拥有动态语言的灵活性和解释执行的跨平台，还拥有静态语言的可靠性和编译执行的高效运行，**GIL 影响的主要是计算密集型的多线程程序**，但换句话说 CPU 密集型任务选择 Python 实现本身就是不明智的。

另一方面 GIL 也确实让开发过程变得简单，因为你不需要考虑多线程之间数据完整性和状态同步的问题，不需要考虑额外的内存锁和同步操作，所以代码库开发者们大量依赖这种特性，几十年下来**想要去掉 GIL 已经是非常困难的事情了，而且去掉 GIL 后性能提升并不是特别显著**。

8.6.5 GIL 和线程锁

既然有了 GIL，为什么还要用线程锁，还要关注线程安全？**GIL 控制的是字节码，线程锁控制的是 Python 代码，粒度不一样，GIL 保证了在字节码层面是线程安全的。**

我们知道 CPython 会把源码编译成字节码，然后解释器逐行执行，虽然由于 GIL 的存在一次只能有一个线程在执行某行字节码，但问题是，**一行代码可能编译成多行字节码，多行字节码处理同一变量就可能会出现问题了**，比如两个线程 A、B 操作的是同一变量，就算它们交替执行，也很容易出错，看一个例子：代码 self.number += 1 划分为原子操作是三步：

1. 读取 self.number 的值。
2. 计算 self.number + 1。
3. 结果写入 self.number。

如果在两个线程 0 和 1 中执行此操作，可能的执行顺序如下：

正确的执行顺序（加锁）	错误的执行顺序（无锁）
thread1 reads self.number (0)	thread1 reads self.number (0)
thread1 calculates number+1 (1)	thread2 reads self.number (0)
thread1 writes 1 to self.number	thread1 calculates number+1 (1)
thread2 reads self.number (1)	thread2 calculates number+1 (1)
thread2 calculates number+1 (2)	thread1 writes 1 to self.number
thread2 writes 2 to self.number	thread2 writes 1 to self.number

所以多线程中对于非原子操作，要考虑加锁，来看一段完整的测试代码：

```
import threading
import time

total = 0
lock = threading.Lock()
def increment_n_times(n):
    global total
    for _ in range(n):
        total += 1

def safe_increment_n_times(n):
    global total
    for _ in range(n):
        lock.acquire()
        total += 1
        lock.release()
```

```
def increment_in_x_threads(x, func, n):
    threads = [threading.Thread(target=func, args=(n,)) \
               for _ in range(x)]
    global total
    total = 0
    begin = time.time()
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
    print('finished in {}s.\ntotal: {}\nexpected: {}\ndifference: {}'
          .format(time.time()-begin, total, n*x, n*x-total))

if __name__ == '__main__':
    print('unsafe:')
    increment_in_x_threads(70, increment_n_times, 100000)
    print('\nwith locks:')
    increment_in_x_threads(70, safe_increment_n_times, 100000)
```

先说结论，不加锁的话正确率无法保证但是速度很快，加锁的话可以保证正确率但速度几乎降了 20 倍：

```
"""
unsafe:
finished in 0.8178131580352783s.
total: 6485996
expected: 7000000
difference: 514004

with locks:
finished in 14.957011461257935s.
total: 7000000
expected: 7000000
difference: 0
"""
```

【Why do we need locks for threads, if we have GIL?】

8.7 线程互斥和同步

互斥和同步是两种典型的并发问题。恰当的使用锁，可以解决同步或者互斥的问题，当然也不仅仅只有锁这一种解决方式。

互斥：是指**某一资源同时只允许一个访问者对其进行访问**，一个操作进行时另一个操作无法进行,具有**唯一性和排它性**。但**互斥无法限制访问者的访问顺序**，即访问是无序的。

例子：线程 A,B,C,D 要对同一资源进行操作，A 执行过程中 B,C,D 只能等， A 释放锁之后 B,C,D 谁抢到就谁执行。

同步：是指**在互斥的基础上**（大多数情况），通过其它机制实现**访问者对资源的有序访问**，比如生产者消费者问题，消费者必须在生产者之后运行。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的，少数情况是指可以允许多个访问者同时访问资源。

例子：A,B,C,D 要对同一资源进行操作，它们要约定一个执行的顺序，比如 B 和 C 都执行完 D 才能执行，而 B 和 C 要执行的话需要 A 先做完。

首先不要钻概念牛角尖，这样没意义（道理我都懂，但是忍不住啊！）。锁就是避免多个线程，对同一个共享的数据并发修改带来的数据混乱。锁要解决的大概就只有这 4 个问题：

1. 谁拿到了锁"这个信息存哪里（可以是当前 class，当前 instance 的 markword，还可以是某个具体的 Lock 的实例）
2. 谁能抢到锁的规则（只能一个人抢到，比如互斥锁；能抢有限多个数量，比如信号量；自己可以反复抢，比如重入锁；读可以反复抢到但是写独占，比如读写锁……）。
3. 抢不到时怎么办（抢不到玩命抢；抢不到暂时睡着，等一段时间再试/等通知再试；或者二者的结合，先玩命抢几次，还没抢到就睡着）。
4. 如果锁被释放了还有其他等待锁的怎么办（不管，让等的线程通过超时机制自己抢；按照一定规则通知某一个等待的线程；通知所有线程唤醒他们，让他们一起抢……）

有了这些选择，你就可以按照业务需求组装出你需要锁。你可以说 Mutex（互斥锁）是专门被设计来解决互斥的；Barrier（栅栏，设置一个栅栏并指定数量，直到指定的线程数量达到指定的数，全部线程才能继续执行）和 Semphore（信号量）是专门来解决同步的。但是这些都离不开上述对上述 4 个问题的处理。同时，如果遇到了其他的具体的并发问题，你也可以定制一个锁来满足需要。

总的来说，两者的区别就是：

1. 互斥是通过竞争对资源的独占使用，彼此之间不需要知道对方的存在，执行顺序是未指定的。

2. 同步是协调多个相互关联线程合作完成任务，彼此之间知道对方存在，执行顺序往往是有序的。

【Java 中线程同步锁和互斥锁有啥区别？看完你还是一脸懵逼？】

8.7.1 互斥锁和信号量

互斥锁（Mutex），用于**多线程多任务互斥，一种安全有序的让多个线程访问内存空间的机制**。因为一个进程内的**多线程共享内存空间**，共享意味着竞争，可能导致**数据不安全**，而解决多线程之间数据完整性和状态同步的最简单方法就是加锁，即一个线程在访问内存空间的时候，其他线程不允许访问，必须等待之前的线程访问结束，才能使用这个内存空间。要注意谁加锁就必须由谁解锁。Mutex 对象的值只有 0 和 1 两个值，这两个值也分别代表了 Mutex 的锁定和空闲两种状态。

信号量（Semaphore），用于**多线程多任务同步**，是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确、合理的使用公共资源。当信号量只允许取 0 或 1 时就是二进制信号量（binary semaphore）。

8.7.2 信号量和互斥锁的区别

信号量是流程上的概念，并不一定要锁定某个资源，比如：有 A,B 两个线程，B 线程要等 A 线程完成某一任务以后再进行自己下面的步骤，这个任务并不一定是锁定某一资源，还可以是进行一些计算或者数据处理之类。

而**互斥锁则是“锁住某一资源”的概念**，在锁定期间内，其他线程无法对被保护的数据进行操作。在有些情况下两者可以互换。

Mutex 的用途，一句话：**保护共享资源**。典型的例子就是买票：票是共享资源，现在有两个线程同时过来买票。如果你不用 mutex 在线程里把票锁住，那么就可能出现“把同一张票卖给两个不同的人（线程）”的情况。

Semaphore 的用途，一句话：**调度线程**，调度线程也就是一些线程生产（increase），同时另一些线程消费（decrease），Semaphore 可以让生产和消费保持合乎逻辑的执行顺序。有的人用 Semaphore 也可以把上面例子中的票“保护”起来以防止共享资源冲突，必须承认这是可行的，但是 Semaphore 不是让你用来做这个的；如果你要做这件事，请用 mutex。

简而言之，**锁是服务于共享资源的；而 Semaphore 是服务于多个线程间的执行的逻辑顺序的**。

线程对于资源有两种操作关系：

1. **竞争关系**，即多个线程竞争同一资源供需；
2. **供需关系**，即进程之间会存在资源的生产者和消费者的关系。

Semaphore 是站在线程的角度考虑问题的，它可以被任意的线程获取和释放，即可以解决供需关系，也可以解决竞争关系。供需关系的例子，有时候会

遇到这样的情况，A 线程需要的资源是 B 线程准备的，那么我就需要 A 先占有信号量，然后等 B 准备好资源后，再释放信号量保证 A 的后续运行。

mutex 是站在资源的角度考虑问题的，解决的竞争关系，如果资源被占用了，为了保证原子操作，无论如何他不能再被任何线程占用，那么理论上二元信号量可以用来完成资源的保护，参与资源的竞争关系，但不建议这么做。

【semaphore 和 mutex 的区别？ - 二律背反的回答 - 知乎】
【semaphore 和 mutex 的区别？ - 人马座的回答 - 知乎】

8.7.3 信号量和条件锁的区别

条件锁，本质上还是锁，它的用途，还是围绕“共享资源”的。条件锁最典型的用途就是：**防止不停地循环去判断一个共享资源是否满足某个条件**。

比如还是买票的例子：我们除了买票的线程外，现在再加一个线程：如果票数等于零，那么就要挂出“票已售完”的牌子。这种情况下如果没有条件锁，我们就不得不在“挂牌子”这个线程里不断地 lock 和 unlock 而在大多数情况下票数总是不等于零，这样的结果就是：占用了很多 CPU 资源但是大多数时候什么都没做。

假如我们还有一个线程，是在票数等于零时向上级部门申请新的票。同理，问题和上面的一样。而如果有了条件锁，我们就可以避免这种问题，而且还可以一次性地通知所有被条件锁锁住的线程。

总之请记住：**条件锁，是为了避免绝大多数情况下都是“lock --> 判断条件 --> unlock”的这种很占资源但又不干什么事情的线程**。它和 Semaphore 的用途是不同的。

【semaphore 和 mutex 的区别？ - 二律背反的回答 - 知乎】

8.7.4 信号量和线程池的区别

虽然 Semaphore 的这种用法和线程池看上去很类似：都是在限制同时执行的线程数量，但是两者是有本质区别的。线程池是通过**用固定数量的线程去执行任务队列里的任务**，来达到**避免反复创建和销毁线程而造成的资源浪费**；而 Semaphore 并没有直接提供这种机制。

线程池用来控制实际工作的线程数量，通过线程复用的方式来减小内存开销。线程池可同时工作的线程数量是一定的，超过该数量的线程需进入线程队列等待，直到有可用的工作线程来执行任务。

使用 Seamphore，你创建了多少线程，实际就会有多少线程进行执行，只是可同时执行的线程数量会受到限制。但使用线程池，你创建的线程只是作为任务提交给线程池执行，实际工作的线程由线程池创建，并且实际工作的线程数量由线程池自己管理。

简单来说，线程池实际工作的线程是 work 线程，不是你自己创建的，是由线程池创建的，并由线程池自动控制实际并发的 work 线程数量。而 Seamphore 相当于一个信号灯，作用是对线程做限流，Seamphore 可以对你自己创建的的线程做限流（也可以对线程池的 work 线程做限流），Seamphore 的限流必须通过手动 acquire 和 release 来实现。

【了解信号量 Semaphore 和线程池的差异】

8.7.5 死锁

互斥锁和信号量虽然解决了数据完整性和状态同步的问题，但也可能引发死锁。所以要**尽可能保证每一个线程只能同时保持一个锁**，这样程序就不会被死锁问题所困扰。

死锁（Deadlock），死锁是指两个或两个以上的线程在执行过程中，由于竞争资源而造成的一种阻塞现象，若无外力作用，它们都将无法推进下去。死锁产生的 4 个必要条件：

1. **互斥：**一个资源同一时刻只允许一个线程进行访问。
2. **占有未释放：**一个线程占有资源，且没有释放资源。
3. **不可抢占：**线程占有的资源只能自己用完后释放，无法被抢占。
4. **循环等待：**两个或者两个以上的线程，本身拥有资源，不释放资源，并且同时尝试获得其他线程所持有的资源，这种资源的申请关系形成一个闭环的链条。

解决死锁的方法：

1. 预防死锁，**破坏产生死锁的必要条件中的至少一个**，比如给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）。
2. 避免死锁，**只允许不会产生死锁的进程申请资源**，避免死锁的代表性算法有银行家算法，避免死锁是在资源动态分配过程中进行的。
3. 死锁检测与解除，在检测到运行系统进入死锁，进行恢复。解除死锁的方式有剥夺资源和撤销进程。

【死锁的四个必要条件和解决办法】

8.7.6 死锁案例

定义两个 ArrayList,将他们都加上锁 A,B，线程 1,2，1 拿住了锁 A ，请求锁 B，2 拿住了锁 B 请求锁 A，在等待对方释放锁的过程中谁也不让出已获得的锁。

8.8 僵尸/孤儿/守护进程

僵尸进程：子进程在结束之后，释放掉其占用的绝大部分内存空间以及 cpu 等资源，但是会留下一个称为僵尸进程的数据结构(包含子进程的 pid)，等待父进程处理。这种情况下的僵尸进程是无害的(待所有的子进程结束后，父进程会统一向操作系统发送回收子进程 pid 的请求，或者使用 join()，其内部也拥有 wait()方法)，但是，如果父进程是一个死循环，不断的创造子进程，而又不发送回收请求，这就造成了大量的 pid 被占用

孤儿进程：

守护进程：

【python 之路---并发编程之进程&僵尸进程/孤儿进程/守护进程】

8.8.1 守护进程和守护线程

8.8.2 join() 和 setDaemon()

Python 多线程中经常用到 join() 和 setDaemon()：

join()：**在子线程完成运行之前，这个子线程的父线程将一直被阻塞**。比如主线程 A 中创建了子线程 B，并且在主线程 A 中调用了 B.join()，那么主线程 A 会在调用的地方阻塞，直到子线程 B 完成后，才可以接着往下执行。

setDaemon(True)：将线程声明为守护线程，**只要主线程完成了，不管子线程是否完成，都要和主线程一起退出**。比如主线程 A 中创建了子线程 B，并且在主线程 A 中调用了 B.setDaemon()，那么就把主线程 A 设置成了守护线程，

这时候要是主线程 A 执行结束了，就不管子线程 B 是否完成,一并和主线程 A 退出。**setDaemon(True)必须在 start() 方法调用之前设置，如果不设置为守护线程程序会被无限挂起**。

守护进程：

（1）守护进程会在主进程结束的时候立马结束

（2）守护进程要设置在 start 之前

（3）守护进程能不能再开启子进程，否则会报错。（错误：AssertionError: daemonic processes are not allowed to have children）

两者基本是相反的（join 主线程等子线程完事【横刀立马】，setDaemon 主线程管好自己就可以了，不等子线程完事【义无反顾】）。

【彻底理解 Python 多线程中的 setDaemon 与 join】

8.8.3 生产者消费者模式

生产者消费者模式，synchronized 锁住一个 LinkedList，一个生产者，只要队列不满，产后往里放，一个消费者只要队列不空，向外取，两者通过 wait()和 notify()进行协调，写好了会问怎样提高效率，最后会聊一聊消息队列设计精要思想及其使用。

第九章 设计模式

9.1 单例模式

单例一共有几种实现方式：饿汉、懒汉、静态内部类、枚举、双检锁，要是写了简单的懒汉式可能就会问：要是多线程情况下怎样保证线程安全呢，面试者可能说双检锁，那么聊聊为什么要两次校验，接着会问光是双检锁还会有什么问题，要是多线程情况下怎样保证线程安全呢，面试者可能说双检锁，那么聊聊为什么要两次校验，接着会问光是双检锁还会有什么问题。

第十章 正则

【正则表达式详解】

第十一章 单元测试

第十二章 疑问

我们知道 python3 中 zip 返回的是一个对象，想要获取列表需要用 list 转换，list 方法会返回一个列表：

```
if __name__ == '__main__':
    a, b = [1, 2], [3, 4]
    x = zip(a, b)
    y = list(x)
    print("y:", x, y)

    z = list(x)
    print('z:', x, z)

    # y: <zip object at 0x00000177F7DE2D00> [(1, 3), (2, 4)]
    # z: <zip object at 0x00000177F7DE2D00> []
```

但是好像只能转换一次？第二次输出的是空列表。