Python 全栈 400 之程序员必备算法练习

334 程序员要知道什么是算法？

我们一直在讲算法，算法，那么什么是一个算法呢？

算法就是用来解决特定问题的指令序列，这句话并不难理解，因为我们平时一直就在写代码，写这些代码当然不是徒劳的，是为解决某个特定问题，代码必然也是指令序列，所以问题出现了：我们平时写的代码也能叫做算法吗？

从算法的定义看，的确是这样，我们平时就是一直在写算法，只不过有些读者编写的算法代码偏向于业务逻辑，更多涉及前后端框架、数据持久化、架构等；

而有些读者编写的代码更多与计算机视觉和自然语言处理等领域相关，背后是机器学习、深度学习和优化理论相关的数学知识。

通过招聘网站上，我们也看到，前者更多被定义为软件工程师或架构师；而后者更多被定义为算法工程师。所以一般而言，算法工作需要涉及到数学理论，公式推导等。

因此，作为程序员，我们有必要了解算法到底包括哪些要素，为什么它要求具备一定的数学推导能力？

算法到底包括哪些要素，至今也没有明确的定义，但是通常来讲，包括的主要要素有如下：

- 算法会有确定输入和输出；
- 算法必须要是确定的，可以被描述为一个由基本操作组成的序列；
- 同时算法必须是可行的，运行有限次基本操作后必须能结束。那么怎么衡量有限次操作？这个就不太容易给出明确的定义，但是直觉告诉我们， 总不能指望一个程序运行100年后才结束，这种算法显然也是不可行的。
- 再有算法必须确保能够解决指定的问题，也就是必须是正确的，就这一点就比较让人头疼，因为某种程序上，只有给出严格的数学论证，才能证明算法的正确性。而不是跑10000次算法，都没有出现bug，我们就下定结论，这个算法是正确的。相比前者，此种方法显然缺少严谨性。

总结而言，算法主要基本要素包括：输入输出、能够分解为确定的基本操作、必须可行能在有生之年跑出结果、最后从数学上证明算法的正确性。

335 通过冒泡排序理解算法的 4 个基本要素

下面通过冒泡排序的案例，进一步加深对算法 4 个基本要素的理解。

冒泡排序的代码，我们不难编写出来，在此直接给出一版实现。前两行代码描述出我们待求解的问题，待排序的元素个数为 15，待排序的元素序列为 a, 且为 [10, 7, 11, 7, 18, 17, 9, 17, 0, 7, 3, 20, 20, 15, 9]

冒泡排序算法被定义为 bubble_sort

它的**输入参数**有 a, n, 也就是待求解的问题;

它的**输出** 为 ac, 也就是排序好的序列

复制

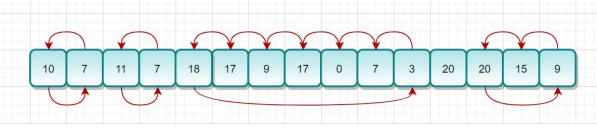
```
n = 15
a = [10, 7, 11, 7, 18, 17, 9, 17, 0, 7, 3, 20, 20, 15, 9]

def bubble_sort(a,n):
    ac = a.copy()
    swap_size = 1
    while swap_size > 0:
        swap_size = 0
        for i in range(n-1):
            if ac[i] > ac[i+1]:
                ac[i],ac[i+1] = ac[i+1],ac[i]
                swap_size += 1
        n-=1
    return ac
```

这版冒泡排序的实现包括的基本操作，排序操作只要满足交换操作次数大于0的条件(`while swap_size > 0`)，就会一轮一轮的进行下去。每轮中如果前一个元素 `i` 大于后一个元素 `i+1`（`ac[i] > ac[i+1]`），就会发生一次交换操作(`ac[i],ac[i+1] = ac[i+1],ac[i]`)，以上就是冒泡排序主要**操作步骤**，这些操作都是**确定的**，没有歧义的，没有模棱两可的。

下面考察它是否一定会在有限次运行后结束，就这版冒泡排序实现代码，它的运行次数为：n + (n-1) + ...+ 1，等于 n*(n-1) / 2，如果采用大O标记，那么它的运行次数趋近于 O(n^2)，也就是算法的最坏时间复杂度为 O(n^2). 不懂这些符号标记的读者不要着急，今天后半部分我们会再次解释经常在算法书籍上看到的大O标记。至此下结论：问题规模为 n 时算法时间复杂度趋于 O(n^2)， 而 O(n^2) 时间复杂度属于多项式级别，是能够接受的一种时间复杂度算法。尽管 n 值很大时，n^2 是一个很大的值，但是它仍然也被定义为一个**运行有限次**就会结束的算法。

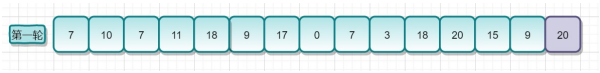
关于冒泡排序的正确性证明，我们借助示意图解释更容易些。如下图，这是我们待排序的序列：

第一轮排序中，红色箭头表示发生元素交换操作，如下所示元素 7 交换到元素 10 所在位置，10 交换到元素 7 所在位置，然后 10 与 11 比较未发生交换，11 与 7 比较发生一次交换，11与 18 不发生元素交换，18 与 17发生交换，18 与 9 发生交换，一直发生元素交换，直到遇到元素 20 不再发生交换，所以元素 18 被交换到元素 3 所在位置，依次类推。

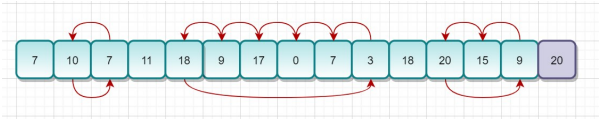


第一轮交换完成后的示意图如下，第二个等于 20 的元素被交换到列表最后，经过一轮比较和交

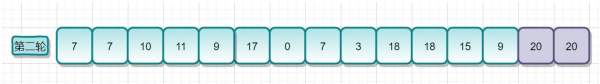
换后，冒泡出序列的最大元素并归位，下一轮比较后只需到 n-1 处即可。



第二轮比较过程同第一轮相似：



第二轮比较后的结果如下，第一个等于 20 的元素也归位，下一轮比较只需到 n-2 处即可。



当进行到第 k 轮时，一定能确保 k 个元素就位，这是确信无疑，不会有任何随机性的事件。

与此同时，进行到第 k 轮时，待排序的序列长度就会变为 n-k，也就是问题的规模会缩减到 n-k，进一步说，问题规模具备严格的单调性，会单调递减。可以预见算法至多经过 n 轮扫描后，算法必然会终止，且能给出正确的排序结果。

336 如何理解算法的有穷性？

算法的有穷性指算法会经过有限次运算后而终止，如冒泡排序在至多 O(n^2) 后会终止。算法一定要在有限次运行后终止，表面上看这个再自然不过，但是算法的有穷性有时却不好判定。

比如考虑下面这个算法，当 n 为 不大于 2 的整数时，直接添加到列表 a 中；当 n 大于 2，且为偶数时，折半后添加到列表 a 中，并折半后递归调用；当 n 大于 2，且为奇数时，3*n + 1 后添加到列表 a 中，并递归调用。

复制

```
def f(n):
    a = []
    if n <= 2:
        a.append(1)
    elif n %2 == 0:
        a.append(n//2)
        a.extend(f(n//2))
    else:
        a.append(n*3+1)
        a.extend(f(n*3 + 1))
    return a
```

n 为奇数时，3*n + 1 一定为偶数，这就意味着本次将 n 放大后，下一次一定会折半减小 n，那么有一个问题，n 越大，递归次数就越多吗？或者 n 与递归次数成正比吗？

为此进行一个实验模拟，取 n 等于 1 到 50，分别调用 f 函数，得到每次调用的递归次数。

复制

```
alen = []
for i in range(50):
    a = f(i)
    print('f(%d) length: %d'%(i,len(a)))
    alen.append(len(a))
print(alen)
```

得到结果，递归迭代次数最多为 111次，发生在 n 为 27 时，很显然 n 不与递归次数成正比。

复制

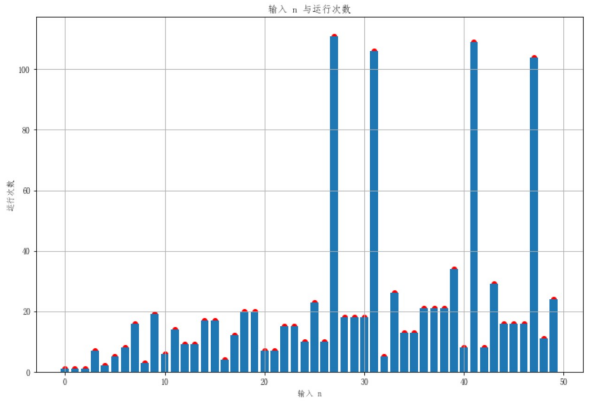
```
[1, 1, 1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7, 7
, 15, 15, 10, 23, 10, 111, 18, 18, 18, 106, 5, 26, 13, 13, 21, 21, 21, 34
, 8, 109, 8, 29, 16, 16, 16, 104, 11, 24]
```

再直观展示 n 与递归次数的关系图：

复制

```
import matplotlib.pyplot as plt
# 显示中文
from pylab import mpl
mpl.rcParams['font.sans-serif'] = ['FangSong']
mpl.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号'-'显示为方块的
问题

plt.figure(figsize=(12,8))
plt.bar(range(50),alen)
plt.scatter(range(50),alen,c='r')
plt.grid()
plt.title('输入 n 与运行次数')
plt.xlabel('输入 n')
plt.ylabel('运行次数')
plt.show()
```

柱状和散点图如下，n 与迭代次数的关系是很复杂的，不是简单的线型关系。



值得注意的是，这个问题目前都无法证明算法是否运行有限次，也就是无法得出问题规模 n 与所

需迭代次数的数学表达式。因此以上 f 函数不能称为一个算法，因为它无法证明具有有穷性，也无法否定不具有有穷性。

337 评价算法好坏之重要的大 O 标记法

我们知道什么是算法，知道算法的必备主要要素后，下一步该学习如何衡量一个算法是否为高效的。

度量算法是否高效，首要的一个标准便是算法的运行时间。解决同样一个问题，算法 A 运行时长比算法 B 更短，那么算法 A 的时间复杂度就比算法 B 要好，因此从时间复杂度角度衡量，算法 A 更高效。实际中，算法的时间复杂度考量往往也是放在首位，主要考虑的指标之一。

那么怎么定量算法的时间复杂度呢？

还是拿冒泡排序来说，它的运行次数为 `n*(n-1)/2` ，然后我们说这个算法的时间复杂度为 `n*(n-1)/2` ，然而这个标记还是稍显复杂，其实如果按照这种标记方法，更复杂的算法的时间复杂度可能为如下：

复制
```
10*n^4 + n^3 + 5*n^2 + 3*n + 10
```

按照此种写法，同行交流算法时间复杂度就会十分费劲。后来计算机科学家引入一种极简的算法标记，`大 O` 标记法。

此方法会放大时间复杂度，也就是会求出算法的最坏时间复杂度，这恰好也是我们最关心的，并且放大后的标记就简单太多。

比如，上面的 `n*(n-1)/2` 使用大 O 标记法后，变为 `O(n^2)` ；而 `10*n^4 + n^3 + 5*n^2 + 3*n + 10` 更是一步到位后变为 `O(n^4)` .

大 O 标记关注的是趋势，随着 n 趋向很大时算法的操作时间。所以，`n^2 /2 - n /2` 就能放大为 `n^2 /2`, 进一步放大为 `n^2` . 同理， `10*n^4 + n^3 + 5*n^2 + 3*n + 10` 就会被放大为 `10*n^4 + n^3*n + 5*n^2 *n^2 + 3*n*n^3 + 10 * n^4` 也就是 `29* n^4` ，大 O 标记前的常数项可以忽略，所以最后时间复杂度的大 O 标记值为 `O(n^4)` .

338 算法时间复杂度之 O(1) 例子

O(1) 时间复杂度，比如操作一步到位这是常数级别的时间复杂，即为 O(1).

复制
```
a = 100 / 2 + 101 * 10 + 5
a
```

339 算法时间复杂度之 O(logn) 例子

梦寐以求的另一种算法复杂度为 `O(logn)` ，如下操作，每次 n 会被除以 3，遍历次数等于 以 3 为底 log n 次。在大 O 标记体系中，以 3 为底和以 5 为底没有区别，所以统一标记为 `O(logn)`

复制
```
def f(n):
    while n > 0:
        print(n)
        n //= 3
```

340 算法时间复杂度之 o(n) 例子

线型复杂度也是很理想的情况，如下面的操作，每次遍历，n 减去 3， 这样总共遍历(n-1)/3 + 1 次后算法终止，根据大 O 标记法，算法的时间复杂为 O(n).

复制
```
def f(n):
    while n > 0:
        print(n)
        n -= 3
```

341 算法时间复杂度之 O(nlogn) 例子

下一讲排序算法中就有时间复杂度为 `nlogn` ，此复杂度的算法也是较为高效的。如下面所示的两层循环，复杂度便是 `nlogn` . 外层循环的运行次数为 `n` , 里层循环的运行次数为 `logn` 次，所以一共需要 `nlogn` 次。

复制
```
def f(n):
    for i in range(n):
        for j in range(1,n,2):
            print(i*j)
```

342 算法时间复杂度之 O(n^2) 例子

O(n^2) 是 多项式时间复杂度的代表，此类算法的时间复杂度已经难以划分到高效算法集合中，它只能是问题的有效解，而不是高效解。如下两层 for 循环，时间复杂度就是 `O(n^2)` .

复制
```
def f(n):
    for i in range(n):
        for j in range(n):
            print(i*j)
```

343 算法时间复杂度之 O(2^n) 例子

时间复杂度为 `O(2^n)` 的算法是指数级增长的，此类复杂度下求解的问题往往都是难题，因为随着问题规模 n 的增长，指数级的增长速度是惊人的。

例如经典的旅行商问题，商人要去 n 个地方拜访，如何规划拜访顺序才能使得旅行距离最短。如果仅拜访肉眼可见的两三个地方时，我们还能穷举所有拜访的组合，进而找到最短路径。

但是当问题规模 n 变大时，目前所有的计算机资源总和都难以在有限的时间里计算出最优的最短路径，这类问题的时间复杂度都为指数级，属于 NP 难问题。

345 各种常见复杂度的比较

以上我们讨论了常见的 6 种时间复杂度，其中 O(logn), O(n) 和 O(nlogn) 是相对高效的算法复杂度，O(n^2) 属于有效解的时间复杂度，但是指数级 O(2^n) 往往是不可行的求解算法时间复杂

度。

下面结合图形直观的观察随着问题规模 n ，求解时间复杂度的增长趋势。如下所示为绘制时间复杂度所使用的的绘制图形代码：

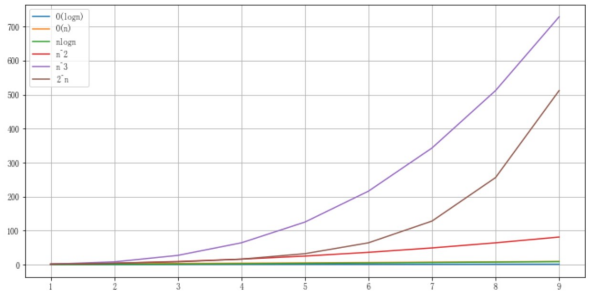
复制

```
import numpy as np

def fo(n):
    plt.figure(figsize=(12,6))
    plt.plot(n,np.log10(n),label="O(logn)")
    plt.plot(n,n,label="O(n)")
    plt.plot(n,n*np.log10(n),label="nlogn")
    plt.plot(n,np.power(n,2),label="n^2")
    plt.plot(n,np.power(n,3),label="n^3")
    plt.plot(n,np.power(2,n),label="2^n")
    plt.legend()
    plt.grid()
    plt.show()
```

调用上面方法，绘制问题规模为 10 时的时间复杂度对比图，看到指数级的时间复杂增长趋势还未达到最快，低于 O(n^3) ，因为此时求解问题规模较小。

复制

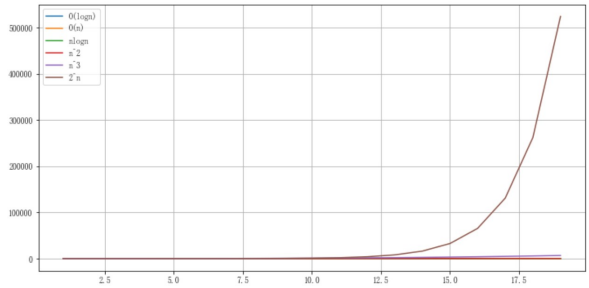
```
n = np.array(list(range(1,10)))
fo(n)
```



但是仅仅当问题规模变为 20 时，指数级的增长之快就一眼可见，从不到 800 迅速增长到 50 万之多。

复制

```
n = np.array(list(range(1,20)))
fo(n)
```



今天一起同大家刷刷 LeetCode 中最经典的练习题，它们代表几大类常用的数据结构和基本算法思想。作为程序员必须要熟练掌握，因为它们经常在面试中被考察到。就算不为面试，掌握它们也会有助于我们编写出更高质量的代码，提升自己的计算机思维。

346 链表反转图文案例

链表作为经典的数据结构，应用极其广泛，列表、二叉树等都会看到它的身影，面试也是经常被问道。不要小瞧链表，老鸟也会经常翻车，所以不要掉以轻心，需要格外小心。

链表 ListNode 对象通常被定义为如下，val 为 ListNode 对象的值域，next 指向下一个 Node，下一个 Node 又指向下下个 Node，从而形成一条链式结构。

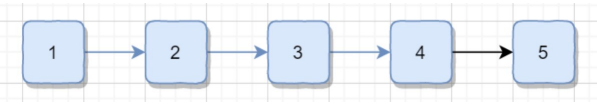
复制

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

链表反转是指反转后原最后 Node 变为头节点，并指向原倒数第二个节点，依次类推。

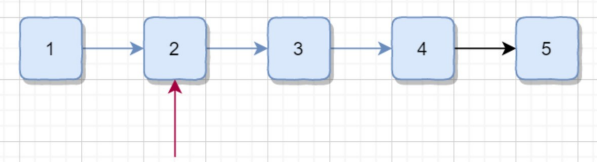
那么，如何实现链表反转？代码其实只有几行，关键如何利用链表的特点构思出链表的反转，下面解释思考的过程，同时理解链表的精髓所在。

首先给出递归版本的求解代码。假设原链表序列如下图所示：

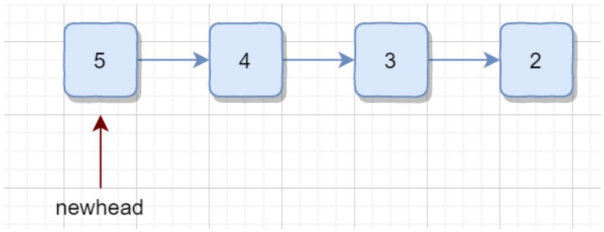


求解代码 reverseList 函数的前两行表示边界条件或递归终止的条件是 head 节点为 None 或 head 指向的下一个节点为 None.

令变量 tmp 指向链表的第二个节点，如下图所示，然后递归调用 reverseList 函数，反转 tmp 指向的链表

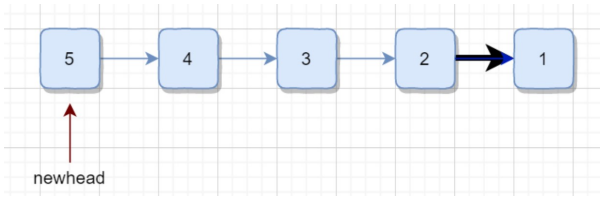


反转后的结果如下图所示，newhead 指向反转后链表的头部：



那么，如何将值等于 1 的节点经过 O(1) 时间复杂度链接到 newhead 链表的最后呢？

这就用到已经标记下来的 tmp 节点，因为反转后 tmp 节点的指向不正是等于 1 的节点吗！所以 tmp.next = head 操作实现建立值等于2的节点和值等于1的节点的链接：



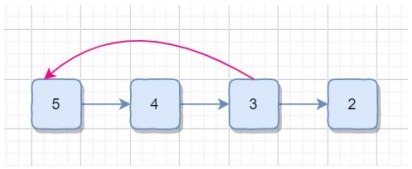
但是务必要小心，此时值为1的节点还是会指向其他节点，所以务必将其next值置为 None.

到此完成递归版链表反转的解释。

复制

```
class Solution:
    def reverseList(self, head):
        if head is None or head.next is None:
            return head
        tmp = head.next
        newhead = self.reverseList(tmp)
        tmp.next = head
        head.next = None
        return newhead
```

链表反转的迭代版本应该怎么写，平时链表使用多的读者会深有体会，链表操作最容易犯错的地方，就是有意无意中形成一个环(cycle)，如下 5->4->3->5 的环结构：

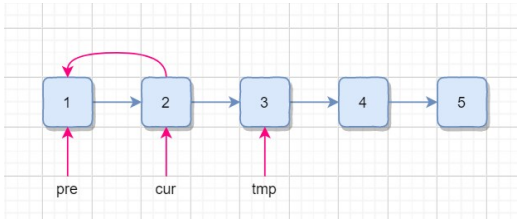


如下链表反转的迭代版，一不小心就写出一个带环的链表结构。下面借助示意图分析这个环结构如何形成的。

复制

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre = head
        cur = head.next
        while cur:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        return pre
```

当执行到 `cur.next = pre` 时，已经形成一个 1->2->1 的环形结构。

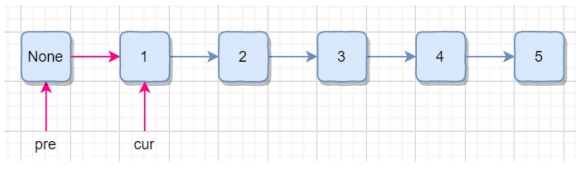


要想避免入坑，使用链表需要慎之又慎。`pre` 和 `cur` 的初始指向非常重要，此处只需对以上代码稍作修改，仅仅修改pre 和 cur 的初始指向，便得到链表反转的迭代版本。

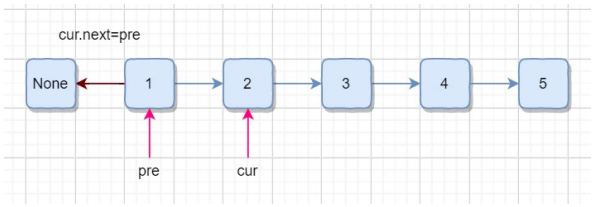
复制

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

修改后的代码，链表反转的迭代步骤示意图。初始状态如下图所示，pre 指向一个虚拟的 None 节点，这是关键一步，保证避免环的存在。



迭代步骤中，先保存住 tmp 节点，通过 `cur.next = pre` 建立 1 节点和 None 节点的指向，然后令 pre 和 cur 的指向分别前移一位，至此完成一轮迭代。迭代时的临时变量为 tmp ，起到修改 cur.next 值前缓存 cur 节点的 next 域的作用。



347 二叉树图文案例

链是单向的，节点的 next 域指向下一个节点。二叉树在单向的链表基础上，又增加出一个维度，它的节点一次指向两个节点，并称它们为儿子节点。以此递归，形成一棵树，并被称作二叉树，也就不难理解了。链表和二叉树实则神态上相似，前者为一维指向，后者为二维指向。

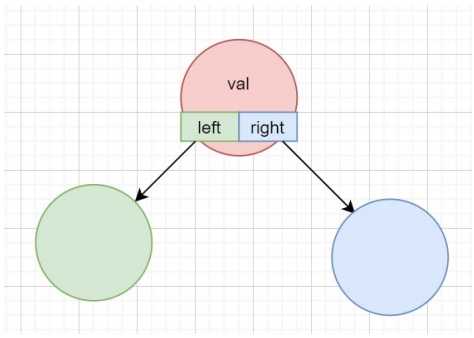
因此，在实际使用二叉树时，以上我们讲解的链表思维都能应用到二叉树中，并且二叉树因为指向两个节点，求解的时间性能往往更优于链表。

二叉树的应用案例，我们先从一道二叉树的路径求和题目开始。

二叉树的节点对象定义为：

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

二叉树示意图如下，父节点对象有三个值，val, left, right 分别为当前节点的值，指向左儿子、右儿子的变量。



读者们请注意，val 这个取值不仅仅局限于我们通常理解的一个数值，它可以扩展为更加具有表达力和想象力的其他各种对象，包括自定义对象。

本题目中求解二叉树从根节点(最开始的头节点)到叶节点(既没有左孩子也没有右孩子的节点)的路径求和等于22，返回所有满足此条件的路径。如 5->4->11->2 求和等于22，就是满足条件的一条路径。

```
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \    / \
7    2  5   1
```

如链表的解题思路相似，二叉树的解题构思一般也是先前进一步，使得问题的求解规模 减少一点，然后递归调用，注意递归调用的终止条件，求得满足条件的解。

完整代码如下，下面分布介绍如何构思出这些代码。

复制
```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> List[List[int]]:
        paths = []
        if root is None:
            return paths
        if root.val == sum and root.left is None and root.right is None:
            paths.append([root.val])
            return paths

        ls = self.pathSum(root.left,sum - root.val)
        for l in ls:
            l.insert(0,root.val)
        if len(ls)>0:
            paths.extend(ls)

        rs = self.pathSum(root.right,sum-root.val)
        for r in rs:
            r.insert(0,root.val)
        if len(rs)>0:
            paths.extend(rs)

        return paths
```

`pathSum` 函数的两个参数 `root`，路径和 `sum` 值，边界条件也是递归调用的终止条件为如下。注意，`if root.val == sum and root.left is None and root.right is None:` 中 `root.left is None and root.right is None` 条件非常重要，确保已搜索到叶节点，如果遗漏此条件，可能搜索出某条从根节点到非叶结点的路径，这与题目中要求的从根节点到叶结点的路径相违背。

复制

```
        paths = []
        if root is None:
            return paths
        if root.val == sum and root.left is None and root.right is None:
            paths.append([root.val])
            return paths
```

调用 `pathSum` 一次后，如果以上 2 个边界条件都不满足，则表明需要继续向下搜索，此时剩余部分的路径总和为 sum-`root.val`，分别探索以 `root.lef` 节点为根节点的子树对应路径，表达为代码就是：`rs = self.pathSum(root.right,sum-root.val)`，rs 等于剩余部分的路径，然后需要插入它的父节点 root 到索引 0 处；搜索 以 `root.right` 为根节点的子树原理相似。最后 paths 就是所有可能的路径。

348 动态规划案例

动态规划(简写为 DP)作为比较难的算法代表，此专栏在前几天单独有一讲。实话讲，这仍然不够，DP 非常灵活，一旦求解问题找到 DP 的解，通常都会带来时间复杂度的大幅改善。苦难是有的，带来的好处也非常明显，所以想培养对 DP 的敏锐度，多做一些题目是必要的。

下面再分析一道使用DP求解的问题：连续最大子列表的乘积，元素类型为整型。

复制

```
输入: [2,3,-2,4]
输出: 6
解释: [2,3] 子数组连续乘积最大为 6
```

如果元素都为非负，求解相对容易，因为是连续子数组，所以 `max(it, it * tmp)` 意味着要么重新从 `it` 开始，要么一直在 `tmp` 上累积：

复制

```
class Solution(object):
    def maxProduct(self, nums):
        ret,tmp = nums[0],nums[0]
        for it in nums[1:]:
            tmp = max(it, it * tmp)
            ret = max(tmp,ret)
        return ret
```

如果列表中也存在负数，相比上面代码需要考虑当前遍历到的元素 `it` 的正负，如果为负数则 tmp_max 被赋值为当前得到的最小累积 `tmp_min`(注意它可能为非负或为负)，如果 `tmp_min` 为非负，则乘以 it ，大概率不会得到一个新的最大值，所以运行到 `ret = max(tmp_max, ret)` 时，会抛弃 `tmp_max`，但是如果 `tmp_min` 为负，乘以 it 后有可能得到一个新的最大值。

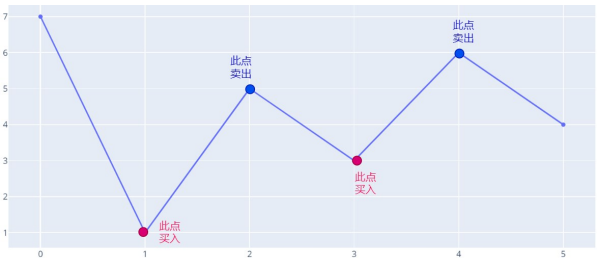
复制

```
class Solution(object):
    def maxProduct(self, nums):
        ret, tmp_min, max_prod = [nums[0]]*3
        for it in nums[1:]:
            if it < 0:
                tmp_min,tmp_max = tmp_max,tmp_min
            tmp_max = max(it, it * tmp_max)
            tmp_min = min(it, it * tmp_min)
            ret = max(tmp_max, ret)
        return ret
```

349 贪心使用案例

贪心算法有时得到的只是可行解，而不是最优解。但在满足某些假定条件下，能获得最优解。如下买卖股票的经典题目，在满足假定条件：卖股票发生前只能先买一次股票下，使用贪心求解便能求出最大收益。

`[7,1,5,3,6,4]` 的最大收益为 7，如下图所示。



绘图代码：

复制

```
import plotly.graph_objects as go
fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x=[0, 1, 2, 3, 4, 5],
        y=[7,1,5,3,6,4]
    ))
fig.show()
```

也就是贪婪的找到所有相邻的呈现上升的点对，并求累加和就是最优解，也就是最大收益。

代码如下所示：

复制

```
class Solution(object):
    def maxProfit(self, prices):
        pair = zip(prices[:-1],prices[1:])
        return sum( x1 - x0 for x0, x1 in pair if x0 < x1 )
```

350 回溯 + DFS 应用案例

回溯就是带有规律的来回尝试搜寻完整解结构的过程，所谓的搜索规律，常见的一种就是结合 DFS 深度优先搜索。比如求解集合的所有子集。

已知列表 `[1,2,3]`，里面没有重复元素，它的所有子集为：

复制

```
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

如何使用回溯+DFS，构思出这个问题的解呢？关键是定义的 `dfs` 方法，第一个参数 `e` 表示一个子集，`beg` 表示待搜索路径的起始位置，`digits` 表示当前子集包括的元素个数。

复制

```
class Solution:
    def subsets(self, nums):
        if nums is None:
            return None
        ret, e = [[]], []

        def dfs(e, beg, digits):
            if len(e) == digits: # dfs 递归的终止条件
                ret.append(e.copy()) # 满足条件表示找到一个子集
                return
            while digits <= len(nums): # 求解框架的第一层
                for i in range(beg, len(nums)): # 求解框架的第二层
                    e.append(nums[i])
                    dfs(e, i+1, digits) # 递归:头部分为e, 路径起始位置i+1, 元素个数digits
                    e.remove(e[-1]) # 下面五行代码实现对 e 部分的出栈控制
                if len(e) == 0:
                    digits += 1
                else:
                    return

        dfs(e, 0, 1) # 调用 dfs
        return ret
```

求解过程的解释说明，请参考代码注释。

351 字符串简单案例

字符串类型题目，比如求由空格分隔的字符串的最后一个单词的字符个数：

复制

```
class Solution:
    def lengthOfLastWord(self, s: str) -> int:
        if len(s)==0:
            return 0
        return len(s.strip().split(' ')[-1])
```

需要注意如果输入的字符串为 `'a '` 需要先使用 `strip` 函数 去掉前后的空格。

352 位运算应用案例

常见的位运算符号 `&` 位与，`|` 位或，`^` 异或，介绍一个使用异或求解的问题，列表中只有1个元素出现1次，其他都出现2次，找出出现1次的元素。

复制

```
class Solution(object):
    def singleNumber(self, nums):
        a = 0
        for i in nums:
            a ^= i
        return a
```

353 栈模拟队列的案例

栈先进后出，队列先进先出，两者可以相互模拟，下面我们使用栈模拟队列，用 head, tail 分别指向 s 的头和尾，以实现队列的 push, pop, peek 都为 O(1) 时间复杂度。下面假定不会在空队列中使用 pop, peek 操作。

复制

```
class MyQueue:

    def __init__(self):
        """
        初始化
        """
        self.s = [] # 模拟栈:先进后出
        self.head = 0 # 头索引
        self.tail = 0 # 尾索引

    def push(self, x: int) -> None:
        """
        Push 元素到队列的尾部
        """
        self.s.append(x)
        self.tail += 1


    def pop(self) -> int:
        """
        移除并返回队列的头元素
        """
        pe = self.s[self.head]
        self.head += 1
        return pe

    def peek(self) -> int:
        """
        返回队列的头元素
        """
        return self.s[self.head]


    def empty(self) -> bool:
        """
        返回队列是否为空
        """
        return self.head == self.tail
```

使用队列：

复制

```
obj = MyQueue()
obj.push(x)
p2 = obj.pop()
p3 = obj.peek()
p4 = obj.empty()
```

下一章

互动评论

说点什么

评论

The Scrapper 1 个月前

后面的算法等回头再看看

鼓掌

存

1