

---

# Table of Contents

Introduction	1.1
001 Two Sum	1.2
002 Add Two Numbers	1.3
003 Longest Substring Without Repeating Characters	1.4
004 Median of Two Sorted Arrays	1.5
005 Longest Palindromic Substring	1.6
006 ZigZag Conversion	1.7
007 String to Integer (atoi)	1.8
008 Reverse Integer	1.9
009 Palindrome Number	1.10
010 Regular Expression Matching	1.11
011 Container With Most Water	1.12
012 Integer to Roman	1.13
013 Roman to Integer	1.14
014 Longest Common Prefix	1.15
015 3Sum	1.16
016 3Sum Closest	1.17
017 Letter Combinations of a Phone Number	1.18
018 4Sum	1.19
019 Remove Nth Node From End of List	1.20
020 Valid Parentheses	1.21
021 Merge Two Sorted Lists	1.22
022 Generate Parentheses	1.23
023 Merge k Sorted Lists	1.24
024 Swap Nodes in Pairs	1.25
025 Reverse Nodes in k-Group	1.26
026 Remove Duplicates from Sorted Array	1.27

---

027 Remove Element	1.28
028 Implement strStr()	1.29
029 Divide Two Integers	1.30
030 Substring with Concatenation of All Words	1.31
031 Next Permutation	1.32
032 Longest Valid Parentheses	1.33
033 Search in Rotated Sorted Array	1.34
034 Search for a Range	1.35
035 Search Insert Position	1.36
036 Valid Sudoku	1.37
037 Sudoku Solver	1.38
038 Count and Say	1.39
039 Combination Sum	1.40
040 Combination Sum II	1.41
041 First Missing Positive	1.42
042 Trapping Rain Water	1.43
043 Multiply Strings	1.44
044 Wildcard Matching	1.45
045 Jump Game II	1.46
046 Permutations	1.47
047 Permutations II	1.48
048 Rotate Image	1.49
049 Group Anagrams	1.50
050 Pow(X, n)	1.51
051 N-Queens	1.52
052 N-Queens II	1.53
053 Maximum Subarray	1.54
054 Spiral Matrix	1.55
055 Jump Game	1.56
056 Merge Intervals	1.57

---

---

057 Insert Interval	1.58
058 Length of Last Word	1.59
059 Spiral Matrix II	1.60
060 Permutation Sequence	1.61
061 Rotate List	1.62
062 Unique Paths	1.63
063 Unique Paths II	1.64
064 Minimum Path Sum	1.65
065 Valid Number	1.66
066 Plus One	1.67
067 Add Binary	1.68
068 Text Justification	1.69
069 Sqrt(X)	1.70
070 Climbing Stairs	1.71
071 Simplify Path	1.72
072 Edit Distance	1.73
073 Set Matrix Zeroes	1.74
074 Search a 2D Matrix	1.75
075 Sort Colors	1.76
076 Minimum Window Substring	1.77
077 Combinations	1.78
078 Subsets	1.79
079 Word Search	1.80
080 Remove Duplicates from Sorted Array II	1.81
081 Search in Rotated Sorted Array II	1.82
082 Remove Duplicates from Sorted List II	1.83
083 Remove Duplicates from Sorted List	1.84
084 Largest Rectangle in Histogram	1.85
085 Maximal Rectangle	1.86
086 Partition List	1.87

---

---

087 Scramble String	1.88
088 Merge Sorted Array	1.89
089 Gray Code	1.90
090 Subsets II	1.91
091 Decode Ways	1.92
092 Reverse Linked List II	1.93
093 Restore IP Addresses	1.94
094 Binary Tree Inorder Traversal	1.95
095 Unique Binary Search Trees II	1.96
096 Unique Binary Search Trees	1.97
097 Interleaving String	1.98
098 Validate Binary Search Tree	1.99
099 Recover Binary Search Tree	1.100
100 Same Tree	1.101
101 Symmetric Tree	1.102
102 Binary Tree Level Order Traversal	1.103
103 Binary Tree Zigzag Level Order Traversal	1.104
104 Maximum Depth of Binary Tree	1.105
105 Construct Binary Tree from Preorder and Inorder Traversal	1.106
106 Construct Binary Tree from Inorder and Postorder Traversal	1.107
107 Binary Tree Level Order Traversal II	1.108
108 Convert Sorted Array to Binary Search Tree	1.109
109 Convert Sorted List to Binary Search Tree	1.110
110 Balanced Binary Tree	1.111
111 Minimum Depth of Binary Tree	1.112
112 Path Sum	1.113
113 Path Sum II	1.114
114 Flatten Binary Tree to Linked List	1.115
115 Distinct Subsequences	1.116
116 Populating Next Right Pointers in Each Node	1.117

---

---

117 Populating Next Right Pointers in Each Node II	1.118
118 Pascal's Triangle	1.119
119 Pascal's Triangle II	1.120
120 Triangle	1.121
121 Best Time to Buy and Sell Stock	1.122
122 Best Time to Buy and Sell Stock II	1.123
123 Best Time to Buy and Sell Stock III	1.124
124 Binary Tree Maximum Path Sum	1.125
125 Valid Palindrome	1.126
126 Word Ladder II	1.127
127 Word Ladder	1.128
128 Longest Consecutive Sequence	1.129
129 Sum Root to Leaf Numbers	1.130
130 Surrounded Regions	1.131
131 Palindrome Partitioning	1.132
132 Palindrome Partitioning II	1.133
133 Clone Graph	1.134
134 Gas Station	1.135
135 Candy	1.136
136 Single Number	1.137
137 Single Number II	1.138
138 Copy List with Random Pointer	1.139
139 Word Break	1.140
140 Word Break II	1.141
141 Linked List Cycle	1.142
142 Linked List Cycle II	1.143
143 Reorder List	1.144
144 Binary Tree Preorder Traversal	1.145
145 Binary Tree Postorder Traversal	1.146
146 LRU Cache	1.147

---

---

147 Insertion Sort List	1.148
148 Sort List	1.149
149 Max Points on a Line	1.150
150 Evaluate Reverse Polish Notation	1.151
151 Reverse Words in a String	1.152
152 Maximum Product Subarray	1.153
153 Find Minimum in Rotated Sorted Array	1.154
154 Find Minimum in Rotated Sorted Array II	1.155
155 Min Stack	1.156
156 Binary Tree Upside Down	1.157
157 Read N Characters Given Read4	1.158
158 Read N Characters Given Read4 II - Call multiple times	1.159
159 Longest Substring with At Most Two Distinct Characters	1.160
160 Intersection of Two Linked Lists	1.161
161 One Edit Distance	1.162
162 Find Peak Element	1.163
163 Missing Ranges	1.164
164 Maximum Gap	1.165
165 Compare Version Numbers	1.166
166 Fraction to Recurring Decimal	1.167
167 Two Sum II - Input array is sorted	1.168
168 Excel Sheet Column Title	1.169
169 Majority Element	1.170
170 Two Sum III - Data structure design	1.171
171 Excel Sheet Column Number	1.172
172 Factorial Trailing Zeroes	1.173
173 Binary Search Tree Iterator	1.174
174 Dungeon Game	1.175
175 Combine Two Tables	1.176
176 Second Highest Salary	1.177

---

---

177 Nth Highest Salary	1.178
178 Rank Scores	1.179
179 Largest Number	1.180
180 Consecutive Numbers	1.181
181 Employees Earning More Than Their Managers	1.182
182 Duplicate Emails	1.183
183 Customers Who Never Order	1.184
184 Department Highest Salary	1.185
185 Department Top Three Salaries	1.186
186 Reverse Words in a String II	1.187
187 Repeated DNA Sequences	1.188
188 Best Time to Buy and Sell Stock IV	1.189
189 Rotate Array	1.190
190 Reverse Bits	1.191
191 Number of 1 Bits	1.192
192 Word Frequency	1.193
193 Valid Phone Numbers	1.194
194 Transpose File	1.195
195 Tenth Line	1.196
196 Delete Duplicate Emails	1.197
197 Rising Temperature	1.198
198 House Robber	1.199
199 Binary Tree Right Side View	1.200
200 Number of Islands	1.201

---

# LeetCode Python

LeetCode的解题思路，代码是Python3版本。



# LeetCode解题之Two Sum

---

## 原题

给一个int型数组，要求找出其中两个和为特定值的数的坐标。

注意点：

- 返回的坐标一要比坐标二小
- 最小的坐标是1，不是0

例子：

输入: numbers={2, 7, 11, 15}, target=9 输出: index1=1, index2=2

## 解题思路

第一遍遍历整个数组，用map记录数值和它的坐标，第二遍遍历数组，判断（目标数字-当前数字）是否在map中，如果在，且它的下标与当前数字的下标不相同，则说明存在这两个数，返回坐标。

## AC源码

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        hash_map = {}
        for index, value in enumerate(nums):
            hash_map[value] = index
        for index1, value in enumerate(nums):
            if target - value in hash_map:
                index2 = hash_map[target - value]
                if index1 != index2:
                    return [index1 + 1, index2 + 1]
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode解题之Add Two Numbers

---

## 原题

定义这样的一个链表，链表的每个节点都存有一个0-9的数字，把链表当成数字，表头为高位，表尾为地位。如1->2->3表示321，现在要对两个这样的链表求和。

注意点：

- 数字的高低位，应该从从地位向高位进位
- 有多种情况要考虑，如链表长度是否相等、是否进位等

例子：

输入: (2 -> 4 -> 3) + (5 -> 6 -> 4) 输出: 7 -> 0 -> 8

## 解题思路

把两个链表当成是相同长度的，短的那个想象成后面都是0；如果进位的话，在初始化下一个节点的时候将其赋值为1即可，所以在计算当前节点的值时要加上自己本来的值。

## AC源码

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

# Define this to check if it works well
def myPrint(self):
    print(self.val)
    if self.next:
        self.next.myPrint()
```

```
class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        result = ListNode(0);
        cur = result;
        while l1 or l2:
            cur.val += self.addTwoNodes(l1, l2)
            if cur.val >= 10:
                cur.val -= 10
                cur.next = ListNode(1)
            else:
                # Check if there is need to make the next node
                if l1 and l1.next or l2 and l2.next:
                    cur.next = ListNode(0)
            cur = cur.next
            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next
        return result

    def addTwoNodes(self, n1, n2):
        if not n1 and not n2:
            # This cannot happen, ignore it
            None
        if not n1:
            return n2.val
        if not n2:
            return n1.val
        return n1.val + n2.val

if __name__ == "__main__":
    list = ListNode(9)
    list.next = ListNode(8)
```

```
print(Solution().addTwoNumbers(list, ListNode(1)).myPrint())
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Longest Substring Without Repeating Characters

---

### 原题

找出一个字符串的最长字符串，要求该字符串中没有重复的字符。

注意点：

- 考虑空字符等特殊情况

例子：

输入: "abcabcbb" 输出: 3

输入: "bbbbbb" 输出: 1

### 解题思路

将没有重复字符的子串称为目标字符串。遍历字符串的所有字符，记录下每个字符最近出现的下标位置，如果该下标比目标字符串的起始下标大，说明目标字符串中已经有该字符了，刷新目标字符串的起始坐标。找出所有目标串的最大长度就是题目要求的答案。

### AC源码

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        if not s:
            return 0
        if len(s) <= 1:
            return len(s)
        locations = [-1 for i in range(256)]
        index = -1
        m = 0
        for i, v in enumerate(s):
            if (locations[ord(v)] > index):
                index = locations[ord(v)]
            m = max(m, i - index)
            locations[ord(v)] = i
        return m

if __name__ == "__main__":
    assert Solution().lengthOfLongestSubstring("abcea") == 4
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Median of Two Sorted Arrays

## 原题

给两个整型的有序数组，要求找出这两个数组中的中位数，时间复杂度为 $O(\log(m+n))$ 。

注意点：

- 数组是有序的
- 时间复杂度为 $O(\log(m+n))$
- 考虑奇偶情况，如果总是为偶数，因返回中间的两个的平均数，且为float类型
- Python版本问题，部分Python3语法LeetCode好像不支持。

例子：

输入: num1=[1, 2], num2=[1, 2, 3] 输出: 2

输入: num1=[], num2=[2, 3] 输出: 2.5

## 解题思路

整体思路类似于在一个无序数组内找最小的k个数。我们通过两个数组各自的中位数将两个数组A、B分为四个部分，分别为A1、A2、B1、B2。现在我们来找出他们中第k小的数。如果A的中位数比B的中位数大，那么B1中的数比A2和B2中的都小，且小于部分A1中的数。此时，如果 $k > \text{len}(A1) + \text{len}(B1)$ ，那么第k个数就不可能在B1，因为比B1的数小的数最多只有B1加上部分的A1，也就是 $k < \text{len}(A1) + \text{len}(B1)$ ，矛盾；同时，我们还知道了A2中的数比A1和B1的大，且比部分B2的大，此时，如果 $k \leq \text{len}(A1) + \text{len}(B1)$ ，那么第k个数就不可能在A2中，因为A2中的数至少比A1加B1中的数大，也就是 $k > \text{len}(A1) + \text{len}(B1)$ ，矛盾。同理可以推理出另外两种情况。

## AC源码



```

class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """
        length1 = len(nums1)
        length2 = len(nums2)
        k = (length1 + length2) // 2
        if (length1 + length2) % 2 == 0:
            return (self.findK(nums1, nums2, k) + self.findK(num
s1, nums2, k - 1)) / 2.0;    # 2 is enough in python3
        else:
            return self.findK(nums1, nums2, k)

    def findK(self, num1, num2, k):
        # Recursive ends here
        if not num1:
            return num2[k]
        if not num2:
            return num1[k]
        if k == 0:
            return min(num1[0], num2[0])

        length1 = len(num1)
        length2 = len(num2)
        if num1[length1 // 2] > num2[length2 // 2]:
            if k > length1 // 2 + length2 // 2:
                return self.findK(num1, num2[length2 // 2 + 1:],
k - length2 // 2 - 1)
            else:
                return self.findK(num1[:length1 // 2], num2, k)
        else:
            if k > length1 // 2 + length2 // 2:
                return self.findK(num1[length1 // 2 + 1:], num2,
k - length1 // 2 - 1)
            else:
                return self.findK(num1, num2[:length2 // 2], k)

```

```
if __name__ == "__main__":  
    assert Solution().findMedianSortedArrays([1, 2], [1, 2, 3])  
    == 2  
    assert Solution().findMedianSortedArrays([], [2, 3]) == 2.5
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Longest Palindromic Substring

---

## 原题

找到一个字符串的最长回文子字符串，该字符串长度不超过1000，且只有唯一一个最长回文子串。

注意点：

- 返回结果是整个子字符串，不是它的长度
- 回文字符串要考虑奇偶的情况

例子：

输入: s="abae"

输出: aba

输入: s="abbae"

输出: abba

## 解题思路

依次把每一个字符当做回文字符串的中间字符，找到以该字符为中间字符的回文串的最大长度。分别对奇偶的情况进行讨论，接下来的关键就是对边界的把握，确保下标不要越界。当子串已经包含首字符或最后一个字符且此时还是回文串的时候，下标分别会向两边多移一位，需要补回来。

## 2017/05/04 更新

感谢/@Insosan 的提醒，原解法的时间复杂度为  $O(n^2)$ ，现添加manacher算法，由于算法比较复杂，暂不展开，可以参考以下的资料：

[https://www.wikiwand.com/en/Longest\\_palindromic\\_substring](https://www.wikiwand.com/en/Longest_palindromic_substring)

<http://www.jianshu.com/p/799bc53d4e3d>

## AC源码

```
class Solution(object):
    def longestPalindrome2(self, s):
        """
        :type s: str
        :rtype: str
        """
        if not s:
            return
        n = len(s)
        if n == 1:
            return s
        # Left index of the target substring
        l = 0
        # Right index of the target substring
        r = 0
        # Length of the longest palindromic substring for now
        m = 0
        # Length of the current substring
        c = 0
        # Whether the substring contains the first character or
        last character
        # and is palindromic
        b = True
        for i in range(0, n):
            # Odd situation
            for j in range(0, min(n - i, i + 1)):
                if (s[i - j] != s[i + j]):
                    b = False
                    break
            else:
                c = 2 * j + 1
        if (c > m):
            l = i - j + 1 - b
            r = i + j + b
            m = c
```

```

        b = True
        # Even situation
        for j in range(0, min(n - i - 1, i + 1)):
            if (s[i - j] != s[i + j + 1]):
                b = False
                break
            else:
                c = 2 * j + 2
        if (c > m):
            l = i - j + 1 - b
            r = i + j + 1 + b
            m = c
        b = True
        return s[l:r]

def longestPalindrome(self, s):
    string = "#" + "#".join(s) + "#"
    i = 0
    maxBorder = 0 # store the max border that has been reached
    maxCenter = 0 # the center of palindrome that has been largest for now
    p = [0 for _ in range(len(string))] # min in (center to i or i to border)
    res = [0, 0]

    while i < len(string):
        p[i] = min(p[2 * maxCenter - i], maxBorder - i) if maxBorder > i else 1
        while i - p[i] >= 0 and i + p[i] < len(string) and string[i - p[i]] == string[i + p[i]]:
            p[i] += 1
        if maxBorder < p[i] + i:
            maxBorder = p[i] + i
            maxCenter = i
            if maxBorder - maxCenter > res[1] - res[0]:
                res = [maxCenter, maxBorder]
        i += 1

    return "".join([x for x in string[2 * res[0] - res[1] + 1

```

```
:res[1]] if x != '#'])  
# Test cases  
if __name__ == "__main__":  
    assert Solution().longestPalindrome("aba") == "aba"  
    assert Solution().longestPalindrome("abba") == "abba"  
    assert Solution().longestPalindrome("xaba") == "aba"  
    assert Solution().longestPalindrome("xabba") == "abba"
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之ZigZag Conversion

## 原题

将所给字符串按特定形状排列，依次将每排的字符连接形成新的字符串。具体看一个例子：给定字符串"PAYPALISHIRING"，相应排列成的形状如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

可以看出是将字符串按照N的书写顺序重新排列了。现在要求我们从第一排开始依次把所有字符组成新的字符串："PAHNAPLSIIGYIR"

注意点：

- 行数不是固定的

例子：

输入: s="PAYPALISHIRING", numRows=3 输出: "PAHNAPLSIIGYIR"

## 解题思路

发现其实新的字符串的顺序在原字符串中是有规律可循的。重新组合后的形状其实就是一个个N，相邻N对应位置之间的字符个数是固定的，如例子中的P-A-H-N之间的字符都是3个。现在假设行数为n，则对应位置字符间的间距都为 $2n-3$ ，现在第一排和最后一排的字符顺序就可以确定了；但中间排的字符就比较特殊，在对应位置之间还会夹着一个字符。以例子中第一列的A和第二列的P来说，假设A的下标为i（从0开始），则A和P的间距为 $2(n-i-1)-1=2n-2i-3$ ，而A和L的间距为 $2n-3$ ，同时去掉P自己，所以P和L的间距为 $(2n-3)-(2n-2i-3)-1=2i-1$ 。这样只需遍历第一列的字符，并依次加上对应的间距来获取后面的字符即可。上文的间距是指中间隔了几个字符。

## AC源码

```
class Solution(object):
    def convert(self, s, numRows):
        """
        :type s: str
        :type numRows: int
        :rtype: str
        """
        if numRows<=1:
            return s
        result = ''
        index = 0
        n = len(s)
        for i in range(0, numRows):
            if i == 0 or i == numRows - 1:
                while index < n:
                    result += s[index]
                    index += 2 * numRows - 2
                index = i + 1
            else:
                while index < n:
                    result += s[index]
                    index += 2 * numRows - 2 * i - 2
                    if index >= n:
                        break
                    result += s[index]
                    index += 2 * i
                index = i + 1
        return result

# Test cases
if __name__ == "__main__":
    Solution().convert("PAYPALISHIRING", 2) == "PYAIHRNAPLSIIG"
    Solution().convert("PAYPALISHIRING", 3) == "PAHNAPLSIIGYIR"
    Solution().convert("PAYPALISHIRING", 4) == "PINALSIGYAHRPI"
```



欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之String to Integer

## 原题

讲一个字符串转化成int类型。题目非常简单，但要额外考虑的因素非常多，如下面的一些字符串的处理："+123", "-23 ", "231ji2", null, ""等等。

注意点：

- 字符串可能为空
- 字符串可能全是空格，或前后有空格
- 要考虑正负号
- 要考虑对于32位整数是否溢出
- 要考虑不是数字的字符，如果首字母不是数字，返回为0，在字符串中的将它及它之后的字符全部忽略
- 异常情况全部返回0

例子：

输入: str="+123" 输出: 123

输入: str="-123fe2" 输出: -123

## 解题思路

这道题把所有情况考虑清楚后就非常简单了，尤其是用Python来处理溢出非常方便。

## AC源码

```
class Solution(object):  
    def myAtoi(self, str):  
        """  
        :type str: str
```

```

:rtype: int
"""
INT_MAX = 2147483647
INT_MIN = -2147483648
# Check None situation
if not str:
    return 0
# Check only whitespace situation
str = str.strip()
if not str:
    return 0
flag = 1
if str[0] in ['+', '-']:
    if str[0] == '-':
        flag = -1
    str = str[1:]
# Check "+", "-" situation and the first char is not number
if not str or not str[0].isdigit():
    return 0
# Ignore all char after the first no-number char
for i, v in enumerate(str):
    if not v.isdigit():
        str = str[:i]
        break
result = 0
for v in str[:]:
    result += ord(v) - ord('0')
    result *= 10
result /= 10
result *= flag
if result > INT_MAX:
    return INT_MAX
if result < INT_MIN:
    return INT_MIN
return result

if __name__ == "__main__":
    assert Solution().myAtoi(" -1123") == -1123

```

```
assert Solution().myAtoi("22222222222222") == 2147483647
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Reverse Integer

---

## 原题

翻转一个integer类型的数字。

注意点：

- 注意末尾有0的情况不能直接当作字符串翻转
- 有正负两种情况
- integer是32位整型，考虑溢出

例子：

输入: x=123 输出: 321

输入: x=-123 输出: -321

## 解题思路

由于Python可以支持几乎无限大的数，直接把末尾的数不断添加到目标书中即可。最后再处理溢出的情况。如果想通过转换为字符然后翻转的方法来实现，需要把末尾的0先去掉。

## AC源码

```
class Solution(object):
    def reverse(self, x):
        """
        :type x: int
        :rtype: int
        """
        # Consider positive and negative situation
        flag = 0
        if x < 0:
            flag = -1
        else:
            flag = 1
        x *= flag
        result = 0
        while x:
            result = result * 10 + x % 10
            x /= 10
        if result > 2147483647:
            return 0
        else:
            return result * flag

if __name__ == "__main__":
    assert Solution().reverse(321000) == 123
    assert Solution().reverse(-321) == -123
    assert Solution().reverse(1534236469) == 0
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Palindrome Number

---

## 原题

判断一个int型数字是否是回文形式，不许用额外的空间。

注意点：

- 负数都是不回文数
- 不许用额外的空间

例子：

输入: x=123 输出: False

输入: x=12321 输出: True

## 解题思路

既然是判断是否是回文数，那就依次获取数的首尾两个数判断是否相等。先通过for循环来获取数字的最高位，这样就可以从头开始获取每个数字，通过%操作从末尾获取每个数字。为了方便操作，可以把比较过的数字去除掉。需要注意的是，去掉首尾后，原来的最高位要除以100，而不是10。

## AC源码

```
class Solution(object):
    def isPalindrome(self, x):
        """
        :type x: int
        :rtype: bool
        """
        if x < 0:
            return False
        div = 1
        while x / div >= 10:
            div *= 10
        while x > 0:
            l = x // div
            r = x % 10

            if l != r:
                return False
            x %= div
            x //= 10
            div /= 100
        return True

if __name__ == "__main__":
    assert Solution().isPalindrome(123) == False
    assert Solution().isPalindrome(12321) == True
    assert Solution().isPalindrome(-121) == False
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



# LeetCode 解题之 Regular Expression Matching

## 原题

简易版正则表达式匹配，只有两种通配符，"."表示任意一个字符，"c\*"表示字符c可以有零个或多个。

注意点：

- 存在一些不合理的字符组合，如"\*\*\*"
- "."可以表示任意字符串
- 需要匹配整个目标串，而不是部分

例子：

输入: s="aab", p="c\*a\*b" 输出: True

## 解题思路

开始用递归实现了一遍，结果超时，改用动态规划。dp[i][j]表示s[i:]和p[j:]的匹配情况，围绕"\*"进行分类讨论，分类的情况比较多，具体请参看代码注释。

## AC源码

```
class Solution(object):
    def isMatch(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: bool
        """
        m = len(s)
        n = len(p)
```

```

# Init dp
dp = [[False for i in range(n + 1)] for i in range(m + 1)
)]

# When string and pattern are all None
dp[m][n] = True

# When the string is None, pattern like "a*" can still match it
for i in range(n - 1, -1, -1):
    if p[i] == "*":
        dp[m][i] = dp[m][i + 1]
    elif i + 1 < n and p[i + 1] == "*":
        dp[m][i] = dp[m][i + 1]
    else:
        dp[m][i] = False

for i in range(m - 1, -1, -1):
    for j in range(n - 1, -1, -1):
        # When the current character is "*"
        if p[j] == "*":
            if j - 1 >= 0 and p[j - 1] != "*":
                dp[i][j] = dp[i][j + 1]
            # If the pattern is starting with "*" or has
            "*" in it

            else:
                return False
        # When the the second character of pattern is "*"

        elif j + 1 < n and p[j + 1] == "*":
            # When the current character matches, there
            are three possible situation
            # 1. "." matches nothing
            # 2. "c*" matches more than one character
            # 3. "c*" just matches one character
            if s[i] == p[j] or p[j] == ".":
                dp[i][j] = dp[i][j + 2] or dp[i + 1][j]
            or dp[i + 1][j + 2]

            # Ignore the first two characters("c*") in pattern since they cannot match
            # the current character in string
            else:

```

```
        dp[i][j] = dp[i][j + 2]
    else:
        # When the current character is matched
        if s[i] == p[j] or p[j] == ".":
            dp[i][j] = dp[i + 1][j + 1]
        else:
            dp[i][j] = False
    return dp[0][0]

if __name__ == "__main__":
    assert Solution().isMatch("aa", "a") == False
    assert Solution().isMatch("aa", "aa") == True
    assert Solution().isMatch("aaa", "aa") == False
    assert Solution().isMatch("aa", "a*") == True
    assert Solution().isMatch("aa", ".*") == True
    assert Solution().isMatch("ab", ".*") == True
    assert Solution().isMatch("aab", "c*a*b") == True
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Container With Most Water

---

## 原题

给定一组长短不一的隔板，挑其中的两块板，使得板子之间能装最多的水。

注意点：

- 两块板之间能装多少水是由短的那块板决定的
- 选定两块板之后，它们之间的板就不存在了

例子：

输入: height=[1,1,1] 输出: 2

## 解题思路

我们把两块板分为左板、右板，现在考虑如下情况。假设左板高度为 $h$ ，且比右板低，两块板之间的距离为 $w$ ，则此时最多能装水 $w*h$ ，此时我们尝试移动隔板。如果将左板向右移，那么有可能使容积变大，例如，左板右边的板子高 $h_1$ （还是比右板低），此时最多装水 $(w-1)*h_1$ ，有可能比 $w*h$ 大；如果将右板向左移，由于水的高度不能高于左板，所以容积最多为 $(w-1)*h$ ，肯定比 $w*h$ 小。基于上面的假设，我们只要把两块隔板依次向中间靠拢，就可以求出最大的容积。

## AC源码

```
class Solution(object):
    def maxArea(self, height):
        """
        :type height: List[int]
        :rtype: int
        """
        if not height:
            return 0
        left = 0
        right = len(height) - 1
        result = 0
        while left < right:
            if height[left] < height[right]:
                area = height[left] * (right - left)
                result = max(result, area)
                left += 1
            else:
                area = height[right] * (right - left)
                result = max(result, area)
                right -= 1
        return result

if __name__ == "__main__":
    assert Solution().maxArea([1, 1]) == 1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Integer to Roman

## 原题

将一个int型的数字转化为罗马数字，范围在1-3999。下面是罗马数字的介绍及基本规则：

罗马数字采用七个罗马字母作数字、即I（1）、X（10）、C（100）、M（1000）、V（5）、L（50）、D（500）。记数的方法：

- 相同的数字连写，所表示的数等于这些数字相加得到的数，如 III=3
- 小的数字在大的数字的右边，所表示的数等于这些数字相加得到的数，如 VIII=8、XII=12
- 小的数字（限于I、X和C）在大的数字的左边，所表示的数等于大数减小数得到的数，如 IV=4、IX=9

注意点：

- 基本数字I、X、C中的任何一个、自身连用构成数目、或者放在大数的右边连用构成数目、都不能超过三个，放在大数的左边只能用一个
- 不能把基本数字V、L、D中的任何一个作为小数放在大数的左边采用相减的方法构成数目，放在大数的右边采用相加的方式构成数目时只能使用一个
- V和X左边的小数字只能用I
- L和C左边的小数字只能用X
- D和M左边的小数字只能用C

例子：

输入: num=99 输出: XCIX

## 解题思路

根据上面的规则和注意点可以看出，罗马字符及其组合能代表的数字有：

阿拉伯数字	罗马数字
1000	M
900	CM
500	D
400	CD
100	C
90	XC
50	L
40	XL
10	X
9	IX
5	V
4	IV
1	I

只需要依次找出数字中包含的最大的可转化为罗马数字的数字即可。例如99，比小的数字是90，减去后剩下9，可以直接转换，所以结果是XCIX。

AC源码

```
class Solution(object):
    def intToRoman(self, num):
        """
        :type num: int
        :rtype: str
        """
        nums = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
        strings = ["M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"]
        result = ''
        for i in range(len(nums)):
            while num >= nums[i]:
                num -= nums[i]
                result += strings[i]
        return result

# Test cases
if __name__ == "__main__":
    assert Solution().intToRoman(12) == "XII"
    assert Solution().intToRoman(21) == "XXI"
    assert Solution().intToRoman(99) == "XCIX"
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Roman to Integer

## 原题

将一个罗马数字转化为阿拉伯数字，范围在1-3999。下面是罗马数字的介绍及基本规则：

罗马数字采用七个罗马字母作数字、即I（1）、X（10）、C（100）、M（1000）、V（5）、L（50）、D（500）。记数的方法：

- 相同的数字连写，所表示的数等于这些数字相加得到的数，如 III=3
- 小的数字在大的数字的右边，所表示的数等于这些数字相加得到的数，如 VIII=8、XII=12
- 小的数字（限于I、X和C）在大的数字的左边，所表示的数等于大数减小数得到的数，如 IV=4、IX=9

注意点：

- 输入的罗马数字是符合规范的，不需要考虑错误情况

例子：

输入: s="XCIX" 输出: 99

## 解题思路

根据罗马数字的规则，只有在前面的字母比当前字母小的情况下要执行减法，其他情况只需要把罗马字母对应的数字直接相加即可。如果发现前一个字母比当前字母小，就减去前一个字母，因为错误的把它加入了结果，且在加上当前字母时还要减去前一个字母的值。

## AC源码

```
class Solution(object):
    def romanToInt(self, s):
        """
        :type s: str
        :rtype: int
        """
        map = {"M": 1000, "D": 500, "C": 100, "L": 50, "X": 10,
               "V": 5, "I": 1}
        result = 0
        for i in range(len(s)):
            if i > 0 and map[s[i]] > map[s[i - 1]]:
                result -= map[s[i - 1]]
                result += map[s[i]] - map[s[i - 1]]
            else:
                result += map[s[i]]
        return result

# Test cases
if __name__ == "__main__":
    assert Solution().romanToInt("XII") == 12
    assert Solution().romanToInt("XXI") == 21
    assert Solution().romanToInt("XCIX") == 99
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Longest Common Prefix

---

## 原题

找出一组字符串中最长的公共前缀。

注意点：

- 字符串长度不一样
- 考虑字符列表为空的情况

例子：

输入: `str=["hello", "heabc", "hewww"]` 输出: `"he"`

## 解题思路

如果列表长度大于一，不妨把第一字符串当做基准，用一个指针来表示在此之前的字符是满足题目要求的。遍历每一个字符串，用指针对应的字符与基准中相应的字符比较，如不同则前面的子字符串就是所要求的结果；如果全都相同，则指针右移。还有一种情况要考虑，后面的字符串可能没有第一个字符串长，如果指针超过了最短的字符串也应该终止。

## AC源码

```
class Solution(object):
    def longestCommonPrefix(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        if not strs:
            return ""
        longest = strs[0]
        for i in range(len(strs[0])):
            for str in strs:
                if len(str) <= i or strs[0][i] != str[i]:
                    return strs[0][:i]
        return strs[0]

if __name__ == "__main__":
    assert Solution().longestCommonPrefix(["hello", "heabc", "hell"]) == "he"
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

## LeetCode 解题之 3Sum

---

### 原题

找出一个列表中所有和为零的三元组。要求求出的三元组中没有重复。

注意点：

- 三元组中的数字要按增序排列( $a \leq b \leq c$ )
- 结果集中没有重复的三元组

例子：

输入: `nums=[-1, 0, 1, 2, -1, -4]` 输出: `[[-1, -1, 2], [-1, 0, 1]]`

### 解题思路

求一个列表中所有和为零的二元组的一种思路是先把列表排序，再用两个指针从两头向中间移动。如果前后两个数的和小于0，则左指针右移；如果和大于0，则右指针左移。求三元组时可以参考这种做法，第一个数`a`确定后，可以理解为求列表中和为`-a`的二元组。由于不要考虑重复的元组，遇到重复的数可以直接跳过。

### AC源码

```
class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        # Sorted array can save a lot of time
        nums.sort()
        result = []
        i = 0
        while i < len(nums) - 2:
            j = i + 1
            k = len(nums) - 1
            while j < k:
                l = [nums[i], nums[j], nums[k]]
                if sum(l) == 0:
                    result.append(l)
                    j += 1
                    k -= 1
                    # Ignore repeat numbers
                    while j < k and nums[j] == nums[j - 1]:
                        j += 1
                    while j < k and nums[k] == nums[k + 1]:
                        k -= 1
                elif sum(l) > 0:
                    k -= 1
                else:
                    j += 1
            i += 1
            # Ignore repeat numbers
            while i < len(nums) - 2 and nums[i] == nums[i - 1]:
                i += 1
        return result

if __name__ == "__main__":
    assert Solution().threeSum([-1, 0, 1, 2, -1, -4]) == [[-1, -1, 2], [-1, 0, 1]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 3Sum Closest

---

### 原题

找出一个列表中三个元素之和与目标值最接近的情况，并返回这个值。假设整个列表中只有一个最接近的值。

注意点：

- 结果要求返回的是和，而不是三元组

例子：

输入: nums=[1, 1, 1, 1], target=-100 输出: 3

### 解题思路

思路与 3Sum 基本相同，现在要额外维护一个表示之前三元组中与目标值的差最小值的变量，这个变量的初始化值应该很大，防止把有意义的三元组直接排除了。此外，由于题目中明确说只有唯一的一组最优解，所有不用考虑重复数字了。

### AC 源码



```
class Solution(object):
    def threeSumClosest(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        nums.sort()
        i = 0
        result = 0
        # Init the distance between result and target with a very large number
        distance = pow(2, 32) - 1
        for i in range(len(nums)):
            j = i + 1
            k = len(nums) - 1
            while j < k:
                l = [nums[i], nums[j], nums[k]]
                if sum(l) == target:
                    return target
                if abs(sum(l) - target) < distance:
                    result = sum(l)
                    distance = abs(sum(l) - target)
                elif sum(l) > target:
                    k -= 1
                else:
                    j += 1
            return result

if __name__ == "__main__":
    assert Solution().threeSumClosest([1, 1, 1, 1], -100) == 3
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Letter Combinations of a Phone Number

## 原题

手机按键上每个数字都对应了多个字母，如2对应了"abc"，现给出一个数字串，要求把其中的每个数字都转化为对应的字母中的一个，列出所有的组合情况。

注意点：

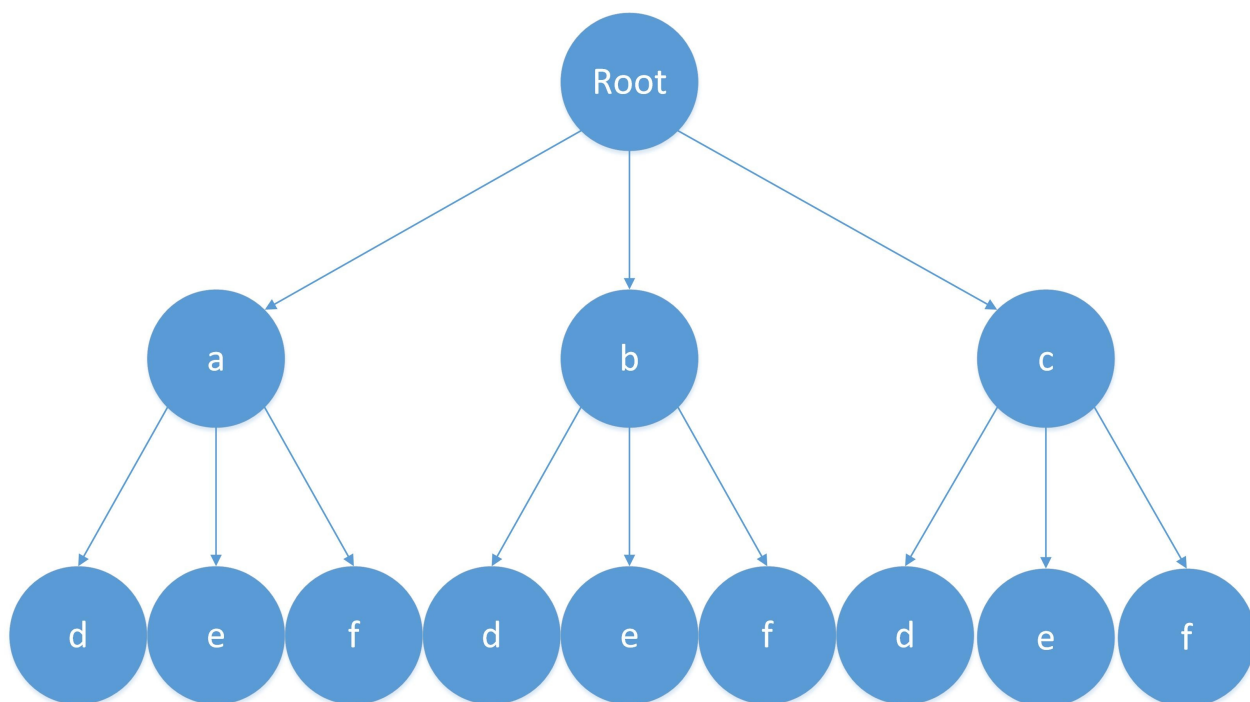
- 对结果的排列顺序没有要求

例子：

输入: digits="23" 输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

## 解题思路

把每个数字对应的字母都当做树的节点，如下图，则所求结果就是从根节点到叶节点的所有路径，采用深度优先遍历算法。



## AC源码

```
class Solution(object):
    digit2letters = {
        '2': "abc",
        '3': "def",
        '4': "ghi",
        '5': "jkl",
        '6': "mno",
        '7': "pqrs",
        '8': "tuv",
        '9': "wxyz",
    }

    def letterCombinations(self, digits):
        """
        :type digits: str
        :rtype: List[str]
        """
        if not digits:
            return []
        result = []
        self.dfs(digits, "", result)
        return result

    def dfs(self, digits, current, result):
        if not digits:
            result.append(current)
            return
        for c in self.digit2letters[digits[0]]:
            self.dfs(digits[1:], current + c, result)

if __name__ == "__main__":
    assert Solution().letterCombinations("23") == ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之4Sum

---

### 原题

找出一个列表中四个元素之和为目标值的情况，打印出所有的情况。

注意点：

- 四元组中的数字要按增序排列( $a \leq b \leq c$ )
- 结果集中没有重复的四元组

例子：

输入: `nums=[1, 0, -1, 0, -2, 2]` 输出: `[[-1, 0, 0, 1], [-2, 0, 0, 2], [-2, -1, 1, 2]]`

### 解题思路

想参照之前的一题3Sum的解法来解决，结果超时了。换个思路，空间换时间。既然有四个数，那就把前两个数之和和后两个数之和分别当做一个数。现在要求的就是在一个列表中哪两个数的和为特定值。可以先遍历一遍列表，存值和它的下标的键值对，第二遍遍历列表寻找（目标值-当前值）是否在之前的键值对中，如果在就是符合的一组解。

### AC源码

```

class Solution(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        if len(nums) < 4:
            return []
        result = set()
        sumsIndexes = {}
        # Get all two elements' sum and indexes map
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if nums[i] + nums[j] in sumsIndexes:
                    sumsIndexes[nums[i] + nums[j]].append((i, j))
            else:
                sumsIndexes[nums[i] + nums[j]] = [(i, j)]

        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                sumNeeded = target - (nums[i] + nums[j])
                if sumNeeded in sumsIndexes:
                    for index in sumsIndexes[sumNeeded]:
                        # Ignore repeating results
                        if index[0] > j:
                            result.add(tuple(sorted([nums[i], nums[j],
                                nums[index[0]], nums[index[1]]])))
        # Format result with list[list] pattern
        result = [list(l) for l in result]
        return result

if __name__ == "__main__":
    assert Solution().fourSum([1, 0, -1, 0, -2, 2], 0) == [[-1, 0, 0, 1], [-2, 0, 0, 2], [-2, -1, 1, 2]]

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Remove Nth Node From End of List

## 原题

将一个链表中的倒数第n个元素从链表中去除。

注意点：

- 不用考虑n是非法的情况
- 尽量做到只遍历一次链表

例子：

输入: list = 1->2->3->4->5, n = 2. 输出: 1->2->3->5

## 解题思路

基本思路就是用两个指针一前一后遍历链表，在第一指针遍历了n节点后，第二个指针开始和它同步前进。需要注意的是如果去除的正好是头节点，那情况就有些特殊，需要分类讨论。可以添加一个假的头节点，使原来的头节点也变为普通的节点，这样就不用分类了。关于节点去除，就是通过前一个节点的指针指向当前节点的后一个节点。

## AC源码

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

# Define this to check if it works well
def myPrint(self):
```



```
        print(self.val)
        if self.next:
            self.next.myPrint()

class Solution(object):
    def removeNthFromEnd(self, head, n):
        """
        :type head: ListNode
        :type n: int
        :rtype: ListNode
        """
        if not head:
            return head
        dummy = ListNode(-1)
        dummy.next=head
        prev = dummy
        cur = dummy
        while prev and n >= 0:
            prev = prev.next
            n -= 1
        while prev:
            prev = prev.next
            cur = cur.next
        cur.next = cur.next.next
        return dummy.next

if __name__ == "__main__":
    n5 = ListNode(5)
    n4 = ListNode(4)
    n3 = ListNode(3)
    n2 = ListNode(2)
    n1 = ListNode(1)
    n1.next = n2
    n2.next = n3
    n3.next = n4
    n4.next = n5
    result = Solution().removeNthFromEnd(n1, 5)
    result.myPrint()
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Valid Parentheses

---

## 原题

判断一个只包含各种括号符号的字符串中括号的匹配情况。

注意点：

- 字符串中只会包含 "(", ")", "[", "]", "{", "}" 这些字符
- 括号匹配要注意顺序，字符串 "([)]" 是错误的匹配

例子：

输入: `s="(){}"` 输出: `True`

输入: `s="(){}["` 输出: `False`

## 解题思路

典型的用栈来解决的问题，遇到左括号就压栈，遇到右括号时如果栈为空（类似 "]]]" 的情况），则失败，否则取栈顶元素，看两个括号是否匹配。如果最后栈不为空（类似 "[[[[" 的情况），则匹配失败。

## AC 源码

```
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        # Valid str must be even
        if len(s) % 2 == 1:
            return False
        stack = []
        left = ("(", "[", "{")
        right = (")", "]", "}")
        zip(left, right)
        for v in s:
            if v in left:
                stack.append(v)
            else:
                if not stack:
                    return False
                p = stack.pop()
                if left.index(p) != right.index(v):
                    return False
        return len(stack) == 0

if __name__ == "__main__":
    assert Solution().isValid("({}){}") == True
    assert Solution().isValid("({})") == False
    assert Solution().isValid("}}}")) == False
    assert Solution().isValid("(((") == False
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Merge Two Sorted Lists

---

## 原题

将两个有序的链表拼接成一个有序的链表。

注意点：

- 不需要额外申请节点，主要把原链表中的节点串联起来
- 原链表中的一个已经全部在新链表中后，另一个链表剩余的部分可以直接拼接

例子：

输入: l1 = 1->2->4, l2 = 3 输出: 1->2->3->4

## 解题思路

为了避免分类讨论，添加一个假的头节点。现在只需要两个指针分别指向原来的两个链表，将其中比较小的节点添加到新的链表中。传入的参数l1和l2正好可以当作遍历两个链表的指针。

## AC源码

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

class Solution(object):
    def mergeTwoLists(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        temp = ListNode(-1)
        head = temp
        while l1 and l2:
            if l1.val > l2.val:
                temp.next = l2
                l2 = l2.next
            else:
                temp.next = l1
                l1 = l1.next
            temp = temp.next
        if l1:
            temp.next = l1
        else:
            temp.next = l2
        return head.next

if __name__ == "__main__":
    assert Solution().mergeTwoLists(ListNode(1), ListNode(2)).val == 1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Generate Parentheses

---

### 原题

罗列出n组括号的所有合法的排列组合。

注意点：

- 只有一种括号形式"()"

例子：

输入: n = 3 输出: ['((()))', '(()())', '(())()', '()(())', '()()()']

### 解题思路

我们先来讨论一下什么样的排列是不合法的，由于只有一种类型的括号，所以我们只要时刻保持字符串中的左括号不少于右括号即可。例如：字符串")\*"上来就右括号数目比左括号多，它就是不合法的，没有左括号来与第一个右括号组合。在进行递归的时候注意优先添加左括号，在现有右括号少于左括号的情况下，可以添加右括号。递归的结束条件是所有的括号都已加入字符串中。

### AC源码



```
class Solution(object):
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        result = []
        self.generate(n, n, "", result)
        return result

    def generate(self, left, right, str, result):
        if left == 0 and right == 0:
            result.append(str)
            return
        if left > 0:
            self.generate(left - 1, right, str + "(", result)
        if right > left:
            self.generate(left, right - 1, str + ")", result)

if __name__ == "__main__":
    assert (Solution().generateParenthesis(3)) == ['((()))', '(()())', '()()()', '()(())', '()()()()']
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Merge k Sorted Lists

---

### 原题

将k个有序的链表拼接成一个有序的链表。

注意点：

- Python中有自带的堆排序实现heapq

例子：

输入: lists = [[1->2->10],[3->9],[5->6]] 输出: 1->2->3->5->6->9->10

### 解题思路

整体思路与[Merge Two Lists](#)相同。不过就是从原来的两个数中取最小的节点改为从k个数中取最小的节点。这是一个典型的堆排序的应用，Python中堆排序可以用heapq实现。

### AC源码

```
import heapq

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

class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        heap = []
        for node in lists:
            if node:
                heapq.heappush(heap, (node.val, node))

        temp = ListNode(-1)
        head = temp
        while heap:
            smallestNode = heapq.heappop(heap)[1]
            temp.next = smallestNode
            temp = temp.next
            if smallestNode.next:
                heapq.heappush(heap, (smallestNode.next.val, smallestNode.next))
        return head.next
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Swap Nodes in Pairs

---

## 原题

将链表中相邻的两个节点交换位置，注意第一个节点与第二个节点要交换位置，而第二个节点不用与第三个节点交换位置。

注意点：

- 不允许修改节点的值
- 只能用常量的额外空间

例子：

输入: head = 1->2->3->4 输出: 2->1->4->3

## 解题思路

比较常见的链表操作。下面看一下典型情况，如要交换链表中A->B->C->D中的B和C需要做如下操作：

1. 将A指向C
2. 将B指向D
3. 将C指向B

在头节点之前加一个假节点就可以使所有的交换都符合上面的情况。

## AC源码

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

class Solution(object):
    def swapPairs(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        prev = ListNode(-1)
        prev.next = head
        temp = prev
        while temp.next and temp.next.next:
            node1 = temp.next
            node2 = temp.next.next
            temp.next = node2
            node1.next = node2.next
            node2.next = node1
            temp = temp.next.next
        return prev.next
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Reverse Nodes in k-Group

## 原题

将一个链表中每k个数进行翻转，末尾不足k个的数不做变化。

注意点：

- 不允许修改节点的值
- 只能用常量的额外空间

例子：

输入: head = 1->2->3->4->5, k = 2 输出: 2->1->4->3->5

输入: head = 1->2->3->4->5, k = 3 输出: 3->2->1->4->5

## 解题思路

这个题是[Swap Nodes in Pairs](#)的升级版。我们来看一下翻转k个节点要进行哪些操作，A->B->C->D->E，现在我们要翻转BCD三个节点。进行以下几步：

1. C->B
2. D->C
3. B->E
4. A->D
5. 返回及节点B

上面做了两件事，把k个节点先翻转（1、2两步），再和前后两个节点连接起来（3、4两步）。

## AC源码

```
# Definition for singly-linked list.  
class ListNode(object):
```

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

class Solution(object):
    def reverseKGroup(self, head, k):
        """
        :type head: ListNode
        :type k: int
        :rtype: ListNode
        """
        if not head or k <= 1:
            return head
        dummy = ListNode(-1)
        dummy.next = head
        temp = dummy
        while temp:
            temp = self.reverseNextK(temp, k)
        return dummy.next

    def reverseNextK(self, head, k):
        # Check if there are k nodes left
        temp = head
        for i in range(k):
            if not temp.next:
                return None
            temp = temp.next

        # The last node when the k nodes reversed
        node = head.next
        prev = head
        curr = head.next
        # Reverse k nodes
        for i in range(k):
            nextNode = curr.next
            curr.next = prev
            prev = curr
            curr = nextNode
        # Connect with head and tail
```

```
node.next = curr  
head.next = prev  
return node
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode解题之Remove Duplicates from Sorted Array

---

## 原题

从一个有序的数组中去除重复的数字，返回处理后的数组长度。

注意点：

- 只能用常量的额外空间
- 将不重复的数字移到数组前部，剩余的部分不需要处理

例子：

输入: `nums = [1, 1, 2]` 输出: 2

## 解题思路

用一个下标`index`来标记下一个不重复的数字存放的位置，另一个下标`start`来表示当前是和哪个数字来比较有没有重复。遍历数字，如果不重复则放到`index`位置，后移`index`，并更新`start`位置；否则继续遍历。返回`index`即为不重复数组的长度。

## AC源码

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0
        # The index where the character needs to be placed
        index = 1
        # The index of repeating characters
        start = 0
        for i in range(1, len(nums)):
            if nums[start] != nums[i]:
                nums[index] = nums[i]
                index += 1
                start = i
        return index

if __name__ == "__main__":
    assert Solution().removeDuplicates([1, 1, 2]) == 2
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Remove Element

---

## 原题

删除一个数组中某一特定数值的元素，返回删除后的数组长度。

注意点：

- 操作结束后的数字排列顺序不需要与之前相同
- 超出返回长度的部分不需要处理

例子：

输入: nums [1, 2, 3, 4, 3, 2, 1]，val = 1 输出: 5

## 解题思路

左右两个指针向中间靠拢，左指针找到一个等于val的值，右指针找到第一个不等于val的值，把右指针指向的值赋值给左指针。继续向中间靠拢。

## AC源码

```
class Solution(object):
    def removeElement(self, nums, val):
        """
        :type nums: List[int]
        :type val: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        while left <= right:
            while left <= right and nums[left] != val:
                left += 1
            while left <= right and nums[right] == val:
                right -= 1
            if left < right:
                nums[left] = nums[right]
                left += 1
                right -= 1
        return right + 1

if __name__ == "__main__":
    assert Solution().removeElement([1, 2, 3, 4, 3, 2, 1], 1) == 5
    assert Solution().removeElement([2], 3) == 1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Implement strStr()

## 原题

实现字符串子串匹配函数 `strStr()`。如果字符串 `A` 是字符串 `B` 的子串，则返回 `A` 在 `B` 中首次出现的地址，否则返回 `-1`。

注意点：

- 空字符串是所有字符串的子串，返回 `0`

例子：

输入: `haystack = "abc"`, `needle = "bc"` 输出: `1`

输入: `haystack = "abc"`, `needle = "gd"` 输出: `-1`

## 解题思路

字符串匹配常见的算法是 KMP，不过感觉该算法理解困难，效率也不是特别高。我用了 Sunday 算法来实现字符串的匹配。大体思路如下：

被搜索的字符串是 `"abcdefg"`，要搜索的字符串是 `"ef"`

```
abcdefg
ef
```

如果当前不匹配，则判断当前尝试匹配的后一位，即 `"c"` 是否在要搜索的字符串中，如果不在，则要搜索的字符串直接后移它自己的长度 `+1`。

```
abcdefg
  ef
```

如果存在，如此时 `"f"` 在 `"ef"` 中，则把该位置对齐。

```
abcdefg  
ef
```

匹配成功返回结果。

## AC源码

```

class Solution(object):
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """
        if not needle:
            return 0
        if not haystack:
            return -1
        i = 0
        needleLength = len(needle)
        while i < len(haystack):
            a = haystack[i:i + needleLength]
            if haystack[i:i + needleLength] == needle:
                return i
            else:
                index = 0
                try:
                    index = needle.rindex(haystack[i + needleLength
gth]))
                except Exception:
                    i += needleLength + 1
                i += needleLength - index
        return -1

if __name__ == "__main__":
    assert Solution().strStr("abcdefg", "ab") == 0
    assert Solution().strStr("abcdefg", "bc") == 1
    assert Solution().strStr("abcdefg", "cd") == 2
    assert Solution().strStr("abcdefg", "fg") == 5
    assert Solution().strStr("abcdefg", "bcf") == -1

```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。





# LeetCode 解题之 Divide Two Integers

---

## 原题

实现两个int型数字的除法，不可以使用乘法、除法和模操作。

注意点：

- 如果结果溢出int类型，返回MAX\_INT
- 注意负数情况

例子：

输入: dividend = 5, divisor = -1 输出: -5

## 解题思路

可以先把符号抽取出来，只考虑两个正数相除的情况。除法其实就是被减数不断减去减数的操作，但直接不断进行减法会超时，应尽量减去大的数字，通过位移操作来快速找到比被减数小一些的减数的倍数current。不断减去且缩小current。溢出只可能是向上溢出，通过min操作进行过滤。

## AC源码

```
class Solution(object):
    def divide(self, dividend, divisor):
        """
        :type dividend: int
        :type divisor: int
        :rtype: int
        """
        MAX_INT = 2147483647
        sign = 1
        if dividend >= 0 and divisor < 0 or dividend <= 0 and di
visor > 0:
            sign = -1
        dividend = abs(dividend)
        divisor = abs(divisor)

        result = 0
        current = divisor
        currentResult = 1
        while current <= dividend:
            current <<= 1
            currentResult <<= 1
        while divisor <= dividend:
            current >>= 1
            currentResult >>= 1
            if current <= dividend:
                dividend -= current
                result += currentResult
        return min(sign * result, MAX_INT)

if __name__ == "__main__":
    assert Solution().divide(5, -1) == -5
    assert Solution().divide(10, 2) == 5
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode 解题之 Substring with Concatenation of All Words

---

### 原题

现有一组长度相等的字符串 `words`，要在原字符串中找出正好包含 `words` 中所有字符串的子字符串的起始位置。

注意点：

- 返回结果的顺序没有关系

例子：

输入: `s = "barfoothefoobarman"`, `words = ["foo", "bar"]`

输出: `[0, 9]`

### 解题思路

因为 `words` 中的单词可能有重复，所以要有一个 `dict` 来记录一下每个字符串的数目。然后在遍历原字符串的时候，只需要遍历单词的长度次即可，如 `"barfoothefoobarman"`，因为目标单词的长度为 3，所以只需遍历：

- `'bar' | 'foo' | 'the' | 'foo' | 'bar' | 'man'`
- `'arf' | 'oot' | 'hef' | 'oob' | 'arm'`
- `'rfo' | 'oth' | 'efo' | 'oba' | 'rma'`

在遍历时，需要两个指针，一个用来标记子字符串的开始，另一个用来标记子字符串的结束。再用一个 `dict` 来记录当前字符串中单词的数量，如果下一个单词不在 `words` 中，那么清空该 `dict`，把前指针直接跳到后指针处；如果在 `words` 中，那么相应的键值要加一，此时如果那个单词的数量超过了目标中的数目，那么前指针要不断后移直到吐出一个那个单词。通过前后指针之差是否等于所有目标单词长度之和来判断是否有目标子字符串。

## AC源码

```

class Solution(object):
    def findSubstring(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: List[int]
        """
        s_length = len(s)
        word_num = len(words)
        word_length = len(words[0])
        words_length = word_num * word_length
        result = []
        words_dict = {}
        for word in words:
            words_dict[word] = words_dict[word] + 1 if word in words_dict else 1
        for i in range(s_length - words_length + 1):
            left = i
            right = i
            curr_dict = {}
            while right + word_length <= s_length:
                word = s[right:right + word_length]
                right += word_length
                if word in words_dict:
                    curr_dict[word] = curr_dict[word] + 1 if word in curr_dict else 1
                    while curr_dict[word] > words_dict[word]:
                        curr_dict[s[left:left + word_length]] -= 1
                        left += word_length
                if right - left == words_length:
                    result.append(left)
            else:
                curr_dict.clear()
                left = right
        return result

```

```
if __name__ == "__main__":  
    assert Solution().findSubstring("barfoothefoobarman", ["foo"  
    , "bar"]) == [0, 9]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Next Permutation

---

## 原题

找出一个数组按字典序排列的后一种排列。

注意点：

- 如果原来就是字典序排列中最大的，将其重新排列为字典序最小的排列
- 不要申请额外的空间
- 小心数组越界问题
- 函数没有返回值，直接修改列表

例子：

输入: [1,2,3] 输出: [1,3,2]

输入: [3,2,1] 输出: [1,2,3]

## 解题思路

通过一个例子来说明，原数组为[1,7,3,4,1]。我们想要找到比173421大一点的数，那就要优先考虑将后面的数字变换顺序，而421从后往前是升序的（也就是这三个数字能组成的最大的数），变换了反而会变小，所以要先找到降序的点。可以看出3是第一个降序的点，要想整个数变大，3就要变大，从后往前找到第一个比3大的数4，将3和4交换位置得到174321，既然原来3所在位置的数字变大了，那整个数肯定变大了，而它之后的数是最大的（从后往前是升序的），应转换成最小的，直接翻转。

## AC源码

```
class Solution(object):
    def nextPermutation(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        length = len(nums)

        targetIndex = 0
        changeIndex = 0
        for i in range(length - 1, 0, -1):
            if nums[i] > nums[i - 1]:
                targetIndex = i - 1
                break
        for i in range(length - 1, -1, -1):
            if nums[i] > nums[targetIndex]:
                changeIndex = i
                break
        nums[targetIndex], nums[changeIndex] = nums[changeIndex], nums[targetIndex]
        if targetIndex == changeIndex == 0:
            nums.reverse()
        else:
            nums[targetIndex + 1:] = reversed(nums[targetIndex + 1:])
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Longest Valid Parentheses

## 原题

找出一个只包含"("和")"的字符串中最长的有效子字符串的长度。有效的意思是指该子字符串中的括号都能正确匹配。

注意点：

- 注意空字符串

例子：

输入:  $s = "()"$  输出: 2

输入:  $s = ")()()"$  输出: 4

## 解题思路

采用了动态规划， $dp[i]$ 表示以 $i$ 为子字符串末尾时的最大长度，最后的结果就是 $dp$ 中的最大值。如果不是空字符串，则 $dp[0]=0$ ，因为一个括号肯定无法正确匹配。递推关系是：

```
) ( ) ( ( ) ) )  
0 1 2 3 4 5 6 7
```

看当前括号的前一个括号的匹配情况，例如在7之前以6结尾的最佳匹配是3-6，看3之前的括号和7是否匹配，不匹配则没有变化；而6之前以5结尾的最佳匹配是4-5，此时3和6匹配，则 $dp[i]+2$ 。此外，如果与当前括号匹配的左括号之前的括号的 $dp$ 值也应该加进来，因为由于添加了当前的括号，那些括号也被连接起来了。例如3和6匹配后，1和2也应该被加到以6结尾的最佳匹配中。

## AC源码

```
class Solution(object):
    def longestValidParentheses(self, s):
        """
        :type s: str
        :rtype: int
        """
        if not s:
            return 0
        length = len(s)
        dp = [0 for __ in range(length)]
        for i in range(1, length):
            if s[i] == ")":
                j = i - 1 - dp[i - 1]
                if j >= 0 and s[j] == "(":
                    dp[i] = dp[i - 1] + 2
                    if j - 1 >= 0:
                        dp[i] += dp[j - 1]
        return max(dp)

if __name__ == "__main__":
    assert Solution().longestValidParentheses("(()())()") == 4
    assert Solution().longestValidParentheses("()") == 2
    assert Solution().longestValidParentheses(")(())") == 4
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Search in Rotated Sorted Array

## 原题

把一个严格升序的数组进行旋转，如 $[0,1,2,3,4,5]$ 旋转3位成为 $[3,4,5,0,1,2]$ 。在这样的数组中找到目标数字。如果存在返回下标，不存在返回-1。

注意点：

- 数组中不存在重复的数字
- 不知道数组旋转了多少位

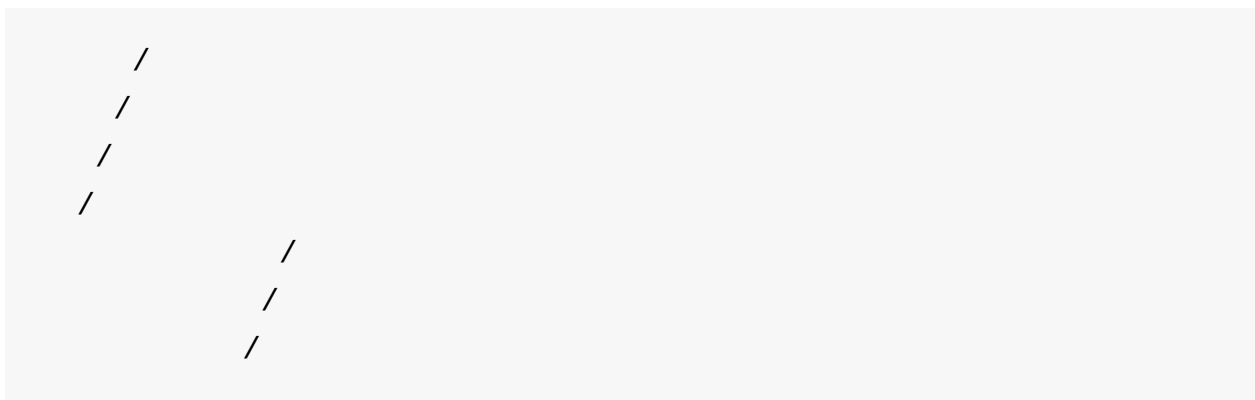
例子：

输入:  $\text{nums} = [4, 5, 6, 7, 0, 1, 2]$ ,  $\text{target} = 6$  输出: 2

输入:  $\text{nums} = [4, 5, 6, 7, 0, 1, 2]$ ,  $\text{target} = 3$  输出: -1

## 解题思路

采用了二分搜索，不过旋转后的数组要讨论的情况增多了。其实旋转后的数组的大小关系一般如下图：



先通过中点与左顶点的关系来分类讨论中点落在了哪一部分，如果在左半边，则继续讨论目标数在中点的左边还是右边；如果在右半边，同样讨论目标数的位置。同时需要注意特殊情况只剩下两个数时，例如 $[3,1]$ ，这时求出的中点也是3，如果中

点不匹配，应考虑1。这种情况不好与上面的情况合并，单独列出。

### AC源码

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid

            if nums[mid] > nums[left]:
                if nums[left] <= target <= nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            elif nums[mid] < nums[left]:
                if nums[mid] <= target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
            else:
                left += 1
        return -1

if __name__ == "__main__":
    assert Solution().search([4, 5, 6, 7, 0, 1, 2], 4) == 0
    assert Solution().search([4, 5, 6, 7, 0, 1, 2], 7) == 3
    assert Solution().search([4, 5, 6, 7, 0, 1, 2], 0) == 4
    assert Solution().search([4, 5, 6, 7, 0, 1, 2], 2) == 6
    assert Solution().search([3, 1], 3) == 0
    assert Solution().search([3, 1], 1) == 1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Search for a Range

### 原题

在一个有序的数组中找到找到目标数字的起始坐标和结束坐标，如果不存在则返回  $[-1, -1]$ 。

注意点：

- 时间复杂度为 $\log(n)$

例子：

输入:  $\text{nums} = [5, 7, 7, 8, 8, 10]$ ,  $\text{target} = 8$  输出:  $[3, 4]$

输入:  $\text{nums} = [5, 7, 7, 8, 8, 10]$ ,  $\text{target} = 6$  输出:  $[-1, -1]$

### 解题思路

通过两次二分法来获的两个边界，左边界的标志是中间数字为目标数字且它前面的数字不存在或不是目标数字；右边界的标志是中间数字为目标数字且它后面的数字不存在或不是目标数字。如果没有找到左边界就可以确定不存在目标数字。需要注意边界的操作。

### AC源码

```
class Solution(object):
    def searchRange(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        result = []
        length = len(nums)
```

```

        start = 0
        end = length
        while start < end:
            mid = (start + end) // 2
            if nums[mid] == target and (mid == 0 or nums[mid - 1]
] != target):
                result.append(mid)
                break
            if nums[mid] < target:
                start = mid + 1
            else:
                end = mid
        if not result:
            return [-1, -1]

        end = length
        while start < end:
            mid = (start + end) // 2
            if nums[mid] == target and (mid == length - 1 or num
s[mid + 1] != target):
                result.append(mid)
                break
            if nums[mid] <= target:
                start = mid + 1
            else:
                end = mid

        return result

if __name__ == "__main__":
    assert Solution().searchRange([5, 7, 7, 8, 8, 10], 8) == [3,
4]
    assert Solution().searchRange([5, 7, 7, 8, 8, 10], 5) == [0,
0]
    assert Solution().searchRange([5, 7, 7, 8, 8, 10], 7) == [1,
2]
    assert Solution().searchRange([5, 7, 7, 8, 8, 10], 10) == [5
, 5]

```





欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Search Insert Position

---

### 原题

在一个有序数组中，如果目标数字存在，则返回它的下标，否则返回它应该插入位置的下标值。

注意点：

- 数组中没有重复数字

例子：

输入: nums = [1, 3, 5, 6], target = 5 输出: 2

输入: nums = [1, 3, 5, 6], target = 2 输出: 1

### 解题思路

又是一道二分搜索的题。分以下几种情况：如果当前数字是目标数字，或者当前数字大于目标数字而它之前的数字小于目标数字，则返回当前下标；如果当前数字为最后一个且它比目标数字小，则返回当前下标的后一位。

### AC源码

```
class Solution(object):
    def searchInsert(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        length = len(nums)
        start = 0
        end = length
        while start < end:
            mid = (start + end) // 2
            if nums[mid] == target or (nums[mid] > target
                and (mid == 0 or nums[mid - 1] < target)):
                return mid
            if mid == length - 1 and nums[mid] < target:
                return mid + 1
            if nums[mid] < target:
                start = mid + 1
            else:
                end = mid

if __name__ == "__main__":
    assert Solution().searchInsert([1, 3, 5, 6], 5) == 2
    assert Solution().searchInsert([1, 3, 5, 6], 2) == 1
    assert Solution().searchInsert([1, 3, 5, 6], 7) == 4
    assert Solution().searchInsert([1, 3, 5, 6], 0) == 0
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Valid Sudoku

## 原题

判断一个给出的数独模型是否符合要求。

注意点：

- 该模型不需要填完整，没有填写的用"."表示
- 不用关心该模型是否有解，只需要判断现有情况

例子：

输入: board=

`[["..4...63.", ".....", "5.....9.", "...56....", "4.3.....1", "...7.....", "...5.....", ".....", "....."]]` 输

出: false

## 解题思路

用三个列表表示横向、纵向和小正方形的情况。特别需要注意的是小正方形内的元素的表示方法：`board[i/3*3+j/3][i%3*3+j%3]`。

## AC源码

```
class Solution(object):
    def isValidSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: bool
        """
        point = "."
        for i in range(9):
            row = []
            column = []
            square = []
            for j in range(9):
                element = board[i][j]
                if element != point:
                    if element in row:
                        return False
                    else:
                        row.append(element)

                element = board[j][i]
                if element != point:
                    if element in column:
                        return False
                    else:
                        column.append(element)

                element = board[i // 3 * 3 + j // 3][i % 3 * 3 +
j % 3]
                if element != point:
                    if element in square:
                        return False
                    else:
                        square.append(element)

            return True
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Sudoku Solver

## 原题

通过程序来解决数独问题。

注意点：

- 有且只有唯一解

例子：

输入：

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

输出：

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

## 解题思路

这次不是像 [Valid Sudoku](#) 一样只需要判断数独是否合法，而是要给出确切的解，但还是可以复用一下判断是否符合数独规则的方法。采用递归来解决，递归的终止条件有：

- 遍历完所有的格子
- 违反数独的规则

遍历每个格子，如果已经有数字，则继续递归下一个位置；如果没有，就放入一个数字并检查是否符合规则，如果符合则继续递归，如果递归失败，则要进行回溯，将数字改为空格，继续尝试别的数字。直到放入一个合适的值后再去填下一个位置。

## AC源码

```
class Solution(object):
    def solveSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        for row in range(9):
            board[row] = list(board[row])
            self.recursive(0, 0, board)
        for row in range(9):
            board[row] = "".join(board[row])

    def recursive(self, i, j, board):
        if j >= 9:
            return self.recursive(i + 1, 0, board)
        if i == 9:
            return True

        if board[i][j] == ".":
            for num in range(1, 10):
                num_str = str(num)
```



```

        if all([board[i][col] != num_str for col in
                 range(9)]) and all([board[row][j] != num
_str for row in range(9)]) and all(
            [board[i // 3 * 3 + count // 3][j // 3 * 3 +
count % 3] != num_str for count in
             range(9)]):
            board[i][j] = num_str
            if not self.recursive(i, j + 1, board):
                board[i][j] = "."
            else:
                return True
        else:
            return self.recursive(i, j + 1, board)
    return False

if __name__ == "__main__":
    sudoku = ["..9748...", "7.....", ".2.1.9...", "..7...24."
, ".64.1.59.", ".98...3..", "...8.3.2.", ".....6",
              "...2759.."]
    Solution().solveSudoku(sudoku)
    assert sudoku == ['519748632', '783652419', '426139875', '35
7986241', '264317598', '198524367', '975863124',
                      '832491756', '641275983']

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Count and Say

---

### 原题

把一个数字用几个几的形式表示出来。如2就是1个2，即12。对12进行数数得到1112，依次类推。假设初始数字是1，求第n个数是什么。起始5个数字为1, 11, 21, 1211, 111221, ...

注意点：

- 题目中的数字都用字符串表

例子：

输入:  $n = 5$  输出: 111221

### 解题思路

用一个下标来表示当前统计的字符的起始位置，一个计数器来表示该字符的数目。不断读取直到字符不相等，添加到结果集中，更新起始位置和计数器。下面代码中的计数器用下标相减代替。

### AC源码

```
class Solution(object):
    def countAndSay(self, n):
        """
        :type n: int
        :rtype: str
        """
        result = "1"
        for __ in range(1, n):
            result = self.getNext(result)
        return result

    def getNext(self, s):
        result = []
        start = 0
        while start < len(s):
            curr = start + 1
            while curr < len(s) and s[start] == s[curr]:
                curr += 1
            result.extend((str(curr - start), s[start]))
            start = curr
        return "".join(result)

if __name__ == "__main__":
    assert Solution().countAndSay(4) == "1211"
    assert Solution().countAndSay(5) == "111221"
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Combination Sum

---

### 原题

在一个集合（没有重复数字）中找到和为特定值的所有组合。

注意点：

- 所有数字都是正数
- 组合中的数字要按照从小到大的顺序
- 原集合中的数字可以出现重复多次
- 结果集中不能够有重复的组合
- 虽然是集合，但传入的参数类型是列表

例子：

输入: candidates = [2, 3, 6, 7], target = 7 输出: [[2, 2, 3], [7]]

### 解题思路

采用回溯法。由于组合中的数字要按序排列，我们先将集合中的数排序。依次把数字放入组合中，因为所有数都是正数，如果当前和已经超出目标值，则放弃；如果和为目标值，则加入结果集；如果和小于目标值，则继续增加元素。由于结果集中不允许出现重复的组合，所以增加元素时只增加当前元素及之后的元素。

### AC源码

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        if not candidates:
            return []
        candidates.sort()
        result = []
        self.combination(candidates, target, [], result)
        return result

    def combination(self, candidates, target, current, result):
        s = sum(current) if current else 0
        if s > target:
            return
        elif s == target:
            result.append(current)
            return
        else:
            for i, v in enumerate(candidates):
                self.combination(candidates[i:], target, current
+ [v], result)

if __name__ == "__main__":
    assert Solution().combinationSum([2, 3, 6, 7], 7) == [[2, 2,
3], [7]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Combination Sum II

---

### 原题

在一个数组（存在重复值）中寻找和为特定值的组合。

注意点：

- 所有数字都是正数
- 组合中的数字要按照从小到大的顺序
- 原数组中的数字只可以出现一次
- 结果集中不能够有重复的组合

例子：

输入: candidates = [10, 1, 2, 7, 6, 1, 5], target = 8 输出: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

### 解题思路

这道题和 [Combination Sum](#) 极其相似，主要的区别是Combination Sum中的元素是没有重复的，且每个元素可以使用无限次；而这题中的元素是有重复的，每个元素最多只能使用一次。最开始的想法是加下一个元素时不要考虑当前元素，且把结果用集合存储以防止重复的组合出现，但结果超时了。改用手动把所有与当前元素相等的元素都去掉即可。

### AC源码

```

class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        if not candidates:
            return []
        candidates.sort()
        result = []
        self.combination(candidates, target, [], result)
        return result

    def combination(self, candidates, target, current, result):
        s = sum(current) if current else 0
        if s > target:
            return
        elif s == target:
            result.append(current)
            return
        else:
            i = 0
            while i < len(candidates):
                self.combination(candidates[i + 1:], target, current + [candidates[i]], result)
                # ignore repeating elements
                while i + 1 < len(candidates) and candidates[i] == candidates[i + 1]:
                    i += 1
                i += 1

if __name__ == "__main__":
    assert Solution().combinationSum2([10, 1, 2, 7, 6, 1, 5], 8) == [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。





# LeetCode解题之First Missing Positive

---

## 原题

找出一个无序数组中缺少的最小的正整数。

注意点：

- 时间复杂度为 $O(n)$
- 只能使用常数级额外空间

例子：

输入: `nums = [1, 2, 0]` 输出: 3

输入: `nums = [3,4,-1,1]` 输出: 2

## 解题思路

由于只需要找出缺少的第一个正整数，我们不妨把所有正数放到对应的位置，再找到第一个位置不匹配的地方原本应该放哪个数。如上面的例子`[1,2,0]`就已经排列好了，而`[3,4,-1,1]`应变为`[1,-1,3,4]`。分别遍历这两个数组，找到`nums[i] != i+1`的位置，如果所有位置都符合，说明所有的数组成了从1开始的连续正整数。进行排列的方法就是依次遍历每个数字，把它们放到应该放置的位子。

## AC源码

```
class Solution(object):
    def firstMissingPositive(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 1
        i = 0
        length = len(nums)
        while i < length:
            current = nums[i]
            if current <= 0 or current > length or nums[current
- 1] == current:
                i += 1
            else:
                nums[current - 1], nums[i] = nums[i], nums[curre
nt - 1]

        for i in range(length):
            if nums[i] != i + 1:
                return i + 1
        return length + 1

if __name__ == "__main__":
    assert Solution().firstMissingPositive([1, 2, 0]) == 3
    assert Solution().firstMissingPositive([1, 2, 3]) == 4
    assert Solution().firstMissingPositive([3, 4, -1, 1]) == 2
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Trapping Rain Water

### 原题

计算一个凹凸不平的模型中可以存放多少的雨水。以下图为例，黑色的地方是砖块，蓝色的地方是积水。



注意点：

- 给的参数数组表示的是砖块的高度（它自身也要占面积），不只是边
- 不会存在负数的情况

例子：

输入: height = [0,1,0,2,1,0,1,3,2,1,2,1] 输出: 6

注：具体看上图

### 解题思路

这题与 [Container With Most Water](#) 非常相似，但那题只要求出面积最大的积水处，而现在要找到所有的积水处的总和。现在考虑任意的一个块砖它上面最终的积水高度是如何求的，我们需要找到它左右两边的最高的砖块，而它最终的高度就是这两个砖块中较矮的那个。所有我们需要先遍历来得到每个砖块左右最高砖块的高度，最后根据这两个高度来确定最终的高度。下面的代码先从后往前遍历得到右边的最高高度，然后从前往后遍历得到左边的最高高度，同时得到两者中小的一个加入总和。

## AC源码

```
class Solution(object):
    def trap(self, height):
        """
        :type height: List[int]
        :rtype: int
        """
        if not height:
            return 0
        length = len(height)
        maxh = [0 for __ in range(length)]
        h = height[length - 1]
        for i in range(length - 2, -1, -1):
            maxh[i] = h
            h = max(h, height[i])

        h = height[0]
        result = 0
        for i in range(1, length - 1):
            h = max(h, height[i])
            result += max(0, min(h, maxh[i]) - height[i])
        return result

if __name__ == "__main__":
    assert Solution().trap([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])
    == 6
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Multiply Strings

---

### 原题

将两个用字符串表示的数进行乘法操作并返回字符串结果。

注意点：

- 给的数是非负整数
- 数字可以无穷大

例子：

输入: num1 = "123", num2 = "20" 输出: "2460"

### 解题思路

根据笔算乘法的公式来看，乘法操作分解开来其实就是先进行每个位的乘法操作，然后将所有结果进行加法操作。首先明确一个 $m$ 位的数乘以一个 $n$ 位的数最多为 $m+n$ 位（都是9的时候试一下）。其次后面的0相等的数在相加时是末尾对齐的。如 $123 \times 456$ ，我们可以看到 $1 \times 6$ 、 $2 \times 5$ 和 $3 \times 4$ 在进行加法的时候是末尾对齐的，我们可以在进行第一轮乘法的时候将这些数先加起来，而后面的零通过在列表中的位置来表示。再用一个循环进行进位加法，最后把开头多余的0去掉。具体步骤看下面的例子：

123\*456

100	400
20	50
3	6

[3\*6, 2\*6+3\*5, 1\*6+2\*5+3\*4, 2\*4+1\*5, 1\*4, 0]

[18, 27, 28, 13, 4, 0]

[8, 27+1, 28, 13, 4, 0]

[8, 8, 28+2, 13, 4, 0]

[8, 8, 0, 13+3, 4, 0]

[8, 8, 0, 6, 5, 0]

"880650" --> "056088"

"56088"

## AC源码

```
class Solution(object):
    def multiply(self, num1, num2):
        """
        :type num1: str
        :type num2: str
        :rtype: str
        """
        num1 = num1[::-1]
        num2 = num2[::-1]
        length1 = len(num1)
        length2 = len(num2)
        temp = [0 for __ in range(length1 + length2)]
        # Do multiply
        for i in range(length1):
            for j in range(length2):
                temp[i + j] += int(num1[i]) * int(num2[j])
        carry = 0
        digits = []
        # Do plus
        for num in temp:
            s = carry + num
            carry = s // 10
            digits.append(str(s % 10))
        result = "".join(digits)[::-1]
        # Remove the surplus zero
        sub_index = 0
        for i in range(length1 + length2 - 1):
            if result[i] == "0":
                sub_index += 1
            else:
                break
        result = result[sub_index:]
        return result

if __name__ == "__main__":
    assert Solution().multiply("120", "20000") == 2400000
    assert Solution().multiply("0", "3421") == 0
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Wildcard Matching

## 原题

万用字符通配符的字符串匹配判断。"?"表示一个任意的字符，"\*"表示任意多个字符。判断目标字符串是否与模式串相匹配。

注意点：

- 整个目标字符串要全部匹配进去，不能只匹配部分

例子：

输入: s = "abc", p = "a\*b\*e"

输出: False

## 解题思路

与 [Regular Expression Matching](#) 同类型的题，不过换了不同类型的通配符。刚开始写了一个动态规划，结果超时了，转换了下思路，改用回溯法。用两个指针分别来表示目标串和模式串遍历到的当前位置，如果两个字符相等(考虑"?"通配符)，则继续前进，如果是"\*"通配符，那么要记录下目标字符串当前位置，及模式串下一个位置，现在假设的是"\*"用来匹配0个字符，继续尝试匹配，如果后面出现不匹配的情况，那么应该回退到这两个位置（目标串的位置要向后移一位，否则会不断回退到原来的位置），发生一次回退，代表着"\*"要多匹配掉一个字符。按照这种方式不断尝试匹配，直到目标串都已经匹配掉或者匹配失败（匹配串中没有"\*"，且不能匹配整个目标串）。这时候要看看匹配串是否还有剩余除了"\*"以外的字符。如果最终匹配串都全部遍历完了，那么说明匹配成功。

## AC源码

```
class Solution(object):  
    def isMatch(self, s, p):
```

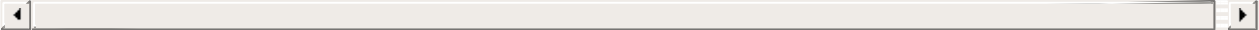
```

"""
:type s: str
:type p: str
:rtype: bool
"""
p_index, s_index, last_s_index, last_p_index = 0, 0, -1,
-1
while s_index < len(s):
    # Normal match including '?'
    if p_index < len(p) and (s[s_index] == p[p_index] or
p[p_index] == '?'):
        s_index += 1
        p_index += 1
    # Match with '*'
    elif p_index < len(p) and p[p_index] == '*':
        p_index += 1
        last_s_index = s_index
        last_p_index = p_index
    # Not match, but there is a '*' before
    elif last_p_index != -1:
        last_s_index += 1
        s_index = last_s_index
        p_index = last_p_index
    # Not match and there is no '*' before
    else:
        return False
    # Check if there is still character except '*' in the pa
ttern
    while p_index < len(p) and p[p_index] == '*':
        p_index += 1
    # If finish scanning both string and pattern, then it ma
tches well
    return p_index == len(p)

if __name__ == "__main__":
    assert Solution().isMatch("aa", "a") == False
    assert Solution().isMatch("aa", "aa") == True
    assert Solution().isMatch("aaa", "aa") == False
    assert Solution().isMatch("aa", "*") == True

```

```
assert Solution().isMatch("aa", "a*") == True
assert Solution().isMatch("ab", "?*") == True
assert Solution().isMatch("aab", "c*a*b") == False
```



欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Jump Game II

---

## 原题

数组中的每个值表示在当前位置最多能向前面跳几步，判断至少跳几步能够跳到最后。

注意点：

- 所有的数字都是正数
- 跳的步数可以比当前的值小
- 保证所有的测试用例都能够跳到最后

例子：

输入: `nums = [2, 3, 1, 1, 4]`

输出: 2

## 解题思路

这是在 [Jump Game](#) 之上给出的问题，题目已经保证能够跳到最后。遍历数组，起始到当前坐标所有跳跃方式能够到达的最远距离是`reach`，我们跳`n`步能到达的最远距离用`longest`表示，如果`longest`不能到达当前坐标，说明就要多跳一步了，直接跳到当前坐标之前的点能够跳到的最远位置。

## AC源码

```
class Solution(object):
    def jump(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        length = len(nums)
        counter = 0
        longest = 0
        reach = 0
        for i in range(length):
            if longest < i:
                counter += 1
                longest = reach
            reach = max(reach, nums[i] + i)
        return counter

if __name__ == "__main__":
    assert Solution().jump([2, 3, 1, 1, 4]) == 2
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Permutations

---

## 原题

输出一个没有重复数字的数组的全排列。

注意点：

- 不用考虑重复数字的情况

例子：

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

## 解题思路

这道题比较单间，采用递归把数组中的数字依次加入当前数组`current`来进行排列组合。

## AC源码

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []
        self.get_permute([], nums, result)
        return result

    def get_permute(self, current, num, result):
        if not num:
            result.append(current + [])
            return
        for i, v in enumerate(num):
            current.append(num[i])
            self.get_permute(current, num[:i] + num[i + 1:], result)
            current.pop()

if __name__ == "__main__":
    assert Solution().permute([1, 2, 3]) == [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Permutations II

---

### 原题

输出一个有重复数字的数组的全排列。

注意点：

- 重复数字的可能导致重复的排列

例子：

输入: `nums = [1, 2, 1]` 输出: `[[1, 1, 2], [1, 2, 1], [2, 1, 1]]`

### 解题思路

这道题是上一题 [Permutations](#) 的加强版，现在要考虑重复的数字了，采用了偷懒的办法，先把数组排序，遍历时直接无视重复的数字，在原来的基础上只要添加两行代码。

### AC源码



```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []
        nums.sort()
        self.get_permute([], nums, result)
        return result

    def get_permute(self, current, num, result):
        if not num:
            result.append(current + [])
            return
        for i, v in enumerate(num):
            if i - 1 >= 0 and num[i] == num[i - 1]:
                continue
            current.append(num[i])
            self.get_permute(current, num[:i] + num[i + 1:], result)
            current.pop()

if __name__ == "__main__":
    assert Solution().permuteUnique([1, 2, 1]) == [[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Rotate Image

## 原题

将一个矩阵顺时针旋转90度。

注意点：

- 最好不要申请额外空间

例子：

输入: matrix = [[1, 2, 3], [8, 9, 4], [7, 6, 5]]

输出: [[7, 8, 1], [6, 9, 2], [5, 4, 3]]

## 解题思路

如果可以申请额外空间，哪怕一个临时的变量，那只要找一下规律还是很容易实现的。但题目要求最好不要申请额外空间，这就需要技巧了，看到一个很巧妙的方法：先将矩阵沿着对角线翻转，再上下翻转，就可以实现顺时针旋转90度的效果。具体看如下的例子：

1 2 3		5 4 3		7 8 1
8 9 4	->	6 9 2	->	6 9 2
7 6 5		7 8 1		5 4 3

两次翻转对应的坐标需要细心，不然很容易搞错。

## AC源码

```
class Solution(object):
    def rotate(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: void Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)
        for row in range(n):
            for column in range(n - row):
                matrix[row][column], matrix[n - 1 - column][n - 1 - row] = matrix[n - 1 - column][n - 1 - row], \
                    matrix[row][column]
        for row in range(n // 2):
            for column in range(n):
                matrix[row][column], matrix[n - 1 - row][column] = matrix[n - 1 - row][column], matrix[row][column]
        # No need, just to test
        return matrix

if __name__ == "__main__":
    assert Solution().rotate([[1, 2, 3], [8, 9, 4], [7, 6, 5]]) == [[7, 8, 1], [6, 9, 2], [5, 4, 3]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Group Anagrams

---

### 原题

将所含字母相同，但排列顺序不同的字符串归并到一起。

注意点：

- 所有输入的字符都是小写的
- 返回结果中每个组的字符串都要按照字典序排列

例子：

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

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

### 解题思路

将每个字符串排序，并把排序后相同的字符串归为一组，将每组字符串排序后即为首要的结果。

### AC源码

```
class Solution(object):
    def groupAnagrams(self, strs):
        """
        :type strs: List[str]
        :rtype: List[List[str]]
        """
        map = {}
        for i, v in enumerate(strs):
            target = "".join(sorted(v))
            if target not in map:
                map[target]=[v]
            else:
                map[target].append(v)

        result = []
        for value in map.values():
            result += [sorted(value)]
        return result

if __name__ == "__main__":
    assert Solution().groupAnagrams(["eat", "tea", "tan", "ate",
                                     "nat", "bat"]) == [['nat', 'tan'], ['bat'],

                                                         ['ate', 'eat', 'tea']]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Pow(x, n)

### 原题

求 $x$ 的 $n$ 次幂。

注意点：

- $n$ 是负数时需要取相反数

例子：

输入:  $x = 2, n = -1$

输出: 0.5

输入:  $x = 2.1, n = 2$

输出: 4.41

### 解题思路

最简答的方法就是把 $n$ 个 $x$ 直接做乘法，但这样要进行 $(n-1)$ 次运算。现在以  $2^{**}8$  (表示2的8次方)作为例子，需要进行7次乘法，但如果当做  $(2^{**}2)^{**}4 - > ((2^{**}2)^{**}2)^{**}2$  来计算就只要做3次乘法。即当 $n$ 为奇数时，直接乘上当前的 $x$ ，偶数时 $x$ 变为 $x$ 的平方， $n$ 除以2。这样就可以较快速的求出结果。当 $n$ 为负数时要取倒数。

### AC源码

```
class Solution(object):
    def myPow(self, x, n):
        """
        :type x: float
        :type n: int
        :rtype: float
        """
        flag = 1 if n >= 0 else -1
        result = 1
        n = abs(n)
        while n > 0:
            if n & 1 == 1:
                result *= x
            n >>= 1
            x *= x
        if flag < 0:
            result = 1 / result
        return result

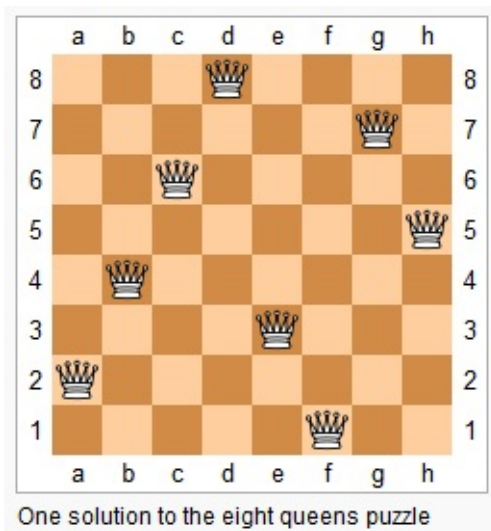
if __name__ == "__main__":
    assert Solution().myPow(2, -1) == 0.5
    assert Solution().myPow(2.1, 2) == 4.41
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

## LeetCode 解题之N-Queens

### 原题

经典的八皇后问题的一般情况，用Python怎样来快速地解决呢？



注意点：

- 皇后用"Q"表示，空白用"."表示

例子：

输入:  $n = 4$

输出:

```
[ ['..Q..',
  '...Q',
  'Q...',
  '..Q.'],

  ['..Q.',
  'Q...',
  '...Q',
  '.Q..']]
```



## 解题思路

用三个数组来表示列、正反对角线的占用情况。一行行的遍历，如果没有冲突就把相应的位置置为占用，继续处理下一行，并记录改行的皇后放在了哪一列，当皇后都放完后，根据记录的列号来拼出结果。进行回溯时要把占用的位置还回去。对角线位置的计算要小心（尤其是反对角线），可以把顶点带进去计算验证一下。

## AC源码

```
class Solution(object):
    def solveNQueens(self, n):
        """
        :type n: int
        :rtype: List[List[str]]
        """
        self.col = [False] * n
        self.diag = [False] * (2 * n)
        self.anti_diag = [False] * (2 * n)
        self.result = []
        self.recursive(0, n, [])
        return self.result

    def recursive(self, row, n, column):
        if row == n:
            self.result.append(list(map(lambda x: '.' * x + 'Q'
+ '.' * (n - 1 - x), column)))
        else:
            for i in range(n):
                if not self.col[i] and not self.diag[row + i] and
not self.anti_diag[n - i + row]:
                    self.col[i] = self.diag[row + i] = self.anti
_diag[n - i + row] = True
                    self.recursive(row + 1, n, column + [i])
                    self.col[i] = self.diag[row + i] = self.anti
_diag[n - i + row] = False

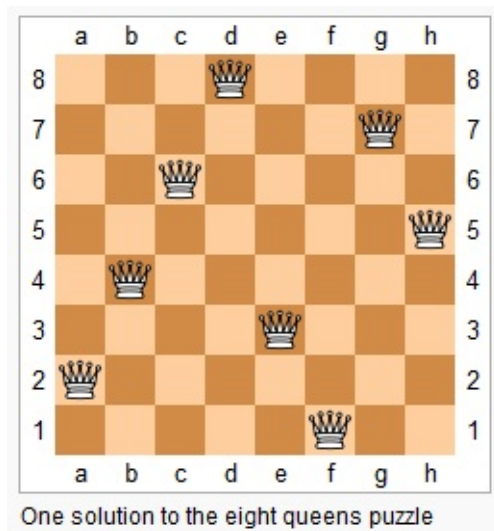
if __name__ == "__main__":
    print(Solution().solveNQueens(5))
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 N-Queens II

### 原题

在 N-Queens 的基础上计算出共有多少种不同的解法。



注意点：

- 只需要计数

例子：

输入:  $n = 8$

输出: 92

### 解题思路

思路与 [N-Queens](#) 一样，不过把原先用来最后拼装的参数之类都去掉了，换了一个计数器来记录数量。

### AC 源码

```
class Solution(object):
    def totalNQueens(self, n):
        """
        :type n: int
        :rtype: int
        """
        self.col = [False] * n
        self.diag = [False] * (2 * n)
        self.anti_diag = [False] * (2 * n)
        self.result = 0
        self.recursive(0, n)
        return self.result

    def recursive(self, row, n):

        if row == n:
            self.result += 1
        else:
            for i in range(n):
                if not self.col[i] and not self.diag[row + i] and
not self.anti_diag[n - i + row]:
                    self.col[i] = self.diag[row + i] = self.anti
_diag[n - i + row] = True
                    self.recursive(row + 1, n)
                    self.col[i] = self.diag[row + i] = self.anti
_diag[n - i + row] = False

if __name__ == "__main__":
    assert Solution().totalNQueens(8) == 92
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Maximum Subarray

## 原题

求一个数组中和最大的子数组。

注意点：

- 需要考虑负数的情况

例子：

输入: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

输出: 6(数组[4, -1, 2, 1]的和)

## 解题思路

又是比较经典的动态规划的题目。主要有以下几个概念，依次计算以第 $k$ 个数作为子数组末尾的最优子数组（和最大）的和 $dp[k]$ ，然后求 $dp$ 中的最大值。那递推关系是怎样的呢，当把下一个数字 $num[k+1]$ 最为末尾数字时，要看它之前与它相连的子数组的和是否是正的，如果是正的，应该加上，否则舍弃。下面的代码把求 $dp$ 和求 $dp$ 中的最大值一起计算了，所以没有额外的数组 $dp$ 。

现在还有一个疑问，就是 $num[k+1]$ 之前与它相连的子数组应该定义为多长，它的起始位置是最靠近它的满足与这个数字相连的子数组的和为负的数字。如 `[-2, 1, -3, 4, -1, 2, 1, -5, 4]` 中-3前子数组的开端是1，-1是4，-5也是4。

## AC源码

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0
        length = len(nums)
        current = nums[0]
        m = current
        for i in range(1, length):
            if current < 0:
                current = 0
            current += nums[i]
            m = max(current, m)
        return m

if __name__ == "__main__":
    assert Solution().maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4
]) == 6
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Spiral Matrix

---

### 原题

将一个矩阵中的内容螺旋输出。

注意点：

- 矩阵不一定是正方形

例子：

输入: matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]

输出: [1, 2, 3, 6, 9, 8, 7, 4, 5]

### 解题思路

控制好当前遍历的边界，不断的向内缩进。需要注意的是，缩到最里面的时候可能会出现以下几种情况：

中心剩下一个数值

—  
|3|  
—

中心横向多个数值

—  
|3 4 5 6|  
—

中心纵向多个数值

—  
|2|

|3|

|4|  
—

分别处理一下即可。

## AC源码

```
class Solution(object):
    def spiralOrder(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        if not matrix:
            return []
        left = top = 0
        right = len(matrix[0]) - 1
        bottom = len(matrix) - 1

        result = []
        while left < right and top < bottom:
            for i in range(left, right):
                result.append(matrix[top][i])
            for i in range(top, bottom):
                result.append(matrix[i][right])
```



```
        result.append(matrix[i][right])
    for i in range(right, left, -1):
        result.append(matrix[bottom][i])
    for i in range(bottom, top, -1):
        result.append(matrix[i][left])
    left += 1
    right -= 1
    top += 1
    bottom -= 1
    if left == right and top == bottom:
        result.append(matrix[top][left])
    elif left == right:
        for i in range(top, bottom + 1):
            result.append(matrix[i][left])
    elif top == bottom:
        for i in range(left, right + 1):
            result.append(matrix[top][i])
    return result

if __name__ == "__main__":
    assert Solution().spiralOrder([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]) == [1, 2, 3, 6, 9, 8, 7, 4, 5]
    assert Solution().spiralOrder([[2], [3]]) == [2, 3]
    assert Solution().spiralOrder([[2, 3]]) == [2, 3]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Jump Game

---

## 原题

数组中的每个值表示在当前位置最多能向前面跳几步，判断给出的数组是否否存在一种跳法跳到最后。

注意点：

- 所有的数字都是正数
- 跳的步数可以比当前的值小

例子：

输入: nums = [2, 3, 1, 1, 4]

输出: True

输入: nums = [3, 2, 1, 0, 4]

输出: False

## 解题思路

先想一下什么时候不能够完成跳跃，在当前位置之前（包括当前位置）能够跳跃到的最远距离就是当前位置，且这时候还没有到终点；什么样的情况就能保证可以跳到终点呢，只要当前最远距离超过终点即可。只要当前的位置没有超过能跳到的最远距离，就可以不断的刷新最远距离来继续前进。

## AC源码

```
class Solution(object):
    def canJump(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        if not nums:
            return False
        length = len(nums)
        index = 0
        longest = nums[0]
        while index <= longest:
            if longest >= length - 1:
                return True
            longest = max(longest, index + nums[index])
            index += 1
        return False

if __name__ == "__main__":
    assert Solution().canJump([2, 3, 1, 1, 4]) == True
    assert Solution().canJump([3, 2, 1, 0, 4]) == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Merge Intervals

---

### 原题

给出多个数据区段，把首尾相连的数据段合并。

注意点：

- 所给的数据段是乱序的

例子：

输入: intervals = [1,3],[2,6],[8,10],[15,18]

输出: [1,6],[8,10],[15,18]

### 解题思路

先把所有的数据段按照起始位置(start)排序，这样可以使可能相连的数据段放到相邻的位置。遍历数据段，并与结果集中最后一个数据段比较能否合并，如果能合并就合并，否则加入结果集。

### AC源码

```
# Definition for an interval.
class Interval(object):
    def __init__(self, s=0, e=0):
        self.start = s
        self.end = e

# To print the result
def __str__(self):
    return "[" + str(self.start) + "," + str(self.end) + "]"

class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[Interval]
        """
        result = []
        if not intervals:
            return result
        intervals.sort(key=lambda x: x.start)
        result.append(intervals[0])
        for interval in intervals[1:]:
            prev = result[-1]
            if prev.end >= interval.start:
                prev.end = max(prev.end, interval.end)
            else:
                result.append(interval)
        return result

if __name__ == "__main__":
    intervals = Solution().merge([Interval(1, 3), Interval(2, 6)
    , Interval(8, 10), Interval(15, 18)])
    for interval in intervals:
        print(interval)
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Insert Interval

---

### 原题

给出多个不重合的数据区段，现在插入一个数据区段，有重合的区段要进行合并。

注意点：

- 所给的区段已经按照起始位置进行排序

例子：

输入: intervals = [2,6],[8,10],[15,18]， newInterval = [13,16]

输出: [2,6],[8,10],[13,18]

### 解题思路

最简单的方式就是复用 [Merge Intervals](#) 的方法，只需先将新的数据区段加入集合即可，但这样效率不高。既然原来的数据段是有序且不重合的，那么我们只需要找到哪些数据段与新的数据段重合，把这些数据段合并，并加上它左右的数据段即可。

### AC源码

```
# Definition for an interval.
class Interval(object):
    def __init__(self, s=0, e=0):
        self.start = s
        self.end = e

# To print the result
def __str__(self):
    return "[" + str(self.start) + "," + str(self.end) + "]"

class Solution(object):
    def insert(self, intervals, newInterval):
        start, end = newInterval.start, newInterval.end
        left = list(filter(lambda x: x.end < start, intervals))
        right = list(filter(lambda x: x.start > end, intervals))
        if len(left) + len(right) != len(intervals):
            start = min(start, intervals[len(left)].start)
            end = max(end, intervals[-len(right) - 1].end)

        return left + [Interval(start, end)] + right

if __name__ == "__main__":
    intervals = Solution().insert([Interval(2, 6), Interval(8, 10)], Interval(15, 18), Interval(13, 16))
    for interval in intervals:
        print(interval)
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Length of Last Word

---

### 原题

找出最后一个单词的长度。

注意点：

- 忽略尾部空格
- 不存在最后一个单词时返回0

例子：

输入: s = "Hello world"

输出: 5

### 解题思路

很简答的一道题，用Python内置函数一行就可以解决 `len(s.strip().split(" ")[-1])`。自己写了一下，从后到前先忽略掉空格，再继续遍历到是空格或者遍历结束，两个者之间就是最后一个单词的长度。

### AC源码

```
class Solution(object):
    def lengthOfLastWord(self, s):
        """
        :type s: str
        :rtype: int
        """
        length = len(s)
        index = length - 1
        while index >= 0 and s[index] == " ":
            index -= 1
        temp = index
        while index >= 0 and s[index] != " ":
            index -= 1
        return temp - index

if __name__ == "__main__":
    assert Solution().lengthOfLastWord(" ") == 0
    assert Solution().lengthOfLastWord(" a") == 1
    assert Solution().lengthOfLastWord(" drfish ") == 6
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Spiral Matrix II

---

### 原题

将一个正方形矩阵螺旋着填满递增的数字。

注意点：

- 无

例子：

输入:  $n = 3$

输出: `[[1, 2, 3], [8, 9, 4], [7, 6, 5]]`

### 解题思路

这道题跟 [Spiral Matrix](#) 正好相反，一个是螺旋着读出数字，一个是螺旋着写入数字，而且这道题还要简单一点，因为形状固定是正方形的，所以只要控制四条边不断向内缩进就可以了。考虑一下奇偶的情况，在奇数的时候要额外加一个中心点。

### AC 源码

```

class Solution(object):
    def generateMatrix(self, n):
        """
        :type n: int
        :rtype: List[List[int]]
        """
        left = top = 0
        right = n - 1
        bottom = n - 1
        num = 1
        result = [[0 for __ in range(n)] for __ in range(n)]
        while left < right and top < bottom:
            for i in range(left, right):
                result[top][i] = num
                num += 1
            for i in range(top, bottom):
                result[i][right] = num
                num += 1
            for i in range(right, left, -1):
                result[bottom][i] = num
                num += 1
            for i in range(bottom, top, -1):
                result[i][left] = num
                num += 1
            left += 1
            right -= 1
            top += 1
            bottom -= 1
        if left == right and top == bottom:
            result[top][left] = num
        return result

if __name__ == "__main__":
    assert Solution().generateMatrix(5) == [[1, 2, 3, 4, 5], [16
, 17, 18, 19, 6], [15, 24, 25, 20, 7],
                                           [14, 23, 22, 21, 8],
                                           [13, 12, 11, 10, 9]]

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Permutation Sequence

---

### 原题

找出由[1,2,3...n]中所有数字组成的序列中第k大的。

注意点：

- n为1-9中某一个数字

例子：

输入: n = 3, k = 3 输出: "213"

### 解题思路

因为n个不同的数字可以组成n!个序列，那么首位确定的序列都有(n-1)!种不同的可能性，而且这些序列都根据首位的大小进行了分组，1...是最小的(n-1)!个，2...是(n-1)!+1到2(n-1)!个，那么现在只需要计算k中有几个(n-1)!就可以确定首位的数字，同样可以通过这样的方法来确定第二位、第三位.....此外，由于列表下标从0开始，所以k要减去1。

### AC源码

```
class Solution(object):
    def getPermutation(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        k -= 1
        factorial = 1
        for i in range(1, n):
            factorial *= i

        result = []
        array = list(range(1, n + 1))
        for i in range(n - 1, 0, -1):
            index = k // factorial
            result.append(str(array[index]))
            array = array[:index] + array[index + 1:]
            k %= factorial
            factorial //= i
        result.append(str(array[0]))
        return "".join(result)

if __name__ == "__main__":
    assert Solution().getPermutation(3, 3) == "213"
    assert Solution().getPermutation(9, 324) == "123685974"
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Rotate List

## 原题

将一个链表中的元素向右旋转 $k$ 个位置。

注意点：

- $k$ 可能非常大
- 最好不要申请额外空间

例子：

输入: list = 1->2->3->4->5->NULL,  $k = 2$

输出: 4->5->1->2->3->NULL

## 解题思路

如果能有链表的长度，就不用担心 $k$ 非常大而不断的循环旋转了。所谓的旋转其实就是在链表中间断开，首尾相连。在获取链表长度的时候顺便把链表的首尾连起来。注意断开的位置是在倒数第 $k$ 个之前。

## AC源码

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

    def myprint(self):
        print(self.val)
        if self.next:
            self.next.myprint()
```



```
class Solution(object):
    def rotateRight(self, head, k):
        """
        :type head: ListNode
        :type k: int
        :rtype: ListNode
        """
        if not head:
            return []
        curr = head
        length = 1
        while curr.next:
            curr = curr.next
            length += 1
        curr.next = head
        cur = head
        shift = length - k % length
        while shift > 0:
            curr = curr.next
            shift -= 1
        result = curr.next
        curr.next = None
        return result

if __name__ == "__main__":
    l1 = ListNode(1)
    l2 = ListNode(2)
    l3 = ListNode(3)
    l4 = ListNode(4)
    l5 = ListNode(5)
    l1.next = l2
    l2.next = l3
    l3.next = l4
    l4.next = l5
    result = Solution().rotateRight(l1, 2)
    result.myprint()
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Unique Paths

## 原题

机器人从起点到终点有多少条不同的路径，只能向右或者向下走。



注意点：

- 格子大小最大为100\*100

例子：

输入:  $m = 3, n = 7$

输出: 28

## 解题思路

很常见的小学生奥数题，可以用排列组合来求解，一共要走 $(m-1)+(n-1)$ 步，其中 $(m-1)$ 步向下， $(n-1)$ 向右，且有公式  $mCn = n!/m!(n-m)!$ ，那么可以用下面的代码求解：

```
import math
class Solution(object):
    def uniquePaths(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        m -= 1
        n -= 1
        return math.factorial(m+n) / (math.factorial(n) * math.factorial(m))
```

当然了，更常见的一种做法就是动态规划，要到达一个格子只有从它上面或者左边的格子走过来，递推关系式： $dp[i][j]=dp[i-1][j]+dp[i][j-1]$ 。初始化条件是左边和上边都只有一条路径，索性在初始化时把所有格子初始化为1。

## AC源码

```
class Solution(object):
    def uniquePaths(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        dp = [[1 for __ in range(n)] for __ in range(m)]
        for i in range(1, n):
            for j in range(1, m):
                dp[j][i] = dp[j - 1][i] + dp[j][i - 1]
        return dp[m - 1][n - 1]

if __name__ == "__main__":
    assert Solution().uniquePaths(3, 7) == 28
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Unique Paths II

### 原题

如果道路上有障碍，机器人从起点到终点有多少条不同的路径，只能向右或者向下走。0表示道路通行，1表示有障碍。

注意点：

- 起点如果也有障碍，那就无法出发了

例子：

输入: [ [0,0,0], [0,1,0], [0,0,0] ]

输出: 2

### 解题思路

思路跟 [Unique Paths](#) 是一样的，不过要分类讨论一下障碍的情况，如果当前格子是障碍，那么到达该格子的路径数目是0，因为无法到达，如果是普通格子，那么由左边和右边的格子相加。

### AC源码

```

class Solution(object):
    def uniquePathsWithObstacles(self, obstacleGrid):
        """
        :type obstacleGrid: List[List[int]]
        :rtype: int
        """
        if obstacleGrid[0][0] == 1:
            return 0
        m = len(obstacleGrid)
        n = len(obstacleGrid[0])
        dp = [[0 for __ in range(n)] for __ in range(m)]
        dp[0][0] = 1
        for i in range(1, m):
            dp[i][0] = dp[i - 1][0] if obstacleGrid[i][0] == 0 e
lse 0
        for j in range(1, n):
            dp[0][j] = dp[0][j - 1] if obstacleGrid[0][j] == 0 e
lse 0
        for i in range(1, m):
            for j in range(1, n):
                if obstacleGrid[i][j] == 1:
                    dp[i][j] = 0
                else:
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
        return dp[m - 1][n - 1]

if __name__ == "__main__":
    assert Solution().uniquePathsWithObstacles([
        [0, 0, 0],
        [0, 1, 0],
        [0, 0, 0]
    ]) == 2

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。





## LeetCode解题之Minimum Path Sum

---

### 原题

从一个矩阵的左上角出发到右下角，只能向右或向下走，找出哪一条路径上的数字之和最小。

注意点：

- 所有数字都是非负的

例子：

输入: `[[1, 2, 4], [2, 4, 1], [3, 2, 1]]`

输出: 9

### 解题思路

思路跟 [Unique Paths](#) 相似，不过从计算到达该点有多少种走法转变为求最小和为多少。找出上面和左边格子的最小值加上当前格子中的数字即可。为了节省空间，把二维dp降为一维的，因为是求最优解，前面的中间值可以抛弃。

### AC源码

```
class Solution(object):
    def minPathSum(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        m = len(grid)
        n = len(grid[0])

        dp = [0 for __ in range(n)]
        dp[0] = grid[0][0]
        for j in range(1, n):
            dp[j] = dp[j - 1] + grid[0][j]
        for i in range(1, m):
            dp[0] += grid[i][0]
            for j in range(1, n):
                dp[j] = min(dp[j], dp[j - 1]) + grid[i][j]
        return dp[-1]

if __name__ == "__main__":
    assert Solution().minPathSum([
        [1, 2, 4],
        [2, 4, 1],
        [3, 2, 1]]) == 9
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Valid Number

## 原题

判断一个字符串是否是数值类型的。这里的数值类型除了一般要考虑的小数、正负数外还要考虑科学计数法e，如"-3.2e-23"是数值类型的。

注意点：

- 小数点前和后没有数字都是合法的
- 科学计数法后面也可以是负数

例子：

输入: s = "-3.2e-23"

输出: True

## 解题思路

比较恶心的一道题，没有明确给出定义，给了一些例子，需要自己不断去尝试。首先要把前后的空字符给去掉。然后依次考虑符号、数字、小数点、数字，如果有这些中连续的几个，表示目前是一个普通的数值。继续判断"e"（注意大小写都可以），接着判断符号、数字，如果e后面没有数字，那么这是一个不正常的科学类数值。最后根据三种情况来综合判断，要满足目标是一个数值类型，那么首先要保证e前面的数是正常的，如果有e的话，要保证它后面的数也是正常的，最后要保证整个字符串都已经遍历玩了，如果没有说明中间出现了一些异常的字符或者末尾多了一些多余的字符。

## AC源码

```
class Solution(object):  
    def isNumber(self, s):  
        """
```

```

:type s: str
:rtype: bool
"""
s = s.strip()
length = len(s)
index = 0
# Deal with symbol
if index < length and (s[index] == '+' or s[index] == '-'):
    index += 1
is_normal = False
is_exp = True
# Deal with digits in the front
while index < length and s[index].isdigit():
    is_normal = True
    index += 1
# Deal with dot and digits behind it
if index < length and s[index] == '.':
    index += 1
    while index < length and s[index].isdigit():
        is_normal = True
        index += 1
# Deal with 'e' and number behind it
if is_normal and index < length and (s[index] == 'e' or
s[index] == 'E'):
    index += 1
    is_exp = False
    if index < length and (s[index] == '+' or s[index] =
= '-'):
        index += 1
        while index < length and s[index].isdigit():
            index += 1
            is_exp = True
# Return true only deal with all the characters and the
part in front of and behind 'e' are all ok
return is_normal and is_exp and index == length

if __name__ == "__main__":
    assert Solution().isNumber("3.e-23") == True

```

```
assert Solution().isNumber(".2e81") == True
assert Solution().isNumber("2e10") == True
assert Solution().isNumber(" 0.1") == True
assert Solution().isNumber("1 b") == False
assert Solution().isNumber("3-2") == False
assert Solution().isNumber("abc") == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Minimum Path Sum

---

### 原题

给一个由包含一串数字的列表组成的非负整数加上一。

注意点：

- 列表前面的数字表示高位
- 注意最高位也可能进位

例子：

输入: [1, 2, 3, 4, 9]

输出: [1, 2, 3, 5, 0]

### 解题思路

从低位到高位，如果后一位有进位的话，那么该位要加上一，否则退出循环。如果最高位也进位，那么在列表前要插入一个一。

### AC源码

```
class Solution(object):
    def plusOne(self, digits):
        """
        :type digits: List[int]
        :rtype: List[int]
        """
        carry = 1
        for i in range(len(digits) - 1, -1, -1):
            digits[i] += carry
            if digits[i] < 10:
                carry = 0
                break
            else:
                digits[i] -= 10
        if carry == 1:
            digits.insert(0, 1)
        return digits

if __name__ == "__main__":
    assert Solution().plusOne([1, 2, 3, 4, 9]) == [1, 2, 3, 5, 0]
    assert Solution().plusOne([9]) == [1, 0]
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

## LeetCode解题之Add Binary

---

### 原题

对两个二进制的字符串求和。

注意点：

- 最高位进位

例子：

输入: `a = "111", b = "1"`

输出: `"1000"`

### 解题思路

从后往前依次相加，通过二进制来计算该位的值和进位值。如果最高位还有进位要添加一个位，最后把结果翻转。

### AC源码



```
class Solution(object):
    def addBinary(self, a, b):
        """
        :type a: str
        :type b: str
        :rtype: str
        """
        result = []
        carry = val = 0
        if len(a) < len(b):
            a, b = b, a
        lengthA = len(a)
        lengthB = len(b)
        for i in range(lengthA):
            val = carry
            val += int(a[-(i + 1)])
            if i < lengthB:
                val += int(b[-(i + 1)])
            carry, val = val // 2, val % 2
            result.append(str(val))
        if carry:
            result.append(str(carry))
        return "".join(result[::-1])

if __name__ == "__main__":
    assert Solution().addBinary("111", "1") == "1000"
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之Text Justification

## 原题

把一个集合的单词按照每行L个字符存放，不足的在单词间添加空格，每行要两端对齐(即两端都要是单词)，如果空格不能均匀分布在所有间隔中，那么左边的空格要多于右边的空格，最后一行靠左对齐，每个单词间一个空格。

注意点：

- 单词的顺序不能发生改变
- 中间行也可能出现只有一个单词，这时要靠左对齐
- 每行要尽可能多的容纳单词

例子：

输入: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16

输出:

```
[
  "This    is    an",
  "example  of text",
  "justification.  "
]
```

## 解题思路

这道题比较繁琐，题目就一大段。采用双指针的方法来标记当前行的单词，如果加上下一个单词的长度和每个单词间至少一个空格时的总长度大于目标长度，说明此时的单词就是该行应该存放的。要分是否只有一个单词还是多个单词进行讨论，如果有多个单词，需要平均分配单词间的空格。现在可以知道总的空格数和单词间隔数，所以计算单词间的间隔比较简单，注意多余的空格要优先添加到左边的单词间隔中。不要忘记添加最后一行的单词。

## AC源码

```

class Solution(object):
    def fullJustify(self, words, maxWidth):
        """
        :type words: List[str]
        :type maxWidth: int
        :rtype: List[str]
        """
        start = end = 0
        result, curr_words_length = [], 0
        for i, word in enumerate(words):
            if len(word) + curr_words_length + end - start > ma
xWidth:
                if end - start == 1:
                    result.append(words[start] + ' ' * (maxWidth
- curr_words_length))
                else:
                    total_space = maxWidth - curr_words_length
                    space, extra = divmod(total_space, end - sta
rt - 1)

                    for j in range(extra):
                        words[start + j] += ' '
                    result.append((' ' * space).join(words[start
:end]))

                    curr_words_length = 0
                    start = end = i
                    end += 1
                    curr_words_length += len(word)
                    result.append(' '.join(words[start:end]) + ' ' * (maxWid
th - curr_words_length - (end - start - 1)))
                return result

if __name__ == "__main__":
    assert Solution().fullJustify(["This", "is", "an", "example"
, "of", "text", "justification."], 16) == [
        "This    is    an",
        "example  of text",

```

```
        "justification.  "  
    ]
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Sqrt(x)

## 原题

求一个数的平方根。

注意点：

- 结果返回整数，舍去小数，不是四舍五入

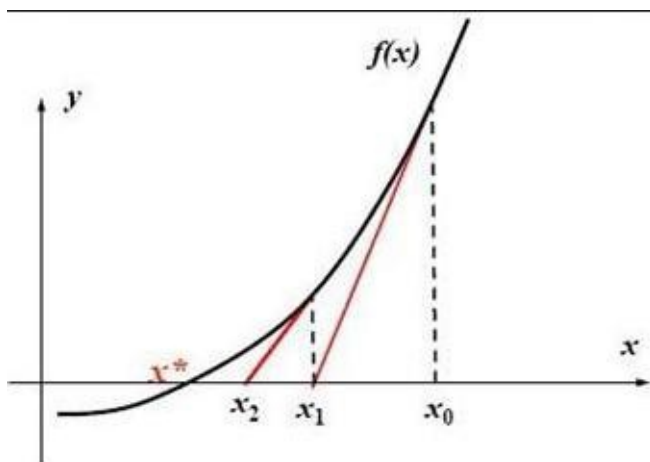
例子：

输入:  $x = 5$

输出: 2

## 解题思路

采用牛顿迭代法，通过逼近来求方程  $y = x^2 + a$  的解。简单介绍一下牛顿迭代法，如下图，求方程曲线与  $y$  轴的交点就是方程的解。随意取一个值  $x_0$ ，找出曲线在  $x_0$  处的切线，该切线与  $y$  轴的交点为  $x_1$ ，再求  $x_1$  处的切线的交点，可以看出来交点会不断的向目标值靠近，现在确定一个阈值就可以找出近似解了。由于平方根是正数，所以初始的取值应为一个正数。



注：图片来源于搜索引擎

## AC源码

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        result = 1.0
        while abs(result * result - x) > 0.1:
            result = (result + x / result) / 2
        return int(result)

if __name__ == "__main__":
    assert Solution().mySqrt(5) == 2
    assert Solution().mySqrt(0) == 0
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Climbing Stairs

---

### 原题

一共有 $n$ 级楼梯，每次能够爬一级或两级，共有多少种不同的爬法爬到顶端。

注意点：

- 无

例子：

输入:  $n = 6$

输出: 13

### 解题思路

典型的动态规划题，递推表达式为  $dp[i]=dp[i-1]+dp[i-2]$ ， $n$ 为1时只有一种方法， $n$ 为2时有两种方法。

### AC源码

```
class Solution(object):
    def climbStairs(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n <= 2:
            return n
        dp = [0 for __ in range(n)]
        dp[0] = 1
        dp[1] = 2
        for i in range(2, n):
            dp[i] = dp[i - 1] + dp[i - 2]
        return dp[n - 1]

if __name__ == "__main__":
    assert Solution().climbStairs(6) == 13
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之Simplify Path

---

## 原题

化简Unix系统下一个文件的绝对路径。

注意点：

- 根目录的上层目录还是根目录
- 可能有多个分隔符同时使用

例子：

输入: path = "/a/./b/../../c/"

输出: "/c"

## 解题思路

用栈来处理，碰到有效字符就压栈，遇到上层目录字符".."且栈不空时就弹出。为了最后连接字符串时头上有根目录，在栈底加一个空字符。

## AC源码

```
class Solution(object):
    def simplifyPath(self, path):
        """
        :type path: str
        :rtype: str
        """
        parts = path.split("/")
        result = ['']
        for part in parts:
            if part:
                if part not in ('.', '..'):
                    if len(result) == 0:
                        result.append('')
                    result.append(part)
                elif part == '..' and len(result) > 0:
                    result.pop()
        if len(result) < 2:
            return "/"
        else:
            return "/".join(result)

if __name__ == "__main__":
    assert Solution().simplifyPath("/a/./b/../../c/") == '/c'
    assert Solution().simplifyPath("/home/") == "/home"
    assert Solution().simplifyPath("/../..") == "/"
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Edit Distance

## 原题

求两个字符串之间的最短编辑距离，即原来的字符串至少要经过多少次操作才能够变成目标字符串，操作包括删除一个字符、插入一个字符、更新一个字符。

注意点：

- 无

例子：

输入: word1 = "heo", word2 = "hello"

输出: 2

## 解题思路

又是一道典型的动态规划。现在用  $dp[i][j]$  来表示字符串  $word1[:i]$  转化到  $word2[:j]$  的最小编辑距离，那么最后一次操作可能有三种情况：

- 在  $word1[:i-1]$  转化为  $word2[:j]$  的基础上再删除  $word1[i]$
- 在  $word1[:i]$  转化为  $word2[:j-1]$  的基础上再插入  $word2[j]$
- 在  $word1[:i-1]$  转化为  $word2[:j-1]$  的基础上将  $word1[i]$  更新为  $word2[j]$ （可能本来就相同）

所以有如下递推式：
$$dp[i][j] = \min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + \text{onemore})$$

## AC源码

```
class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        m = len(word1)
        n = len(word2)
        dp = [[0 for __ in range(m + 1)] for __ in range(n + 1)]
        for j in range(m + 1):
            dp[0][j] = j
        for i in range(n + 1):
            dp[i][0] = i
        for i in range(1, n + 1):
            for j in range(1, m + 1):
                onemore = 1 if word1[j - 1] != word2[i - 1] else
0
                dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1
, dp[i - 1][j - 1] + onemore)
            return dp[n][m]

if __name__ == "__main__":
    assert Solution().minDistance("", "a") == 1
    assert Solution().minDistance("faf", "efef") == 2
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Set Matrix Zeroes

## 原题

如果矩阵中存在0，那么把0所在的行和列都置为0。要求在所给的矩阵上完成操作。

注意点：

- 最好只申请常量级的额外空间

例子：

输入：

```
matrix =  
[[1, 0, 1, 1],  
 [1, 1, 0, 1],  
 [1, 1, 1, 0],  
 [1, 1, 1, 1]]
```

输出：

```
[[0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [1, 0, 0, 0]]
```

## 解题思路

一边遍历，一边将相应的行和列置为0是行不通的，会影响后面元素的遍历判断，所以要记录下哪些行和哪些列是要置为0的。为了节约空间，在原矩阵中借两条边，如果该行或者列要置为0，则把左边或者上边的相应位置置为0。如果左边和上边本来就有0，那么需要额外标记一下，最后把左边或者右边也全部置为0。

## AC源码

```

class Solution(object):
    def setZeroes(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: void Do not return anything, modify matrix in-place instead.
        """
        first_row = False
        first_col = False
        m = len(matrix)
        n = len(matrix[0])
        for i in range(m):
            if matrix[i][0] == 0:
                first_col = True
        for j in range(n):
            if matrix[0][j] == 0:
                first_row = True
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = matrix[0][j] = 0
        for i in range(1, m):
            for j in range(1, n):
                if matrix[0][j] == 0 or matrix[i][0] == 0:
                    matrix[i][j] = 0
        if first_row:
            for j in range(n):
                matrix[0][j] = 0
        if first_col:
            for i in range(m):
                matrix[i][0] = 0

if __name__ == "__main__":
    matrix = [[1, 0, 1, 1],
               [1, 1, 0, 1],
               [1, 1, 1, 0],
    ]

```

```
        [1, 1, 1, 1]]
Solution().setZeroes(matrix)
assert matrix == [[0, 0, 0, 0],
                  [0, 0, 0, 0],
                  [0, 0, 0, 0],
                  [1, 0, 0, 0]]
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Search a 2D Matrix

## 原题

在一个每行从左到右依次递增，且下一行第一个数字比上一行最后一个数字大的矩阵中，判断目标数字是否存在。

注意点：

- 无

例子：

输入：

```
matrix =  
[  
    [1,   3,   5,   7],  
    [10, 11, 16, 20],  
    [23, 30, 34, 50]  
]  
target = 3
```

输出: True

## 解题思路

把矩阵从左到右、从上到下连起来就是一个递增的数组，可以用二分搜索来查找。现在只要找出数组下标到矩阵的映射关系就可以了：`i -> [i // n][i % n]`，其中*i*是数组中的下标，*n*是矩阵的宽。

## AC源码



```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        m = len(matrix)
        n = len(matrix[0])
        l, h = 0, m * n - 1
        while l <= h:
            mid = l + (h - l) // 2
            if matrix[mid // n][mid % n] == target:
                return True
            elif matrix[mid // n][mid % n] < target:
                l = mid + 1
            else:
                h = mid - 1
        return False

if __name__ == "__main__":
    assert Solution().searchMatrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 5) == True
    assert Solution().searchMatrix([[1, 2], [3, 4]], 4) == True
    assert Solution().searchMatrix([[1]], 2) == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Sort Colors

## 原题

给出一个由红、白、蓝三种颜色组成的数组，把相同颜色的元素放到一起，并整体按照红、白、蓝的顺序。用0表示红色，1表示白色，2表示蓝色。这题也称为荷兰国旗问题。

注意点：

- 尽量只遍历一次

例子：

输入: nums = [1, 2, 1, 2, 0, 2, 1, 0, 2, 0, 0, 2]

输出: [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 2]

## 解题思路

如果只有两种颜色，那么很容易想到一前一后两个指针向中间遍历，颜色不对就交换位置。三种颜色仍然可以这么做，只不过要多一个指针，前后两个指针用来分隔已经排好的红色和蓝色，中间的指针来遍历元素：

- 如果是红色，那么和前指针交换，并两个一起向后移，前指针换过来的一定是白色的，因为中指针已经扫描过那些元素了
- 如果是白色，那么继续向后移
- 如果是蓝色，那么和后指针交换，后指针向前移，中指针不能后移，因为此时不确定换过来的元素是什么颜色

## AC源码

```
class Solution(object):
    def sortColors(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        left = mid = 0
        right = len(nums) - 1
        while mid <= right:
            if nums[mid] == 0:
                nums[mid], nums[left] = nums[left], nums[mid]
                left += 1
                mid += 1
            elif nums[mid] == 1:
                mid += 1
            else:
                nums[mid], nums[right] = nums[right], nums[mid]
                right -= 1

if __name__ == "__main__":
    l = [1, 2, 1, 2, 0, 2, 1, 0, 2, 0, 0, 2]
    Solution().sortColors(l)
    assert l == [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 2]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Minimum Window Substring

## 原题

给定两个字符串S和T，要求在O(n)的时间内找到包含T中所有字符的S的最短子字符串。

注意点：

- 如果不存在满足要求的子字符串，则返回""
- 如果存在多个子字符串满足要求，可以保证其中只有一个最短的

例子：

输入: s = "ADOBECODEBANC", t = "ABC"

输出: "BANC"

## 解题思路

通过前后指针来确定当前的子字符串，先不断移动后指针，直到子字符串中已经包含了所有T中的字符，尝试把前指针后移，并不断刷新最短长度和对应的起始位置，如果移动前指针后不再包含所有T中的字符，则继续移动后指针。交替移动前后指针，直到遍历完整个字符串S。

## AC源码

```
from collections import defaultdict

class Solution(object):
    def minWindow(self, s, t):
        """
```

```

:type s: str
:type t: str
:rtype: str
"""

MAX_INT = 2147483647
start = end = 0
char_need = defaultdict(int)    # the count of char needed by current window, negative means current window has it but not needs it

count_need = len(t)             # count of chars not in current window but in t
min_length = MAX_INT
min_start = 0
for i in t:
    # current window needs all char in t
    char_need[i] += 1
while end < len(s):
    if char_need[s[end]] > 0:
        count_need -= 1
    # current window contains s[end] now, so does not need it any more
    char_need[s[end]] -= 1
    end += 1
    while count_need == 0:
        if min_length > end - start:
            min_length = end - start
            min_start = start
        # current window does not contain s[start] anymore
        char_need[s[start]] += 1
        # when some count in char_need is positive, it means there is char in t but not current window
        if char_need[s[start]] > 0:
            count_need += 1
        start += 1
    return "" if min_length == MAX_INT else s[min_start:min_start + min_length]

if __name__ == "__main__":

```

```
assert Solution().minWindow("ADOBECODEBANC", "ABC") == "BANC"
```



欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Combinations

---

## 原题

求在1到n个数中挑选k个数的所有的组合类型。

注意点：

- 每个数字只能够用一遍
- 组合的排列没有顺序要求

例子：

输入:  $n = 4, k = 2$

输出:  $[[1, 4], [2, 4], [3, 4], [1, 3], [2, 3], [1, 2]]$

## 解题思路

采用递归的方式，在n个数中选k个，如果n大于k，那么可以分类讨论，如果选了n，那么就是在1到(n-1)中选(k-1)个，否则就是在1到(n-1)中选k个。递归终止的条件是k为1，这时候1到n都符合要求。

## AC源码

```
class Solution(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """
        if k == 1:
            return [[i + 1] for i in range(n)]
        result = []
        if n > k:
            result = [r + [n] for r in self.combine(n - 1, k - 1)] + self.combine(n - 1, k)
        else:
            result = [r + [n] for r in self.combine(n - 1, k - 1)]
        return result

if __name__ == "__main__":
    assert Solution().combine(4, 2) == [[1, 4], [2, 4], [3, 4], [1, 3], [2, 3], [1, 2]]
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Subsets

## 原题

给定一个由不同数字组成的集合，罗列出该集合的所有子集。

注意点：

- 子集要包括空集合和该集合自己
- 每个子集中的元素要按照不降序的顺序排列
- 结果集没有顺序要求

例子：

输入: `nums = [1,2,3]`

输出:

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

## 解题思路

与 [Combinations](#) 是一类题目，都可以用递归来解决。递归是倒过来解决问题，要求  $n$  的情况，就要先求  $n-1$ 。在这里尝试顺序的来解决，通过不断迭代的方法来求所有的子集。现在举个例子，集合 `[1]` 有 `[]`, `[1]` 两个子集，当向其中添加一个元素时，

[1,2]有[[],[1],[2],[1,2]]四个子集，可以看出，在新添加一个元素的时候，是在原来子集的基础上，添加原子集中所有元素加上新元素的总集合。为了每个子集中的元素都是不降序的，要先把所有元素都排序。

## AC源码

```
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = [[]]
        for num in sorted(nums):
            result += [item + [num] for item in result]
        return result

if __name__ == "__main__":
    assert Solution().subsets([1, 2, 3]) == [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Word Search

## 原题

在一个二维矩阵中，每个元素都是一个字母，要判断目标字符串能否由该矩阵中的元素连接而成。所谓连接就是从矩阵中的某一个元素开始，向前后左右不断前进，但不允许再次经过走过的元素。

注意点：

- 尝试遍历周围元素的时候小心越界

例子：

输入：

```
board =  
[  
    ['A', 'B', 'C', 'E'],  
    ['S', 'F', 'C', 'S'],  
    ['A', 'D', 'E', 'E']  
]  
word = "ABCCED"
```

输出: True

## 解题思路

采用深度优先遍历的方法，以每一个元素为起点进行查找。在此之前，可以做一个简单的前置判断，如果目标字符串中的某一个字母的数目比矩阵中所有该字母的数目还多，那么肯定是找不到目标字符串的。在进行深度遍历的时候，如果所有当前的遍历的位置越界或者与预期的值不等则返回，如果值相等，那么暂时把当前的值用特殊字符代替，防止继续遍历的时候又访问到访问过的点。

## AC源码

```

from collections import defaultdict

class Solution(object):
    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """
        if self._hasEnoughCharacters(board, word):
            m = len(board)
            n = len(board[0])
            for i in range(m):
                for j in range(n):
                    if self._exist(board, i, j, m, n, word):
                        return True
            return False
        else:
            return False

    def _exist(self, board, i, j, m, n, word):
        if len(word) == 0:
            return True
        if i < 0 or i >= m or j < 0 or j >= n or board[i][j] != word[0]:
            return False
        temp = board[i][j]
        board[i][j] = "."
        next_target = word[1:]
        next_result = self._exist(board, i - 1, j, m, n, next_target) \
            or self._exist(board, i + 1, j, m, n, next_target) \
            or self._exist(board, i, j - 1, m, n, next_target) \
            or self._exist(board, i, j + 1, m, n, next_target)

```

```
_target)
    board[i][j] = temp
    return next_result

def _hasEnoughCharacters(self, board, word):
    character_counts = defaultdict(int)
    for ch in word:
        character_counts[ch] += 1
    return all(sum(map(lambda line: line.count(ch), board))
               >= count for ch, count in character_counts.items())

if __name__ == "__main__":
    assert Solution().exist([
        ['A', 'B', 'C', 'E'],
        ['S', 'F', 'C', 'S'],
        ['A', 'D', 'E', 'E']
    ], "ABCCED") == True
    assert Solution().exist([
        ['A', 'B', 'C', 'E'],
        ['S', 'F', 'C', 'S'],
        ['A', 'D', 'E', 'E']
    ], "SEE") == True
    assert Solution().exist([
        ['A', 'B', 'C', 'E'],
        ['S', 'F', 'C', 'S'],
        ['A', 'D', 'E', 'E']
    ], "ABCB") == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Remove Duplicates from Sorted Array II

## 原题

在 [Remove Duplicates from Sorted Array](#)（从一个有序的数组中去除重复的数字，返回处理后的数组长度）的基础上，可以使每个数字最多重复一次，也就是说如果某一个数字的个数大于等于2个，结果中应保留2个该数字。

注意点：

- 只能用常量的额外空间
- 将要保留的数字移到数组前部，剩余的部分不需要处理

例子：

输入: nums = [1,1,1,2,2,3]

输出: 5 ([1,1,2,2,3])

## 解题思路

首先记住原数组是有序的，再看一下以下几种情况：

- [1,1]
- [1,1,2]
- [1,1,2,2]
- [1,1,2,2,3]

在每一次插入过程中，其实只要把要插入的元素和倒数第二个元素进行比较，如果相同，就忽略，因为倒数第一个数是夹在它们中间的，如果它们相等，那么就会有三个数相等；如果不同，就可以插入，因为在这样的情况下，最多只有倒数第二、倒数第一两个数相等。

## AC源码

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        count = 0
        for i in range(len(nums)):
            if count < 2 or nums[count - 2] != nums[i]:
                nums[count] = nums[i]
                count += 1
        return count

if __name__ == "__main__":
    l = [1, 1, 1, 2, 2, 3]
    r = Solution().removeDuplicates(l)
    assert l == [1, 1, 2, 2, 3, 3]
    assert r == 5
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

## LeetCode解题之Search in Rotated Sorted Array II

---

### 原题

把一个不降序的数组进行旋转，如[0,1,1,1,2,3,4,5]旋转3位成为[3,4,5,0,1,1,1,2]。  
在这样的数组中判断目标数字是否存在。

注意点：

- 不知道数组旋转了多少位

例子：

输入: nums = [4, 5, 5, 6, 7, 0, 1, 2], target = 4

输出: True

### 解题思路

当时写 [Search in Rotated Sorted Array](#) 的时候已经考虑到数字重复的问题了，所以直接把原来的代码修改一下，返回下标改为返回布尔值。

### AC源码



```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return True
            if nums[mid] > target:
                if nums[left] <= target or nums[mid] < nums[left]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[left] > target or nums[mid] >= nums[left]:
                    left = mid + 1
                else:
                    right = mid - 1
        return False

if __name__ == "__main__":
    assert Solution().search([4, 5, 5, 6, 7, 0, 1, 2], 4) == True

    assert Solution().search([4, 5, 6, 7, 7, 7, 7, 0, 1, 2], 7) == True
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode解题之Remove Duplicates from Sorted List II

## 原题

把一个有序链表中所有重复的数字全部删光，删除后不再有原先重复的那些数字。

注意点：

- 头节点也可能是重复的对象

例子：

输入: 1->1->2->3->3

输出: 2

## 解题思路

先按照 [Remove Duplicates from Sorted List](#) 的方法把重复的数字减至一个，接下来要考虑怎样可以把这一个节点也删除，那么只要在前一个节点对当前节点去重后，再把前一个节点的指针指向当前节点的后一个节点。因为要通过当前节点的前一个节点来操作，所以加了一个假的头节点。而如果节点没有重复的话不需要跳过当前节点，所以要一个额外的变量来记录当前节点是否有重复节点。

## AC源码

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

    def my_print(self):
```

```
        print(self.val)
        if self.next:
            print(self.next.val)

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(-1)
        dummy.next = head
        curr = dummy
        is_repeat = False
        while curr.next:
            while curr.next.next and curr.next.val == curr.next.
next.val:
                curr.next = curr.next.next
                is_repeat = True
            if is_repeat:
                curr.next = curr.next.next
                is_repeat = False
            else:
                curr = curr.next
        return dummy.next

if __name__ == "__main__":
    n1 = ListNode(1)
    n2 = ListNode(1)
    n3 = ListNode(2)
    n1.next = n2
    n2.next = n3
    r = Solution().deleteDuplicates(n1)
    r.my_print()
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Remove Duplicates from Sorted List

---

### 原题

删除一个有序链表中重复的元素，使得每个元素只出现一次。

注意点：

- 无

例子：

输入: 1->1->2->3->3

输出: 1->2->3

### 解题思路

顺序遍历所有节点，如果当前节点有后一个节点，且它们的值相等，那么当前节点指向后一个节点的下一个节点，这样就可以去掉重复的节点。

### AC源码

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

    def my_print(self):
        print(self.val)
        if self.next:
            print(self.next.val)

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        curr = head
        while curr:
            while curr.next and curr.val == curr.next.val:
                curr.next = curr.next.next
            curr = curr.next
        return head

if __name__ == "__main__":
    n1 = ListNode(1)
    n2 = ListNode(1)
    n3 = ListNode(2)
    n1.next = n2
    n2.next = n3
    r = Solution().deleteDuplicates(n1)
    r.my_print()
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

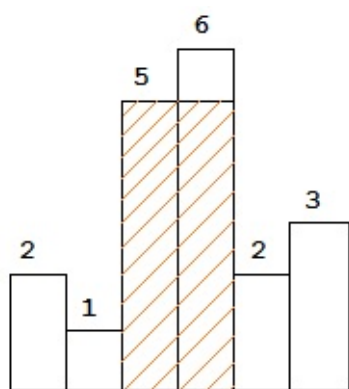




## LeetCode 解题之 Largest Rectangle in Histogram

### 原题

给定一个柱状图，求它能包含的最大的矩形的面积。如下图中阴影部分就是要求的矩形。



注意点：

- 所有的柱的宽度都为1

例子：

输入: heights = [2,1,5,6,2,3]

输出: 10

### 解题思路

这道题卡了很久，一直没想到好的解决方案。查了一些资料，比较优雅的方法是通过栈来解决。

首先明确最终矩形的高度一定是某一个柱的高度，以上面的例子为例，可能是2,1,5,6,2,3中的一个，如果像这样把每个柱穷举出来，并尝试向两边扩展来得到最大面积，时间复杂度就太大。仔细想一想，为什么要尝试向两边扩展，如例子中的

第二个2（以它为高的话最多可以涵盖4个柱），因为它的左右两边相邻的柱都有比它高的，也就是说柱的高度不是有序的，如果我们能得到一个有序递增的排列，那么就只要像右扩展，而不要向左扩展了，因为左边的柱比它自己低。

依次遍历柱状结构，如果是递增的则压栈，如果不是则把比它高的柱依次弹出（只弹出比当前柱高的可以保证把当前柱压栈后，栈中的柱依旧是依次递增的），并计算以该柱为高的矩形的面积。计算面积时，宽度应该是栈顶元素到遍历到元素之间的宽，如当弹出第二个2（2后面没有比它小的元素，为了使该元素能顺利弹出，在原柱状图末尾加一个0）时，栈顶元素是1，这样就能方便计算出宽度为4。还有一个问题是弹出1时栈中没有元素，无法计算宽度，所以在初始化时要在栈底加一个-1来应对所有元素都出栈的情况。

## AC源码

```
class Solution(object):
    def largestRectangleArea(self, heights):
        """
        :type heights: List[int]
        :rtype: int
        """
        heights.append(0)
        stack = [-1]
        result = 0
        for i in range(len(heights)):
            while heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]
                w = i - stack[-1] - 1
                result = max(result, h * w)
            stack.append(i)
        heights.pop()
        return result

if __name__ == "__main__":
    assert Solution().largestRectangleArea([2, 1, 5, 6, 2, 3]) == 10
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Maximal Rectangle

## 原题

一个矩阵仅包含1和0，找出其中面积最大的只含有1的矩形，并返回它的面积。

注意点：

- 矩阵中的元素类型是字符串

例子：

输入：

```
matrix =  
[['1', '1', '0', '1', '0', '1'],  
 ['0', '1', '0', '0', '1', '1'],  
 ['1', '1', '1', '1', '0', '1'],  
 ['1', '1', '1', '1', '0', '1']]
```

输出: 8

## 解题思路

这道题和 [Largest Rectangle in Histogram](#) 的关系非常密切。如果把1看做柱状的实体，那么这就是一个中间有空缺的柱状图。依次遍历每一行，把每一行当做柱状图的底边，就能将题目转化为Largest Rectangle in Histogram来解决。如以第二行为底，则是这样一个柱状图[1,2,0,0,1,1]，容易发现规律：遍历下一行时，如果是1，则在原来的高度上加一，否则将高度置为0。

## AC源码

```

class Solution(object):
    def maximalRectangle(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        if not matrix or not matrix[0]:
            return 0
        n = len(matrix[0])
        heights = [0 for __ in range(n + 1)]
        result = 0
        for row in matrix:
            for i in range(n):
                heights[i] = heights[i] + 1 if row[i] == '1' else 0

            stack = [-1]
            for i in range(n + 1):
                while heights[i] < heights[stack[-1]]:
                    h = heights[stack.pop()]
                    w = i - stack[-1] - 1
                    result = max(result, h * w)
                stack.append(i)
        return result

if __name__ == "__main__":
    assert Solution().maximalRectangle([['1', '1', '0', '1', '0'],
                                         ['0', '1', '0', '0', '1'],
                                         ['1', '1', '1', '1', '0'],
                                         ['1', '1', '1', '1', '0'],
                                         ['1', '1', '1', '1', '0']],
                                         == 8

```

欢迎查看我的 [Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



# LeetCode 解题之 Partition List

## 原题

给定一个链表以及一个目标值，把小于该目标值的所有节点都移至链表的前端，大于等于目标值的节点移至链表的尾端，同时要保持这两部分在原先链表中的相对位置。

注意点：

- 链表的排序一般通过重新连接指针来完成

例子：

输入: head = 1->4->3->2->5->2, x = 3

输出: 1->2->2->4->3->5

## 解题思路

看成有一串珠子，有红和蓝两种颜色，现在要把红色和蓝色分别集中到一起。可以遍历每个珠子，如果是蓝色就串在一条线上，红色的串在另一条线上，最后把两条线连起来就可以了。注意，在比较大的那串数中，最后的指针要置为None，因为那是排序后的最后一个节点。

## AC源码

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

    def to_list(self):
        return [self.val] + self.next.to_list() if self.next else None
```

```
[self.val]

class Solution(object):
    def partition(self, head, x):
        """
        :type head: ListNode
        :type x: int
        :rtype: ListNode
        """
        dummy = ListNode(-1)
        dummy.next = head
        small_dummy = ListNode(-1)
        large_dummy = ListNode(-1)

        prev = dummy
        small_prev = small_dummy
        large_prev = large_dummy
        while prev.next:
            curr = prev.next
            if curr.val < x:
                small_prev.next = curr
                small_prev = small_prev.next
            else:
                large_prev.next = curr
                large_prev = large_prev.next
            prev = prev.next
        large_prev.next = None
        small_prev.next = large_dummy.next
        return small_dummy.next

if __name__ == "__main__":
    n1 = ListNode(1)
    n2 = ListNode(4)
    n3 = ListNode(3)
    n4 = ListNode(2)
    n5 = ListNode(5)
    n6 = ListNode(2)
    n1.next = n2
```



```
n2.next = n3
n3.next = n4
n4.next = n5
n5.next = n6
r = Solution().partition(n1, 3)
assert r.to_list() == [1, 2, 2, 4, 3, 5]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Scramble String

## 原题

一个字符串可以拆分成两个都不为空的子字符串，而子字符串（长度大于等于二）也可以不断这样拆分下去，现在可以任意交换拆分出来两部分的位置来改变字符串中字符的顺序。判断两个字符串能否通过这种方式相互转换。

注：这道题比较难用语言描述，可以参见原题中的图例

原题请点 [这里](#)

注意点：

- 给的两个字符串的长度相等

例子：

输入: s1 = "rgtae", s2 = "great"

输出: True ("rgtae" -> "grtae" -> "greta" -> "great")

## 解题思路

对三维动态规划还不是很熟练，偷懒用了最简单的递归方式，以后会补上动态规划解法。要判断两个字符串S和T能否转化，先把它们各自分为两部分，如果S的前半部分和T的前半部分能转换，它们的后半部分也能转换，说明它们就能转换；但也有可能S的前半部分和后半部分是在最后一交换中转换回来的，也就是S的前半部分和T的后半部分能够转换，而T的前半部分和S的后半部分能够转换同样能够达到目的。还可以在我代码的基础上再做一些优化，如提前判断两个要转化的字符串中各字符的数目是否相等来进行剪枝，来减少没有用的递归。经过剪枝的递归算法的运行速度还是很快的。

## AC源码

```
from collections import defaultdict

class Solution(object):
    def isScramble(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: bool
        """
        if s1 == s2:
            return True
        count1 = defaultdict(int)
        count2 = defaultdict(int)
        for e1, e2 in zip(s1, s2):
            count1[e1] += 1
            count2[e2] += 1
        if count1 != count2:
            return False
        for i in range(1, len(s1)):
            if self.isScramble(s1[:i], s2[:i]) and self.isScramble(s1[i:], s2[i:]) \
                or self.isScramble(s1[:i], s2[-i:]) and self.isScramble(s1[i:], s2[:len(s2) - i]):
                return True
        return False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Merge Sorted Array

---

## 原题

将两个有序数组合并成为一个。

注意点：

- 第一个数组有充足的空间来存放第二个数组中的元素
- 第一个数组的有效长度为m，第二个的有效长度为n
- 在原数组上修改，没有返回值

例子：

输入: `nums1 = [1, 1, 2, 2, 4, 0, 0, 0, 0]`, `m = 5`, `nums2 = [0, 0, 2, 3]`, `n = 4`

输出: 无 (`nums1`变为`[0, 0, 1, 1, 2, 2, 2, 3, 4]`)

## 解题思路

知道两个数组原先的长度，就可以知道合并后的长度，倒叙遍历两个数组，大的数优先放到合并后的数组对应下标处。如果第一个数组先遍历完，那应该把第二个数组剩下的元素复制过来；如果第二个先遍历玩，就不用变化了，因为第一个数组剩余的元素已经在目标位置。

## AC源码

```
class Solution(object):
    def merge(self, nums1, m, nums2, n):
        """
        :type nums1: List[int]
        :type m: int
        :type nums2: List[int]
        :type n: int
        :rtype: void Do not return anything, modify nums1 in-place instead.
        """
        index = m + n - 1
        m -= 1
        n -= 1
        while m >= 0 and n >= 0:
            if nums1[m] > nums2[n]:
                nums1[index] = nums1[m]
                m -= 1
            else:
                nums1[index] = nums2[n]
                n -= 1
            index -= 1
        if m < 0:
            nums1[:n + 1] = nums2[:n + 1]

if __name__ == "__main__":
    num1 = [1, 1, 2, 2, 4, 0, 0, 0, 0]
    num2 = [0, 0, 2, 3]
    Solution().merge(num1, 5, num2, 4)
    assert num1 == [0, 0, 1, 1, 2, 2, 2, 3, 4]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Gray Code

## 原题

格雷码表示在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同。现给定二进制码的位数，要求打印出格雷码序列。

注意点：

- 格雷码序列有多种可能，可以先改变低位或高位

例子：

输入:  $n = 2$

输出: `[0,1,3,2]`

```
00 - 0
01 - 1
11 - 3
10 - 2
```

## 解题思路

根据维基百科上的关于 [格雷码和二进制数的转换关系](#) 实现的代码。

## AC 源码

```
class Solution(object):
class Solution(object):
    def grayCode(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        result = [(i >> 1) ^ i for i in range(pow(2, n))]
        return result

if __name__ == "__main__":
    assert Solution().grayCode(2) == [0, 1, 3, 2]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Subsets II

## 原题

罗列出一个包含重复数字的集合的所有的子集。

注意点：

- 子集中的元素需要按照不降序排列
- 结果集中不能重复

例子：

输入: `nums = [1,2,2]`

输出：

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

## 解题思路

在 [Subsets](#) 迭代版本的基础上思考。在迭代重复元素的时候会生成重复的结果，那么在迭代重复元素时要特殊处理一下。就拿`[1,2,2]`来说，在迭代完1之后结果集为`[], [1]`，迭代第一个2后，`[], [1], [2], [1,2]`，接下来就要迭代重复的元素2了，此时如果遍历在迭代第一个2之前就存在的结果集元素（`[], [1]`）时，就会产生重复，我们只能在上一轮迭代产生的新的结果中继续添加。所以要一个额外的变量来表示在结果集中的哪个位置开始遍历。



## AC源码

```
class Solution(object):
    def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = [[]]
        nums.sort()
        temp_size = 0
        for i in range(len(nums)):
            start = temp_size if i >= 1 and nums[i] == nums[i - 1]
            else 0
            temp_size = len(result)
            for j in range(start, temp_size):
                result.append(result[j] + [nums[i]])
        return result

if __name__ == "__main__":
    assert Solution().subsetsWithDup([1, 2, 2]) == [[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Decode Ways

## 原题

现在有如下的字母与数字的对应关系：1-A, 2-B, ...26-Z。给定一个由数字组成的字符串，判断按照上面的映射可以转换成多少种不同的字符串。

注意点：

- 如果字符不能正常转换，如"70"，那么返回0

例子：

输入: s = "12"

输出: 2 (包括"AB"(1 2)和"L"(12))

## 解题思路

第一感觉就是动态规划，先写了一个从前往后的版本，要复杂的分类要论，代码很乱，后来发现从后开始遍历更加容易。来看一下递推式，假设原先的字符串为"y231"，现在在它之前加一个数字得到"xy231"，如果x不为0，此时，如果"xy"不在1-26之间，那么原先能转换的种类不变，只是在每个字符串之前增加一个x转换后的字母；如果"xy"在1-26之间，那么除了原先每个字符串之前增加x转换后的字母，还可能是在"231"转化之后的字符串前增加"xy"转化的字母。那如果x等于0呢，此时是一个非法字符串，让它默认为0。但不能跳出循环，因为在前面继续增加数字可能将字符串变为合法的。

## AC源码

```
class Solution(object):
    def numDecodings(self, s):
        """
        :type s: str
        :rtype: int
        """
        length = len(s)
        if length == 0:
            return 0
        dp = [0 for __ in range(length + 1)]
        dp[length] = 1
        dp[length - 1] = 1 if s[length - 1] != '0' else 0
        for i in range(length - 2, -1, -1):
            if s[i] != '0':
                dp[i] = dp[i + 1] + dp[i + 2] if int(s[i:i + 2])
                <= 26 else dp[i + 1]
            return dp[0]

if __name__ == "__main__":
    assert Solution().numDecodings("110") == 1
    assert Solution().numDecodings("40") == 0
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Reverse Linked List II

## 原题

在只遍历一遍且不申请额外空间的情况下将一个链表的第m到n个元素进行翻转。

注意点：

- m和n满足如下条件： $1 \leq m \leq n \leq \text{链表长度}$

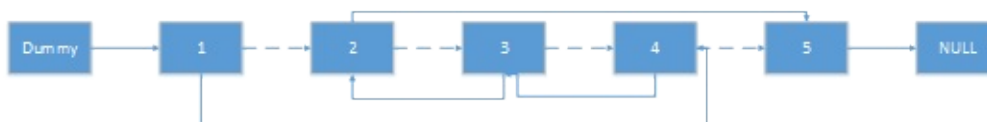
例子：

输入: 1->2->3->4->5->NULL, m = 2, n = 4

输出: 1->4->3->2->5->NULL

## 解题思路

通过 [Reverse Linked List](#) 已经可以实现链表的翻转。看下图，把要翻转的一段先进行翻转，再把它和前后的链表接起来。因为可能要把第一个节点也进行翻转，为了一致性增加一个假的头节点。



## AC源码

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

    def to_list(self):
        return [self.val] + self.next.to_list() if self.next else []
```

```
[self.val]
```

```
class Solution(object):
    def reverseBetween(self, head, m, n):
        """
        :type head: ListNode
        :type m: int
        :type n: int
        :rtype: ListNode
        """
        dummy = ListNode(-1)
        dummy.next = head
        node = dummy
        for __ in range(m - 1):
            node = node.next
        prev = node.next
        curr = prev.next
        for __ in range(n - m):
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        node.next.next = curr
        node.next = prev
        return dummy.next

if __name__ == "__main__":
    n1 = ListNode(1)
    n2 = ListNode(2)
    n3 = ListNode(3)
    n4 = ListNode(4)
    n5 = ListNode(5)
    n1.next = n2
    n2.next = n3
    n3.next = n4
    n4.next = n5
    r = Solution().reverseBetween(n1, 2, 4)
    assert r.to_list() == [1, 4, 3, 2, 5]
```



欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Restore IP Addresses

---

## 原题

找出一个由纯数字组成的序列能够构成的不同的IP地址。

注意点：

- 每个IP段的范围是0-255
- 要用整个序列，而不是它的子集

例子：

输入: s = "25525511135"

输出: ["255.255.11.135", "255.255.111.35"]

## 解题思路

每次把1到3个数字当作一个IP段，多个数字时要注意首位不能为0，因为

01.0.0.0 这样的IP是不符合规范的，此外三个数字时还不能超过255。当递归的序列为空，且此时正好集齐四个IP段，则得到一个正确答案。在递归的序列为空或者IP段数目达到4时都应该终止递归。

## AC源码

```
class Solution(object):
    def restoreIpAddresses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        result = []
        self._restoreIpAddresses(0, s, [], result)
        return result

    def _restoreIpAddresses(self, length, s, ips, result):
        if not s:
            if length == 4:
                result.append('.'.join(ips))
            return
        elif length == 4:
            return
        self._restoreIpAddresses(length + 1, s[1:], ips + [s[:1]], result)
        if s[0] != '0':
            if len(s) >= 2:
                self._restoreIpAddresses(length + 1, s[2:], ips + [s[:2]], result)
            if len(s) >= 3 and int(s[:3]) <= 255:
                self._restoreIpAddresses(length + 1, s[3:], ips + [s[:3]], result)

if __name__ == "__main__":
    assert Solution().restoreIpAddresses("25525511135") == ['255.255.11.135', '255.255.111.35']
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Binary Tree Inorder Traversal

## 原题

不用递归来实现树的中序遍历。

注意点：

- 无

例子：

输入: {1,#,2,3}

```
  1
   \
    2
   /
  3
```

输出: [1,3,2]

## 解题思路

通过栈来实现，从根节点开始，不断寻找左节点，并把这些节点依次压入栈内，只有在该节点没有左节点或者它的左子树都已经遍历完成后，它才会从栈内弹出，这时候访问该节点，并它的右节点当做新的根节点一样不断遍历。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result = []
        stack = []
        p = root
        while p or stack:
            # Save the nodes which have left child
            while p:
                stack.append(p)
                p = p.left
            if stack:
                p = stack.pop()
                # Visit the middle node
                result.append(p.val)
                # Visit the right subtree
                p = p.right
        return result

if __name__ == "__main__":
    n1 = TreeNode(1)
    n2 = TreeNode(2)
    n3 = TreeNode(3)
    n1.right = n2
    n2.left = n3
    assert Solution().inorderTraversal(n1) == [1, 3, 2]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Unique Binary Search Trees II

## 原题

给定1到n这n个数，用它们能够构成多少种形状不同的二叉搜索树。将所有的二叉搜索树罗列出来。

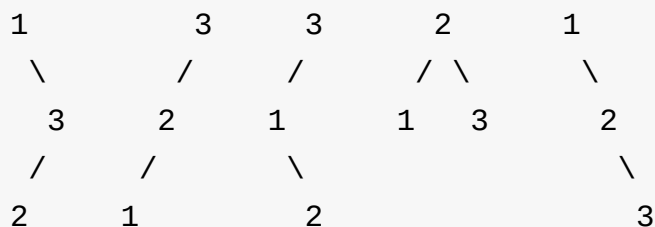
注意点：

- 这n个数都要是二叉搜索树的节点，不能只取部分

例子：

输入:  $n = 3$

输出：



## 解题思路

[Unique Binary Search Trees](#) 只要求不同二叉搜索树的数目，现在要求把所有的树的结构都打印出来。所以在递归的时候要把树拼装出来，而不是仅仅计算数目。而且同一个子树可能会在不同的二叉搜索树中多次出现，为了不重复计算，就用一个map来缓存构造过的树形结构。

## AC源码

```

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def generateTrees(self, n):
        """
        :type n: int
        :rtype: List[TreeNode]
        """
        if n == 0:
            return []
        self.cache = {}
        return self._generateTrees(1, n)

    def _generateTrees(self, start, end):
        if (start, end) not in self.cache:
            roots = []
            for root in range(start, end + 1):
                for left in self._generateTrees(start, root - 1):
                    for right in self._generateTrees(root + 1, end):
                        node = TreeNode(root)
                        node.left = left
                        node.right = right
                        roots.append(node)
            self.cache[(start, end)] = roots
        return self.cache[(start, end)] or [None]

if __name__ == "__main__":
    None

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Unique Binary Search Trees

## 原题

给定1到n这n个数，用它们能够构成多少种形状不同的二叉搜索树。

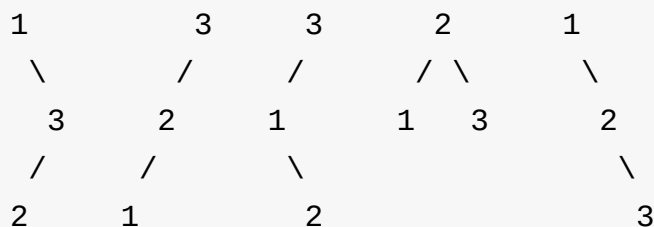
注意点：

- 这n个数都要是二叉搜索树的节点，不能只取部分

例子：

输入:  $n = 3$

输出: 5



## 解题思路

首先明确n个不等的数它们能构成的二叉搜索树的种类都是相等的。而且1到n都可以作为二叉搜索树的根节点，当k是根节点时，它的左边有k-1个不等的数，它的右边有n-k个不等的数。以k为根节点的二叉搜索树的种类就是左右可能的种类的乘积。用递推式表示就是  $h(n) = h(0)*h(n-1) + h(1)*h(n-2) + \dots + h(n-1)h(0)$  (其中 $n \geq 2$ )，其中 $h(0)=h(1)=1$ ，因为0个或者1个数能组成的形状都只有一个。从1到n依次算出h(x)的值即可。此外这其实就是一个卡特兰数，可以直接用数学公式计算，不过上面的方法更加直观一些。

## AC源码

```
class Solution(object):
    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        dp = [1 for __ in range(n + 1)]
        for i in range(2, n + 1):
            s = 0
            for j in range(i):
                s += dp[j] * dp[i - 1 - j]
            dp[i] = s
        return dp[-1]

if __name__ == "__main__":
    assert Solution().numTrees(5) == 42
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



# LeetCode解题之Interleaving String

## 原题

输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交替而成且不改变s1和s2中各个字符原有的相对顺序。

注意点：

- 无

例子：

输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbcbcbac"

输出: True

## 解题思路

典型的二维动态规划题目， $dp[i][j]$ 表示s1[:i+1]和s2[:j+1]能否交替组成s3[:i+j+1]，两个空字符串可以组成空字符串，所以 $dp[0][0]$ 为True。边界的情况是一个字符串为空，另一个字符串的头部是否与目标字符串的头像相同。而在一般情况下，只有当以下两种情况之一成立时 $dp[i][j]$ 为True：

1.  $s1[i] == s3[i+j]$ ，而且 $dp[i-1][j]$ 为True
2.  $s2[j] == s3[i+j]$ ，而且 $dp[i][j-1]$ 为True

递推关系式是： $dp[i + 1][j + 1] = (dp[j + 1][i] \text{ and } s1[i] == s3[i + j + 1]) \text{ or } (dp[j][i + 1] \text{ and } s2[j] == s3[i + j + 1])$

考虑到不同纬度间的数据不干扰，所有可以把二维dp降为一维。

## AC源码

```
class Solution(object):
    def isInterleave(self, s1, s2, s3):
        """
        :type s1: str
        :type s2: str
        :type s3: str
        :rtype: bool
        """
        m = len(s1)
        n = len(s2)
        l = len(s3)
        if m + n != l:
            return False
        dp = [True for __ in range(m + 1)]
        for i in range(m):
            dp[i + 1] = dp[i] and s1[i] == s3[i]
        for j in range(n):
            dp[0] = dp[0] and s2[j] == s3[j]
            for i in range(m):
                dp[i + 1] = (dp[i] and s1[i] == s3[i + j + 1]) or
                    (dp[i + 1] and s2[j] == s3[i + j + 1])
        return dp[m]

if __name__ == "__main__":
    assert Solution().isInterleave("aabcc", "dbbca", "aadbcbcbcac")
    == True
    assert Solution().isInterleave("aabcc", "dbbca", "aadbabbaccc")
    == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Validate Binary Search Tree

## 原题

判断一棵二叉搜索树是否有效。有效是指每个节点的值大于左节点，小于右节点（如果有对应节点的话），且它的左节点和右节点也满足这种条件。

注意点：

- 无

例子：

输入：

```
    2
   /\
  1  3
```

输出: True

## 解题思路

在 [Binary Tree Inorder Traversal](#) 的基础上进行了修改。在树的中序遍历中，节点的顺序是左节点、根节点、右节点。这就说明一棵二叉搜索树要符合要求时，它的中序遍历序列一定是递增的。如果在中序遍历中出现前面的节点大于后面的节点，则说明不符合要求。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def isValidBST(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        stack = []
        curr = root
        prev = None
        while curr or stack:
            while curr:
                stack.append(curr)
                curr = curr.left

            if stack:
                curr = stack.pop()
                if prev and curr.val <= prev.val:
                    return False
                prev = curr
                curr = curr.right
        return True

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Recover Binary Search Tree

## 原题

一棵二叉搜索树中的两个节点交换了位置，找出并调整。

注意点：

- 最好只用常量的空间

例子：

输入：

```
    3
   /\
  1  2
```

输出：

```
    2
   /\
  1  3
```

## 解题思路

二叉搜索树的中序遍历可以使它的节点排列成一个递增的序列，那就可以把问题简化成在一个递增序列中有两个元素交换了位置。而前面的元素大于后面的元素表示顺序出错，如果整个序列就一处这样的问题，那只要交换这两个元素，如果有两处，在只有两个元素顺序有问题的前提下，应该交换第一次错位的前元素（大的元素应该往后放）与第二次错位的后元素（小元素往前放）。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def __init__(self):
        self.node1 = None
        self.node2 = None
        self.pre = None

    def recoverTree(self, root):
        """
        :type root: TreeNode
        :rtype: void Do not return anything, modify root in-place instead.
        """
        self.__scan(root)
        self.node1.val, self.node2.val = self.node2.val, self.node1.val

    def __scan(self, root):
        if root is None:
            return
        self.__scan(root.left)
        if self.pre is not None:
            if root.val < self.pre.val:
                if self.node1 is None:
                    self.node1 = self.pre
                    self.node2 = root
                else:
                    self.node2 = root
        self.pre = root
        self.__scan(root.right)
```

```
if __name__ == "__main__":  
    None
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode 解题之 Same Tree

### 原题

判断两棵二叉树是否相等。两棵二叉树仅在它们的形状相同且每个节点的值相等时才判为相等。

注意点：

- 无

例子：

输入：

```
      2      2
     / \    / \
    p = /  \ q = /  \
         1  3   1  3
```

输出: True

### 解题思路

树相关的问题一般用递归的方法最好理解。如果两棵树对应的节点都为空，则相等；如果值相等，那么就分别判断它们的左右子树是否相等，否则认为两棵树不相等。

### AC 源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def isSameTree(self, p, q):
        """
        :type p: TreeNode
        :type q: TreeNode
        :rtype: bool
        """
        if not q and not p:
            return True
        elif not p or not q:
            return False
        elif p.val != q.val:
            return False
        else:
            return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Symmetric Tree

## 原题

判断一棵树是否是镜面对称的。最好同时提供递归和迭代的解法。

注意点：

- 无

例子：

输入：

```
      1
     /\
    2  2
   /\ /\
  3 4 4 3
```

输出: True

输入：

```
      1
     /\
    2  2
     \  \
     3   3
```

输出: False

## 解题思路

看一棵二叉树是否对称，就要首先看根节点的左右节点A和B是否有相同的值，如果A和B的值相等，那么要继续判断A的左节点和B的右节点以及A的右节点和B的左节点是否对称，通过这样的方式来递归得到结果。用迭代方法来解决的话，就把要判断是否对称的点按序（注意需要判断哪些节点是否相等）放到两个栈中，不断出栈和压栈来判断。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # Solve it recursively
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if not root:
            return True
        return self._isSymmetric(root.left, root.right)

    def _isSymmetric(self, left, right):
        if not left and not right:
            return True
        if not left or not right:
            return False
        if left.val != right.val:
            return False
        return self._isSymmetric(left.left, right.right) and self._isSymmetric(left.right, right.left)

    # Solve it iteratively
```

```
def isSymmetric_iterate(self, root):  
    """  
    :type root: TreeNode  
    :rtype: bool  
    """  
    if not root:  
        return True  
    stack1, stack2 = [], []  
    stack1.append(root.left)  
    stack2.append(root.right)  
    while stack1 and stack2:  
        size1 = len(stack1)  
        size2 = len(stack2)  
        if size1 != size2:  
            return False  
        for __ in range(size1):  
            curr1, curr2 = stack1.pop(), stack2.pop()  
            if not curr1 and not curr2:  
                continue  
            if not curr1 or not curr2:  
                return False  
            if curr1.val != curr2.val:  
                return False  
            stack1.append(curr1.left)  
            stack1.append(curr1.right)  
            stack2.append(curr2.right)  
            stack2.append(curr2.left)  
    return not stack1 and not stack2  
  
if __name__ == "__main__":  
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Binary Tree Level Order Traversal

## 原题

实现树的广度优先遍历，每一层上的数据按照从左到右的顺序排列。

注意点：

- 无

例子：

输入：

```
    3
   / \
  9  20
   / \
  15  7
```

输出：

```
[
  [3],
  [9,20],
  [15,7]
]
```

## 解题思路

将树每一层的节点存在一个列表中，遍历列表中的元素，如果该节点有左右节点的话，就把它加入一个临时列表，这样当遍历结束时，下一层的节点也按照顺序存储好了，不断循环直到下一层的列表为空。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def levelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        result = []
        if not root:
            return result
        curr_level = [root]
        while curr_level:
            level_result = []
            next_level = []
            for temp in curr_level:
                level_result.append(temp.val)
                if temp.left:
                    next_level.append(temp.left)
                if temp.right:
                    next_level.append(temp.right)
            result.append(level_result)
            curr_level = next_level
        return result

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



# LeetCode 解题之 Binary Tree Zigzag Level Order Traversal

## 原题

实现树的弯曲遍历，即奇数层从左到右遍历，偶数层从右到左遍历。

注意点：

- 无

例子：

输入：

```
    3
   / \
  9  20
   / \
  15  7
```

输出：

```
[
  [3],
  [20,9],
  [15,7]
]
```

## 解题思路

这道题跟 [Binary Tree Level Order Traversal](#) 非常相似，本质上也是树的广度优先遍历，只是在遍历的时候每一层的遍历顺序不同。那么我们只要一个变量来区分当前层是从前往后还是从后往前遍历。偷了下懒，当要反过来遍历节点时直接把原有

的列表翻转了，而没有在生成列表的时候倒过来添加。

## AC源码

```
class Solution(object):
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        result = []
        if not root:
            return result
        curr_level = [root]
        need_reverse = False
        while curr_level:
            level_result = []
            next_level = []
            for temp in curr_level:
                level_result.append(temp.val)
                if temp.left:
                    next_level.append(temp.left)
                if temp.right:
                    next_level.append(temp.right)
            if need_reverse:
                level_result.reverse()
                need_reverse = False
            else:
                need_reverse = True
            result.append(level_result)
            curr_level = next_level
        return result

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Maximum Depth of Binary Tree

## 原题

求一颗二叉树的最大深度，最大深度指跟节点到最底层叶子节点的距离。

注意点：

- 无

例子：

输入：

```
      3
     /\
    9  20
   /\  \
  15  7
 /\
14
```

输出：4

## 解题思路

用递归的方法，当前节点的最大深度就是左节点的最大深度和右节点的最大深度之中取大的加一。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if not root:
            return 0
        return max(self.maxDepth(root.left), self.maxDepth(root.
right)) + 1

if __name__ == "__main__":
    None
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Construct Binary Tree from Preorder and Inorder Traversal

## 原题

通过一棵二叉树的前序和中序排列来得出它的树形结构。

注意点：

- 无

例子：

输入: preorder = [3,9,20,15,14,7], inorder = [9,3,14,15,20,7]

输出：

```
      3
     /\
    9  20
   /\  \
  15  7
 /\
14
```

## 解题思路

1. 因为先序中的节点为根节点,所以取出先序的第一个节点
2. 用先序的第一个节点可以将中序分成左右子树，然后取出先序的第二个节点将左右子树再次划分
3. 当将中序全部划分为单个点时就结束

开始递归的时候用了列表作为参数，结果内存溢出，改用下标作参数，先序和中序序列作为全局变量。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
        self.preorder = preorder
        self.inorder = inorder
        return self._buildTree(0, len(preorder), 0, len(inorder))

    def _buildTree(self, pre_start, pre_end, in_start, in_end):
        if pre_start >= pre_end:
            return None
        root = TreeNode(self.preorder[pre_start])
        offset = self.inorder[in_start:in_end + 1].index(root.val)
        root.left = self._buildTree(pre_start + 1, pre_start + offset + 1, in_start, in_start + offset)
        root.right = self._buildTree(pre_start + offset + 1, pre_end, in_start + offset + 1, in_end)
        return root

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Construct Binary Tree from Inorder and Postorder Traversal

## 原题

通过一棵二叉树的中序和后序排列来得出它的树形结构。

注意点：

- 无

例子：

输入: inorder = [9,3,14,15,20,7], postorder = [9,14,15,7,20,3]

输出：

```
      3
     / \
    9  20
     / \
    15  7
     /
    14
```

## 解题思路

1. 因为后序中的节点为根节点,所以取出后序的最后一个节点
2. 用后序的最后一个节点可以将中序分成左右子树,然后取出后序的倒数第二个节点将左右子树再次划分
3. 当将中序全部划分为单个点时就结束

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def buildTree(self, inorder, postorder):
        """
        :type inorder: List[int]
        :type postorder: List[int]
        :rtype: TreeNode
        """
        self.postorder = postorder
        self.inorder = inorder
        return self._buildTree(0, len(inorder))

    def _buildTree(self, start, end):
        if start < end:
            root = TreeNode(self.postorder.pop())
            index = self.inorder.index(root.val)
            root.right = self._buildTree(index + 1, end)
            root.left = self._buildTree(start, index)
            return root

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Binary Tree Level Order Traversal II

## 原题

实现树的广度优先遍历的倒序遍历，即从最底层依次向上遍历，每一层上的数据按照从左到右的顺序排列。

注意点：

- 无

例子:

输入:

```
      3
     /\
    9  20
   /\  \
  15   7
```

输出:

```
[
  [15,7],
  [9,20],
  [3]
]
```

## 解题思路

直接复用了 [Binary Tree Level Order Traversal](#) 的代码，只是最后把序列翻转了。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def levelOrderBottom(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        result = []
        if not root:
            return result
        curr_level = [root]
        while curr_level:
            level_result = []
            next_level = []
            for temp in curr_level:
                level_result.append(temp.val)
                if temp.left:
                    next_level.append(temp.left)
                if temp.right:
                    next_level.append(temp.right)
            result.append(level_result)
            curr_level = next_level
        result.reverse()
        return result

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Convert Sorted Array to Binary Search Tree

## 原题

给定一个升序的序列，将它转化为高度平衡的二叉搜索树。

注意点：

- 同一个序列转化成的二叉搜索树可能有多种

例子：

输入: `nums = [1,2,3]`

输出：

```
    2
   /\
  1  3
```

## 解题思路

平衡二叉搜索树的要求是每个节点左右子树的高度差最多为1。那只要取序列的中间数作为根节点，左边的序列再组成它的左子树，右边的序列组成它的右子树，递归完成构造。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        return self._sortedArrayToBST(nums, 0, len(nums))

    def _sortedArrayToBST(self, nums, left, right):
        if left == right:
            return None
        mid = (left + right) >> 1
        root = TreeNode(nums[mid])
        root.left = self._sortedArrayToBST(nums, left, mid)
        root.right = self._sortedArrayToBST(nums, mid + 1, right)
        return root

if __name__ == "__main__":
    None
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Convert Sorted List to Binary Search Tree

## 原题

给定一个升序的单向链表，将它转化为高度平衡的二叉搜索树。

注意点：

- 同一个序列转化成的二叉搜索树可能有多种

例子：

输入: nums = 1->2->3

输出：

```
    2
   /\
  1  3
```

## 解题思路

这题就是 [Convert Sorted Array to Binary Search Tree](#) 的升级版，可以先把链表转化为列表再解答。如果直接用链表解决的话，可以看出链表的特点是从头到尾依次遍历，因为是递增的，所以也就是从小到大依次遍历。而二叉搜索树的中序遍历的结果就是一个递增的序列，所以只要按照树的中序遍历的方式来构造即可。

## AC源码

```
# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
```



```
        self.next = None

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def sortedListToBST(self, head):
        """
        :type head: ListNode
        :rtype: TreeNode
        """
        node, length = head, 0
        while node:
            node = node.next
            length += 1
        self.curr = head
        return self._sortedListToBST(0, length - 1)

    def _sortedListToBST(self, left, right):
        if left > right:
            return None
        mid = (left + right) // 2
        left = self._sortedListToBST(left, mid - 1)
        root = TreeNode(self.curr.val)
        root.left = left
        self.curr = self.curr.next
        root.right = self._sortedListToBST(mid + 1, right)
        return root

if __name__ == "__main__":
    None
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Balanced Binary Tree

## 原题

判断一棵二叉树是否是平衡二叉树，只有当每个节点的左右两棵子树的高度差不大于1时，这棵树才是平衡的。

注意点：

- 无

例子:

输入:

```
      3
     /\
    9  20
   /\  \
  15   7
 /\
14
```

输出: False

## 解题思路

一个节点的高度是它左右两棵子树的高度较大值加一，为了使代码简练，定义不平衡的子树的高度为-1，所以在计算每个节点的高度时，要额外判断左右两棵子树是否平衡，如果不平衡（左子树不平衡 | 右子树不平衡 | 左右子树高度差大于1）就直接返回-1。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def isBalanced(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        return self._isBalanced(root) >= 0

    def _isBalanced(self, root):
        if not root:
            return 0
        left, right = self._isBalanced(root.left), self._isBalanced(root.right)
        if left >= 0 and right >= 0 and abs(left - right) <= 1:
            return 1 + max(left, right)
        else:
            return -1

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Minimum Depth of Binary Tree

## 原题

求一棵二叉树的最小高度，即从根节点到最近叶子节点的路径经过的节点数。

注意点：

- 无

例子:

输入:

```
    3
   / \
  9  20
   / \
  15  7
   /
  14
```

输出: 2

## 解题思路

可以通过树的广度优先遍历 [Binary Tree Level Order Traversal](#) 来实现，在广度优先遍历的过程中，每遍历一层就高度加一，如果某一个节点是叶子节点，那么当前的高度就是最小高度。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def minDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root is None:
            return 0
        depth, curr_level = 0, [root]
        while curr_level:
            depth += 1
            next_level = []
            for n in curr_level:
                left, right = n.left, n.right
                if left is None and right is None:
                    return depth
                if left:
                    next_level.append(left)
                if right:
                    next_level.append(right)
            curr_level = next_level
        return depth

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode 解题之 Path Sum

### 原题

判断一棵二叉树是否有一条从根节点到某一叶子节点的路径，该路径上所有节点的和为一个特定值。

注意点：

- 无

例子：

输入: sum = 12

```
      3
     /\
    9 20
   /\ 
  15 7
  /\
 14
```

输出: True (3->9)

### 解题思路

直接通过递归来解决，判断一个节点时，先把要求的值先减去该节点的值，如果剩下要求的值为0且当前的节点就是一个叶子节点，那么从根节点到当前节点的路径就符合题目的要求。否则就要继续递归当前节点的左右节点。

### AC源码



```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root:
            return False
        sum -= root.val
        if sum == 0 and root.left is None and root.right is None:
            return True
        return self.hasPathSum(root.left, sum) or self.hasPathSum(root.right, sum)

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Path Sum II

### 原题

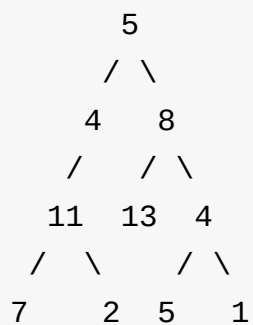
找出一棵二叉树所有的从根节点到某一叶子节点的路径，该路径上所有节点的和为一个特定值。

注意点：

- 无

例子：

输入：



输出：

```
[
  [5, 4, 11, 2],
  [5, 8, 4, 5]
]
```

### 解题思路

**Path Sum** 是判断是否有这样一条路径，现在要把所有的路径都求出来，那只要在 **dfs** 时将符合要求的路径加入到结果集。注意加入结果集的数据不要是引用，否则可能之后会再次被修改。

## AC源码

```

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: List[List[int]]
        """
        result = []
        self._pathSum(root, sum, [], result)
        return result

    def _pathSum(self, root, sum, curr, result):
        if not root:
            return
        sum -= root.val
        if sum == 0 and root.left is None and root.right is None:
            result.append(curr + [root.val])
        if root.left:
            self._pathSum(root.left, sum, curr + [root.val], result)
        if root.right:
            self._pathSum(root.right, sum, curr + [root.val], result)

if __name__ == "__main__":
    None

```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Flatten Binary Tree to Linked List

## 原题

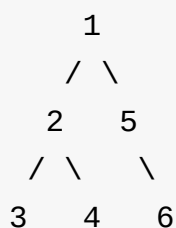
把一棵二叉树变为链表，也就是一棵所有节点要么没有子节点，要么只有右节点的二叉树。

注意点：

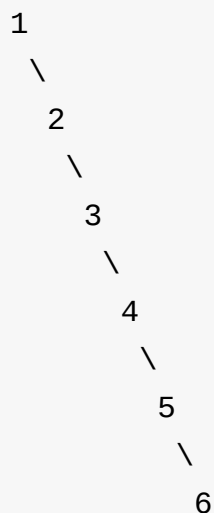
- 无

例子：

输入：



输出：



## 解题思路

可以看出来变化后每个节点其实都是指向了在先序遍历中的后一个节点。所以就通过栈的方式来先序遍历原树，如果一个节点有左节点，那么把它的右节点压栈（如果有的话），右指针指向原来的左节点；如果一个节点没有子节点，应该把它的右指针指向栈顶的节点。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def flatten(self, root):
        """
        :type root: TreeNode
        :rtype: void Do not return anything, modify root in-place instead.
        """
        stack = []
        while root:
            if root.left:
                if root.right:
                    stack.append(root.right)
                root.right, root.left = root.left, None
            if not root.right and stack:
                root.right = stack.pop()
            root = root.right

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Distinct Subsequences

---

### 原题

给定两个字符串S和T，求T有多少种从属于S的子序列的情况。或者说S可以删除它自己任意个字符，但是不能改变字符的相对位置，那一共有多少种删法可以使S变为T。

注意点：

- 删除任意个字符包括不删除字符

例子:

输入: s = "rabbbit", t = "rabbit"

输出: 3

### 解题思路

典型的动态规划问题， $dp[i][j]$  表示字符串S[:i]和T[:j]的不同子序列数目，如果S[i-1]和T[j-1]不相等，那么只能在S[:i-1]和T[:j]中匹配，即 $dp[i][j] = dp[i-1][j]$ ；而当S[i-1]和T[j-1]相等时，可以是这两个字符正好匹配，也可以忽略S[i-1]，使T[j-1]在S[:i-1]中匹配，所以  $dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$ 。进一步观察可以将二维数组压缩为一维，只需要从后往前计算即可。

### AC源码

```
class Solution(object):
    def numDistinct(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: int
        """
        m = len(s)
        n = len(t)
        dp = [0 for __ in range(n + 1)]
        dp[0] = 1
        for i in range(m):
            for j in range(n - 1, -1, -1):
                if t[j] == s[i]:
                    dp[j + 1] += dp[j]
        return dp[-1]

if __name__ == "__main__":
    assert Solution().numDistinct("rabbbit", "rabbit") == 3
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Populating Next Right Pointers in Each Node

### 原题

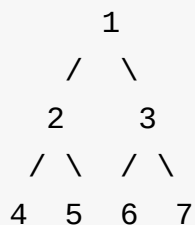
为二叉树的节点都添加一个next指针，指向跟它在同一高度的右边的节点，如果右边没有节点，就指向None。

注意点：

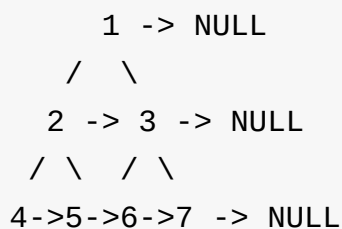
- 最好只用常量的空间
- 这是一棵完全二叉树

例子：

输入：



输出：



### 解题思路

这里采用了思路最清晰的解法，但是用的空间不是常量的。可以看出其实就是把树的每一层都串联起来了，要处理每一层的节点，可以使用广度优先遍历，把每一层的节点暂存在列表中，再把这些节点都连接起来。

## AC源码

```
# Definition for binary tree with next pointer.
class TreeLinkNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
        self.next = None

class Solution(object):
    def connect(self, root):
        """
        :type root: TreeLinkNode
        :rtype: nothing
        """
        if not root:
            return
        current_level = [root]
        while current_level:
            next_level = []
            for node in current_level:
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)
            for i in range(len(next_level) - 1):
                next_level[i].next = next_level[i + 1]
            current_level = next_level

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Populating Next Right Pointers in Each Node II

### 原题

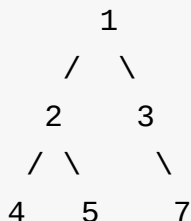
为二叉树的节点都添加一个next指针，指向跟它在同一高度的右边的节点，如果右边没有节点，就指向None。与 [Populating Next Right Pointers in Each Node](#) 的区别就是这里的二叉树可以是不完全二叉树。

注意点：

- 最好只用常量的空间

例子：

输入：



输出：



### 解题思路

**Populating Next Right Pointers in Each Node** 中广度优先遍历的方法已经能够处理不是完全二叉树的情况，但没有做到常量空间，这里对代码进行一下优化。仔细想一下，既然上一层的节点已经通过`next`指针连起来了，那么就只要能得到上一层的第一个节点就可以依次把上一层节点的子节点串联起来了。通过添加一个假节点来标记当前行的首节点。其中`root`表示上一行遍历到的节点，`node`表示当前行的节点。

## AC源码

```
# Definition for binary tree with next pointer.
class TreeLinkNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
        self.next = None

class Solution(object):
    def connect(self, root):
        """
        :type root: TreeLinkNode
        :rtype: nothing
        """
        dummy = TreeLinkNode(-1)
        node = dummy
        while root:
            while root:
                node.next = root.left
                node = node.next or node
                node.next = root.right
                node = node.next or node
                root = root.next
            root, node = dummy.next, dummy

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode 解题之 Pascal's Triangle

### 原题

要求得到一个n行的杨辉三角。

注意点：

- 无

例子：

输入: numRows = 5

输出：

```
[
  [1],
  [1,1],
  [1,2,1],
  [1,3,3,1],
  [1,4,6,4,1]
]
```

### 解题思路

杨辉三角的特点是每一行的第一个和最后一个元素是1，其它元素是上一行它左右两个元素之和。以[1,3,3,1]为例，下一行的中间元素就是[1+3,3+3,3+1]，也就是[1,3,3]和[3,3,1]对应数字求和。

### AC源码

```
class Solution(object):
    def generate(self, numRows):
        """
        :type numRows: int
        :rtype: List[List[int]]
        """
        if not numRows:
            return []
        result = [[1]]
        while numRows > 1:
            result.append([1] + [a + b for a, b in zip(result[-1]
[: -1], result[-1][1:])] + [1])
            numRows -= 1
        return result

if __name__ == "__main__":
    assert Solution().generate(4) == [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

## LeetCode解题之Pascal's Triangle II

### 原题

用 $O(k)$ 的空间得到杨辉三角第 $k$ 行的数值。

注意点：

- 从0开始计算行数，即第0行为[1]

例子：

输入:  $k = 3$

输出: [1,3,3,1]

```
[
  [1],
  [1,1],
  [1,2,1],
  [1,3,3,1],
  [1,4,6,4,1]
]
```

### 解题思路

现在要考虑的是在 [Pascal's Triangle](#) 的基础上节约空间，我们可以知道第 $k$ 行需要 $(k+1)$ 的空间，且下一行占用的长度都比上一行长，所以在计算下一行的值时适合从后往前计算。

### AC源码

```
class Solution(object):
    def getRow(self, rowIndex):
        """
        :type rowIndex: int
        :rtype: List[int]
        """
        result = [1] * (rowIndex + 1)
        for i in range(2, rowIndex + 1):
            for j in range(1, i):
                result[i - j] += result[i - j - 1]
        return result

if __name__ == "__main__":
    assert Solution().getRow(3) == [1, 3, 3, 1]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Triangle

## 原题

将一个二维数组排列成金字塔的形状，找到一条从塔顶到塔底的路径，使路径上的所有点的和最小，从上一层到下一层只能挑相邻的两个点中的一个。

注意点：

- 最好将空间复杂度控制在 $O(n)$ ， $n$ 是金字塔的高度

例子:

输入:

```
[
  [2],
  [3,4],
  [6,5,7],
  [4,1,8,3]
]
```

输出: 11 ( $2 + 3 + 5 + 1 = 11$ )

## 解题思路

典型的动态规划问题，先将问题转化一下，把每一行的数列都左对齐，如下：

```
[
  [2],
  [3,4],
  [6,5,7],
  [4,1,8,3]
]
```

可以看出来，其实上一行到下一行就两个选择，横坐标不变或加一。dp[i]表示从底层到这一层的第i个元素所有路径中最小的和。递推关系就是  $dp[j] = triangle[i][j] + \min(dp[j], dp[j + 1])$ ，即下一行与它相邻的两个节点中和比较小的再加上它自己的值。

## AC源码

```
class Solution(object):
    def minimumTotal(self, triangle):
        """
        :type triangle: List[List[int]]
        :rtype: int
        """
        n = len(triangle)
        dp = triangle[-1]
        for i in range(n - 2, -1, -1):
            for j in range(i + 1):
                dp[j] = triangle[i][j] + min(dp[j], dp[j + 1])
        return dp[0]

if __name__ == "__main__":
    assert Solution().minimumTotal([
        [2],
        [3, 4],
        [6, 5, 7],
        [4, 1, 8, 3]
    ]) == 11
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Best Time to Buy and Sell Stock

---

## 原题

给定每天的股票价格，如果只允许进行一轮交易，也就是买进一次和卖出一次，求所能获得的最大的利润。

注意点：

- 无

例子：

输入: `prices = [2, 4, 6, 1, 3, 8, 3]`

输出: 7(在价格为1的时候买入，价格为8时卖出)

## 解题思路

从前往后遍历数列，把当前出现过的最低价格作为买入价格，并计算以当前价格出售的收益，整个遍历过程中，出现过的最大收益就是题目所求。

## AC源码

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if len(prices) < 2:
            return 0
        min_price = prices[0]
        max_profit = 0
        for price in prices:
            if price < min_price:
                min_price = price
            if price - min_price > max_profit:
                max_profit = price - min_price
        return max_profit

if __name__ == "__main__":
    assert Solution().maxProfit([2, 4, 6, 1, 3, 8, 3]) == 7
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode解题之Best Time to Buy and Sell Stock II

---

## 原题

给定每天的股票价格，如果允许进行多次交易，即可以多次买入和卖出，但手中最多只能持有一支股票，在再次买入的时候必须将之前的股票卖出，求能获取的最大利润。

注意点：

- 无

例子:

输入: prices = [2, 4, 6, 1, 3, 8, 3]

输出: 11([2,6]、[1,8]是两次进行买入卖出的时机)

## 解题思路

可以进行多次交易的话，为了获取最多的利润，应该在每一段价格上升的区间的开头买入，末尾卖出。从前往后遍历数组，如果价格下跌，则在前一天卖出，在下落的那天再次买入。不要忘记最后的上升段之后没有下落的情况，要额外加上。

## AC源码

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if not prices:
            return 0
        low = high = prices[0]
        profit = 0
        for i in range(1, len(prices)):
            if prices[i] >= prices[i - 1]:
                high = prices[i]
            else:
                profit += high - low
                low = high = prices[i]
        profit += high - low
        return profit

if __name__ == "__main__":
    assert Solution().maxProfit([2, 4, 6, 1, 3, 8, 3]) == 11
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Best Time to Buy and Sell Stock III

## 原题

给定每天的股票价格，如果最多允许两次交易，但手中最多只能持有一支股票，在再次买入的时候必须将之前的股票卖出，求能获取的最大利润。

注意点：

- 无

例子：

输入: prices = [2, 4, 6, 1, 3, 8, 3]

输出: 11([2,6]、[1,8]是两次进行买入卖出的时机)

## 解题思路

因为最多只能进行两次交易，所以可以将时间一划为二，分别找这两段时间内进行一次（也可能不进行交易）所能获得的最大利润（方法参见 [Best Time to Buy and Sell Stock](#)），将两者相加就是在这种划分情况下最多进行两次交易所能获取的最大利润。遍历所有的划分可能，就能找出最终的最大利润。如果每次都直接将时间段直接调用Best Time to Buy and Sell Stock的方法，复杂的用例会超时。我们可以先从前往后遍历，并缓存最多进行一次交易所能获取的最大利润；再从后往前遍历计算最多进行一次交易所能获取的最大利润，与对应的缓存相加就是在一次划分下的最大利润。

## AC源码

```

class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        total_max_profit = 0
        n = len(prices)
        first_profits = [0] * n
        min_price = float('inf')

        for i in range(n):
            min_price = min(min_price, prices[i])
            total_max_profit = max(total_max_profit, prices[i] -
min_price)
            first_profits[i] = total_max_profit

        max_profit = 0
        max_price = float('-inf')
        for i in range(n - 1, 0, -1):
            max_price = max(max_price, prices[i])
            max_profit = max(max_profit, max_price - prices[i])
            total_max_profit = max(total_max_profit, max_profit
+ first_profits[i - 1])
        return total_max_profit

if __name__ == "__main__":
    assert Solution().maxProfit([2, 4, 6, 1, 3, 8, 3]) == 11
    assert Solution().maxProfit([1, 2]) == 1

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Binary Tree Maximum Path Sum

### 原题

求一棵二叉树中最大的路径和。该路径可以是二叉树中某一节点到树中任意一个节点的所经过的路径，不允许重复经过一个节点，不必经过根节点。

注意点：

- 无

例子:

输入:

```
    1
   / \
  2   3
```

输出: 6

### 解题思路

采用递归的方法，我们知道一条路径必定有一个节点最接近根节点，而该条路径的和就是这个节点的值加上它左右路径的和。我们现在要求最大路径和，那么就要分别得到左右两条路径的最大和。而左路径的最大和为左节点的值加上它左右路径中较大的路径和，右路径最大和为右节点的值加上它左右路径中较大的路径和。注意如果某条子路径的和为负，应该将该条子路径直接砍掉。

### AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def maxPathSum(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.maxSum = float('-inf')
        self._maxPathSum(root)
        return self.maxSum

    def _maxPathSum(self, root):
        if root is None:
            return 0
        left = self._maxPathSum(root.left)
        right = self._maxPathSum(root.right)
        left = left if left > 0 else 0
        right = right if right > 0 else 0
        self.maxSum = max(self.maxSum, root.val + left + right)
        return max(left, right) + root.val
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Valid Palindrom

---

### 原题

判断一个字符串是否是回文字符串，只考虑字母和数字，并且忽略大小写。

注意点：

- 空字符串在这里也定义为回文串

例子：

输入: s = "A man, a plan, a canal: Panama"

输出: True

输入: s = "race a car"

输出: False

### 解题思路

先将字符串中的非字母和数字的字符去除，同时把所有的字母转换为小写，再判断新的字符串与自己翻转的字符串是否相等。这种方法代码比较简练，也可以采用双指针的方法从两端开始遍历数列来进行判断。

### AC源码

```
class Solution(object):
    def isPalindrome(self, s):
        """
        :type s: str
        :rtype: bool
        """
        alphanumericS = [c for c in s.lower() if c.isalnum()]
        return alphanumericS == alphanumericS[::-1]

if __name__ == "__main__":
    assert Solution().isPalindrome("A man, a plan, a canal: Panama") == True
    assert Solution().isPalindrome("race a car") == False
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode 解题之 Word Ladder II

### 原题

给定一个起始字符串和一个目标字符串，现在将起始字符串按照特定的变换规则转换为目标字符串，求所有转换次数最少的转换过程。转换规则为每次只能改变字符串中的一个字符，且每次转换后的字符串都要在给定的字符串集合中。

注意点：

- 所有给出的字符串的长度都相等
- 所有的字符都为小写字母

例子：

输入: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

输出：

```
[
  ["hit", "hot", "dot", "dog", "cog"],
  ["hit", "hot", "lot", "log", "cog"]
]
```

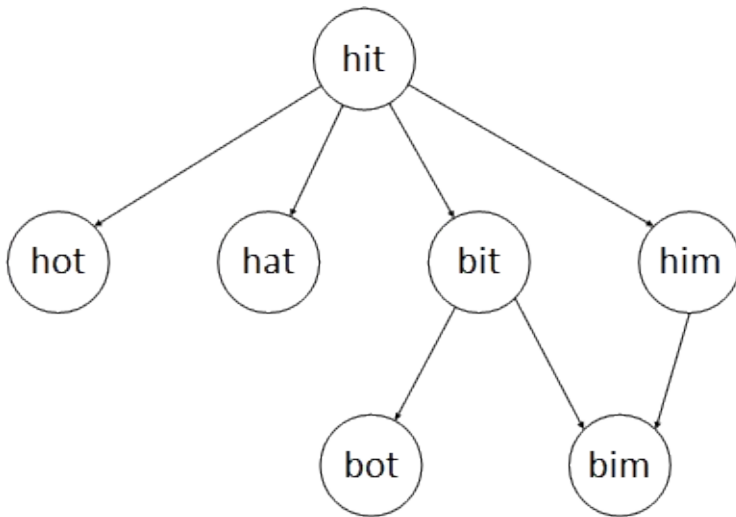
### 解题思路

整体思想与 [Word Ladder](#) 一样，都是采用DFS的方法。不过要做出以下的变化：

1. 在 Word Ladder 中，广度优先遍历是从上层到下层一层层遍历下去的，还是以下面的图为例，树结构可能是中间宽两头窄的情况。而一个节点转换为下层节点时，要依次将它字符串中的每个字符用其他字母替换后与给定字符串集合进行比较。如果树某一层的节点非常多，那么这个尝试转换的操作开销就很大。还需要明确的一点是，题目中的转换是可逆的，A 可以转换为 B，那么 B 也可以转换为 A，且 A 和 B 都能转换为它们转换过程的某一个状态 C。综上所述，我们可以从起始字符串和目标字符串同时进行转换，哪一端的节点数目少，我们就

选这些节点继续进行转换，直到它们汇合到同一个节点或者转换终止（也就是下一层没有节点）。代码中由变量`is_forward`来表示从哪一端转换。

2. 现在是要求所有最少转换次数的转换方法，所以要将所有的转换可能都找出来。如图中`bit`和`him`转换为`bim`的转换关系我们都要找出来。
3. 我们还需要记录转换的路径，我们将从上一层到下一层的转换关系记录下来，等到确定能够转换成功了，再通过深度优先遍历的方法将转换路径组装起来。



注：图中的有些单词没有意义，只是单纯为了举例子，图对应的起始字符串为`hit`，给定的字符串集合为`{"hot","hat","bit","him","bot","bim"}`

## AC源码

```

class Solution(object):
    def findLadders(self, beginWord, endWord, wordlist):
        """
        :type beginWord: str
        :type endWord: str
        :type wordlist: Set[str]
        :rtype: List[List[int]]
        """

        def bfs(front_level, end_level, is_forward, word_set, path_dic):
            if len(front_level) == 0:
                return False
            if len(front_level) > len(end_level):
                return bfs(end_level, front_level, not is_forward, word_set, path_dic)

```

```

d, word_set, path_dic)
    for word in (front_level | end_level):
        word_set.discard(word)
    next_level = set()
    done = False
    while front_level:
        word = front_level.pop()
        for c in 'abcdefghijklmnopqrstuvwxyz':
            for i in range(len(word)):
                new_word = word[:i] + c + word[i + 1:]
                if new_word in end_level:
                    done = True
                    add_path(word, new_word, is_forward,
path_dic)
                else:
                    if new_word in word_set:
                        next_level.add(new_word)
                        add_path(word, new_word, is_forw
ard, path_dic)
    return done or bfs(next_level, end_level, is_forward
, word_set, path_dic)

def add_path(word, new_word, is_forward, path_dic):
    if is_forward:
        path_dic[word] = path_dic.get(word, []) + [new_w
ord]
    else:
        path_dic[new_word] = path_dic.get(new_word, [])
+ [word]

def construct_path(word, end_word, path_dic, path, paths)
:
    if word == end_word:
        paths.append(path)
        return
    if word in path_dic:
        for item in path_dic[word]:
            construct_path(item, end_word, path_dic, pat
h + [item], paths)

```

```
        front_level, end_level = {beginWord}, {endWord}
        path_dic = {}
        bfs(front_level, end_level, True, wordlist, path_dic)
        path, paths = [beginWord], []
        construct_path(beginWord, endWord, path_dic, path, paths
    )

    return paths

if __name__ == "__main__":
    assert Solution().findLadders("hit", "cog", {"hot", "dot", "dog", "lot", "log"}) == [
        ["hit", "hot", "dot", "dog", "cog"],
        ["hit", "hot", "lot", "log", "cog"]
    ]
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Word Ladder

## 原题

给定一个起始字符串和一个目标字符串，现在将起始字符串按照特定的变换规则转换为目标字符串，求最少要进行多少次转换。转换规则为每次只能改变字符串中的一个字符，且每次转换后的字符串都要在给定的字符串集合中。

注意点：

- 如果无法完成转换则返回0
- 所有给出的字符串的长度都相等
- 所有的字符都为小写字母

例子：

输入: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

输出: 5 ("hit" -> "hot" -> "dot" -> "dog" -> "cog")

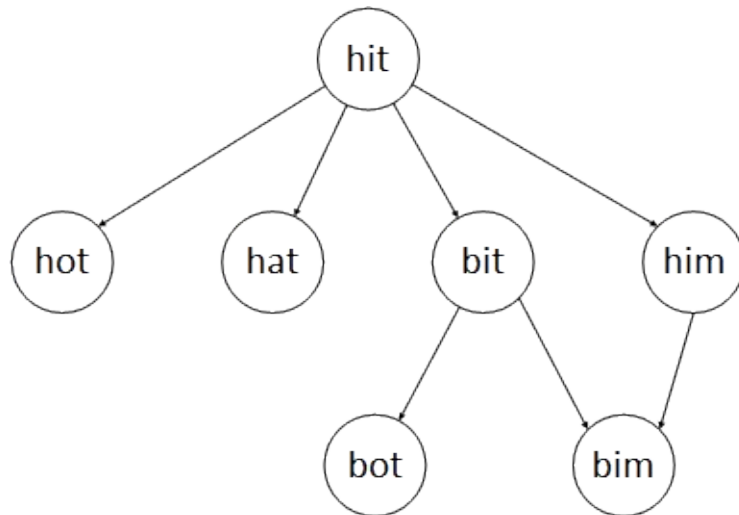
## 解题思路

因为每次变换后的字符串都要在给定的字符串组中，所以每次变化的情况都是有限的。现在把变化过程做成一个树的结构，由某一个字符串变化而来的字符串就成为该字符串的子树。参看下图的例子，我们可以得到以下几点结论：

1. 我们把起始字符串当成根节点，如果在变化过程中，某一个节点是目标字符串，那么就找到了一条变化路径。
2. 节点所在的高度能够反映出变化到该节点时经历了几次变化，如hot在根节点的下一层，表示变化了一次，hut和bot在更下一层，表示变化了两次。
3. 在树上层出现过的字符串没必要在下层再次出现，因为如果该字符串是转换过程中必须经过的中间字符串，那么应该挑选上层的该字符串继续进行变化，它的转换次数少。
4. 如果上一层有多个字符串可以转换为下一层同一个字符串，那么只需要找到其中一个转换关系即可，如例子中的bit和him都可以转为bim，我们只需要知道有

一条关系可以走到**bim**就可以了，没必要找到所有的转换关系，因为这样已经可以确定进行两次转换就能变为**bim**。

5. 基于第3和第4点，当集合中的字符串在树中出现后，就可以把它从集合中删除。这样可以防止字符串不断地循环转化。
6. 至此，这个问题就变为一个深度优先遍历问题，只需要依次遍历每一层的节点，如果在该层找到了目标字符串，只要返回相应的变化次数。如果到某一层树的节点无法继续向下延伸，且没有找到目标字符串，那么就是不存在这样的



转换关系，返回0。

注：图中的有些单词没有意义，只是单纯为了举例子，图对应的起始字符串为hit，给定的字符串集合为{"hot","hat","bit","him","bot","bim"}。

## AC源码

```
class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: Set[str]
        :rtype: int
        """
        wordList.add(endWord)
        cur_level = [beginWord]
        next_level = []
        depth = 1
        n = len(beginWord)
        while cur_level:
            for item in cur_level:
                if item == endWord:
                    return depth
                for i in range(n):
                    for c in 'abcdefghijklmnopqrstuvwxyz':
                        word = item[:i] + c + item[i + 1:]
                        if word in wordList:
                            wordList.remove(word)
                            next_level.append(word)
            depth += 1
            cur_level = next_level
            next_level = []
        return 0

if __name__ == "__main__":
    assert Solution().ladderLength("hit", "cog", {"hot", "dot",
        "dog", "lot", "log"}) == 5
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Longest Consecutive Sequence

---

## 原题

给定一组无序的整数，找出其中连续整数的最长长度。

注意点：

- 算法时间复杂度为 $O(n)$

例子：

输入: `nums = [100, 4, 200, 1, 3, 2]`

输出: 4 (连续整数为[1,2,3,4])

## 解题思路

采用了哈希法，所有的整数放入集合中，取出某一个元素向两边扩展，如果两边的元素也在集合中，从集合中去除这些元素的同时继续判断外围的元素，这样可以得到每一个连续整数组的长度。从这些长度中找到最长的长度。

## AC源码



```
class Solution(object):
    def longestConsecutive(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        numset, maxlen = set(nums), 0
        for n in set(nums):
            currlen, tmp = 1, n + 1
            while tmp in numset:
                currlen += 1
                numset.discard(tmp)
                tmp += 1
            tmp = n - 1
            while tmp in numset:
                currlen += 1
                numset.discard(tmp)
                tmp -= 1
            maxlen = max(maxlen, currlen)
        return maxlen

if __name__ == "__main__":
    assert Solution().longestConsecutive([100, 4, 200, 1, 3, 2])
    == 4
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Sum Root to Leaf Numbers

## 原题

一棵树的每个节点都是0-9中的某一个数字，现在把从根节点到某一个叶子节点之间所有节点的数字依次连接起来组成一个新的数字。要求所有从根节点到叶子节点组成的数字的和。

注意点：

- 无

例子：

输入：

```
    1
   /\
  2  3
```

输出: 25

## 解题思路

如果在叶子节点下再添加一个节点，则表示的数字可以通过将之前的数字乘以10再加上添加节点的数字。通过这一递推关系，我们可以递归地计算每条路径上的表示的数字，代码中参数s表示到当前节点之前该路径上所组成的数字。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def sumNumbers(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        return self._sumNumbers(root, 0)

    def _sumNumbers(self, root, s):
        if root is None:
            return 0
        s = s * 10 + root.val
        return sum([self._sumNumbers(r, s) for r in (root.left,
root.right)]) or s

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Surrounded Regions

### 原题

有一个二维的面板，上面由"X"或者"O"填充。现在要求将被"X"包围的"O"都改成"X"。

注意点：

- 无

例子：

输入：

```
X X X X
X O O X
X X O X
X O X X
```

输出：

```
X X X X
X X X X
X X X X
X O X X
```

### 解题思路

直接去找被X包围的O比较麻烦，不如转换一下思路，找出哪些O是没有被X包围的。首先在面板四周的O肯定是没有被X包围的，与它们相连的O也是没有被包围的，其它的O都是被X包围的。问题简化为将与四周的O相连的O都找出来，这些点

不用变，其它点都变为X。首先将四周的O压入栈内，依次访问栈内元素，并将它们标记，接着去判断它们四周的元素是否也是O，如果是且没有被标记过，则将其压入栈中。当遍历完栈中的元素后，将有标记的元素变为O，其余都是X。

## AC源码

```
class Solution(object):
    def solve(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return
        n = len(board)
        m = len(board[0])
        queue = []
        # Get all 'O' on edge
        for i in range(n):
            for j in range(m):
                if ((i in (0, n - 1)) or (j in (0, m - 1))) and board[i][j] == 'O':
                    queue.append((i, j))
        # Mark all 'O' which can connect to 'O' on edge
        while queue:
            r, c = queue.pop(0)
            if 0 <= r < n and 0 <= c < m and board[r][c] == 'O':
                board[r][c] = 'M'
                if r - 1 >= 0 and board[r - 1][c] == 'O':
                    queue.append((r - 1, c))
                if r + 1 < n and board[r + 1][c] == 'O':
                    queue.append((r + 1, c))
                if c - 1 >= 0 and board[r][c - 1] == 'O':
                    queue.append((r, c - 1))
                if c + 1 < m and board[r][c + 1] == 'O':
                    queue.append((r, c + 1))
        # Update characters
```

```
    for i in range(n):
        for j in range(m):
            if board[i][j] == 'M':
                board[i][j] = 'O'
            else:
                board[i][j] = 'X'

if __name__ == "__main__":
    board = [
        ['X', 'X', 'X', 'X'],
        ['X', 'O', 'O', 'X'],
        ['X', 'X', 'O', 'X'],
        ['X', 'O', 'X', 'X']
    ]
    expected_board = [
        ['X', 'X', 'X', 'X'],
        ['X', 'X', 'X', 'X'],
        ['X', 'X', 'X', 'X'],
        ['X', 'O', 'X', 'X']
    ]
    Solution().solve(board)
    assert board == expected_board
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之 Palindrome Partitioning

---

### 原题

将一个字符串分割成若干个子字符串，使得子字符串都是回文字符串，要求列出所有的分割方案。

注意点：

- 无

例子：

输入: `s = "aab"`

输出: `result = [["a", "a", "b"], ["aa", "b"]]`

### 解题思路

采用了最简单的递归方法，将一个字符串分为前后两部分，如果第一部分是一个回文字符串，则对第二部分再次分割，不断递归，直到递归的终止条件——字符串为空为止；如果第一部分不是一个回文字符串，则尝试下一种分割方法。

### AC源码

```
class Solution(object):
    def partition(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
        """
        if not s:
            return [[]]
        result = []
        for i in range(len(s)):
            if self.isPalindrome(s[:i + 1]):
                for r in self.partition(s[i + 1:]):
                    result.append([s[:i + 1]] + r)
        return result

    def isPalindrome(self, s):
        return s == s[::-1]

if __name__ == "__main__":
    assert Solution().partition("aab") == [
        ["a", "a", "b"],
        ["aa", "b"]
    ]
```

欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Palindrome Partitioning II

## 原题

将一个字符串分割成若干个子字符串，使得子字符串都是回文字符串，要求最少需要几次分割能够满足需求。

注意点：

- 无

例子：

输入: `s = "aab"`

输出: 1 (分为 "aa" 和 "b")

## 解题思路

可以通过动态规划解决， $dp[i]$  表示字符串  $s[:i+1]$  需要的最少的切割次数， $dp[i]$  的初始值为  $i$ ，因为长度为  $i+1$  的字符串最多切割  $i$  次就能满足题目要求。当添加一个字符后，我们需要依次判断以它为末尾的子字符串是否是回文字符串，如果是，则要计算剩余字符串需要的最少切割次数加上一次是否能使当前的最少切割次数更少，注意如果此时整个字符串就是回文字符串，则最少切割次数为 0。递推表达式如下：

```
dp[i] = 0, 如果 s[:i+1] 是回文串
dp[i] = min(dp[i], dp[j-1]+1), 如果 s[j:i+1] 是回文串
```

为了减少判断回文字符串时的计算，我们通过一个二维数组 `isPal[j][i]` 来缓存判断结果，`isPal[j][i]` 表示字符串  $s[j:i+1]$  是否是回文字符串。

## AC 源码

```
class Solution(object):
    def minCut(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
        """
        n = len(s)
        dp = [0 for __ in range(n)]
        isPal = [[False for __ in range(n)] for __ in range(n)]
        for i in range(n):
            m = i
            for j in range(i + 1):
                if s[j] == s[i] and (j + 1 > i - 1 or isPal[j + 1
][i - 1]):
                    isPal[j][i] = True
                    m = 0 if j == 0 else min(m, dp[j - 1] + 1)
            dp[i] = m
        return dp[-1]

if __name__ == "__main__":
    assert Solution().minCut("aab") == 1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Clone Graph

## 原题

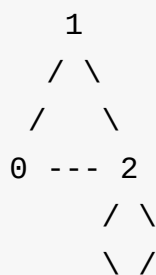
对一个无向图进行复制，图中的每一个节点都有自己的标签和自己相邻节点的列表。

注意点：

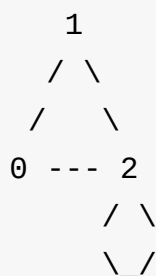
- 无

例子:

输入:



输出:



## 解题思路

因为图中可能存在环，所以直接将节点和它的相邻节点进行复制，并对它的相邻节点进行相同操作可能会进入死循环。为了避免循环访问，要对已经复制过的节点进行缓存，我们通过一个由标志和节点组成的字典来记录已经访问过的节点。当我们通过相邻关系来访问一个节点时，如果它是第一次被访问，则要将其加入一个栈中，在栈中的元素表示要继续访问它相邻的元素，并记录它已经被访问过，同时要跟新已经被访问过的节点中与其相邻的节点的邻居列表。当栈为空时，表示所有的节点都已经访问完毕，图也复制成功。

## AC源码

```

# Definition for a undirected graph node
class UndirectedGraphNode(object):
    def __init__(self, x):
        self.label = x
        self.neighbors = []

class Solution(object):
    def cloneGraph(self, node):
        """
        :type node: UndirectedGraphNode
        :rtype: UndirectedGraphNode
        """
        if not node:
            return node
        visited = {}
        first = UndirectedGraphNode(node.label)
        visited[node.label] = first
        stack = [node]
        while stack:
            top = stack.pop()
            for n in top.neighbors:
                if n.label not in visited:
                    visited[n.label] = UndirectedGraphNode(n.la
el)
                    stack.append(n)
                    visited[top.label].neighbors.append(visited[n.la
bel])
        return first

if __name__ == "__main__":
    None

```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



## LeetCode解题之Gas Station

### 原题

在一条环形的路上有N个加油站，每个加油站里有 $gas[i]$ 的汽油，从第i个加油站到第 $i+1$ 个加油站需要花费 $cost[i]$ 的汽油。假设汽车的油箱可以装无数的汽油，判断一辆没有油的汽车是否可以从其中的某一个加油站出发并行驶一圈后返回该加油站。如果可以的话，返回起始加油站的下标，否则返回-1。

注意点：

- 如果有答案的话，答案是唯一的
- 不用考虑逆向行驶

例子：

输入:  $gas = [5, 1, 2, 3, 4]$ ,  $cost = [4, 4, 1, 5, 1]$

输出: 4

### 解题思路

选择从一个加油站出发，如果车要能够到达下一个加油站，就需要这个加油站的 $gas > cost$ 。不妨设 $c[i] = gas[i] - cost[i]$ ， $c[i]$ 表示的是从某一加油站得到的汽油减去到达下一个加油站需要耗费的汽油后剩余的汽油数目，对 $c$ 求和得到的是从出发开始到当前加油站剩余的汽油数目，如果这个这个和为负，说明当前这种行驶方案无法到达当前的加油站。也就是说要使车能够不断的向前行进，就要保证途中对 $c$ 的求和始终大于0。

如果 $cost$ 的和大于 $gas$ 的和，显然汽车是无法成功走完一圈的，下面证明如果 $cost$ 的和小于等于 $gas$ 的和，必然存在一种方案能够让汽车走完一圈。

现在有  $c[0] + c[1] + \dots + c[n-2] + c[n-1] \geq 0$ ，我们对 $c$ 的前 $i$ 项求和，假设当 $i=j$ 时，这个和是所有和中最小的，也就是说：

```

c[0]+c[1]+...+c[j-1]<=c[0]+c[1]+...c[j]
c[0]+c[1]+...+c[j-1]<=c[0]+c[1]+...c[j]+c[j+1]
...
c[0]+c[1]+...+c[j-1]<=c[0]+c[1]+...c[j]+c[j+1]+...+c[n-1]

```

也就是说：

```

c[j]>=0
c[j]+c[j+1]>=0
...
c[j]+c[j+1]+...+c[n-1]>=0

```

同时，因为前j项的求和是最小的，还能得到下面的不等式：

```

c[0]+c[1]+...+c[j-1]<=c[0]+c[1]+...+c[j-2]
c[0]+c[1]+...+c[j-1]<=c[0]+c[1]+...+c[j-3]
...
c[0]+c[1]+...+c[j-1]<=c[0]

```

转换可以得到：

```

c[j-1]<=0
c[j-2]+c[j-1]<=0
...
c[1]+c[1]+...+c[j-1]<=0

```

再组合最初始的条件  $c[0]+c[1]+...+c[n-2]+c[n-1]>=0$ ，我们可以得到：

```

c[j]+c[j+1]+...+c[n-1]+c[0]+c[1]+...+c[j-2]>=0
c[j]+c[j+1]+...+c[n-1]+c[0]+c[1]+...+c[j-3]>=0
c[j]+c[j+1]+...+c[n-1]+c[0]>=0

```

至此我们可以看出，如果从j出发的话，对c的求和始终都满足大于等于零的要求，也就是说j是我们选择出发的加油站。



## AC源码

```
class Solution(object):
    def canCompleteCircuit(self, gas, cost):
        """
        :type gas: List[int]
        :type cost: List[int]
        :rtype: int
        """
        if sum(gas) < sum(cost):
            return -1
        min_sum, min_index, total = 0, 0, 0
        for i in range(len(gas)):
            total += gas[i] - cost[i]
            if min_sum > total:
                min_sum, min_index = total, i + 1
        return -1 if total < 0 else min_index

if __name__ == "__main__":
    assert Solution().canCompleteCircuit([5], [4]) == 0
    assert Solution().canCompleteCircuit([5, 1, 2, 3, 4], [4, 4, 1, 5, 1]) == 4
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode解题之Candy

---

### 原题

一直线上站了N个孩子，每个孩子都有一个属于自己的数字，现在按照如下规则给孩子分发糖果：每个孩子至少有一个糖果；相邻的孩子中数字比较大的那个拿的糖果也比较多。求最少要发掉多少个糖果。

注意点：

- 无

例子：

输入: ratings = [1, 2, 3, 2]

输出: 7

### 解题思路

从前往后遍历的时候，我们只考虑升序的序列，对于其中一段升序的序列，最理想的情况是按照1,2,3...这样分发糖果；而对于降序的序列，如果从后往前遍历就也变成升序的了。通过前序和后序遍历后，升序与降序的交接处那个点会有两个值，因为要比两边的孩子拿到的糖果都多，所以取较大的那个值。这时候得到的数组就是在满足题目要求前提下每个孩子拿到的最少的糖果数，返回它的和即可。

### AC源码

```
class Solution(object):
    def candy(self, ratings):
        """
        :type ratings: List[int]
        :rtype: int
        """
        n = len(ratings)
        candy = [1] * n
        for i in range(1, n):
            if ratings[i] > ratings[i - 1]:
                candy[i] = candy[i - 1] + 1
        for i in range(n - 2, -1, -1):
            if ratings[i] > ratings[i + 1]:
                candy[i] = max(candy[i], candy[i + 1] + 1)
        return sum(candy)

if __name__ == "__main__":
    assert Solution().candy([1, 2, 3, 7, 4, 3, 2, 1]) == 21
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode解题之Single Number

---

## 原题

一个数组中除了一个数字出现过一次外，其余的数字都出现了两次，找出那个只出现一次的数字。

注意点：

- 算法时间复杂度要求为 $O(n)$
- 空间复杂度为 $O(1)$

例子：

输入: `nums = [1, 2, 3, 4, 3, 2, 1]`

输出: 4

## 解题思路

非常常见的一道算法题，将所有数字进行异或操作即可。对于异或操作明确以下三点：

- 一个整数与自己异或的结果是0
- 一个整数与0异或的结果是自己
- 异或操作满足交换律，即 $a^b=b^a$

所以对所有数字进行异或操作后剩下的就是那个只出现一次的数字。

## AC源码

```
class Solution(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = nums[0]
        for i in nums[1:]:
            result ^= i
        return result

if __name__ == "__main__":
    assert Solution().singleNumber([1, 2, 3, 4, 3, 2, 1]) == 4
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Single Number II

## 原题

一组数字中，有一个数字只出现一次，其余的都出现了三次，找出只出现了一次的数字。

注意点：

- 线性时间复杂度
- 最好不用额外申请空间

例子:

输入: `nums = [1, 1, 1, 2, 3, 3, 3]`

输出: 2

## 解题思路

如果要不额外申请空间，可以通过三个变量来记录每位出现的次数，注意顺序上应该先算出现三次、两次，再一次的，如果反过来的话会出现重复计算（如计算第一个数字时，`one = one | num -> num`，`two = two | one & num -> num`，`three = two & num -> num`，而此时显然不应该存在有位已经出现两次或三次）。当一个位的出现次数已经到三次了，那么就将这些位抹去。最后剩下的就是那个只出现一次的数。

上面的方法不是很通用，如果除一个数字外其余的数字出现了n次，那上面的方法就不容易解决了。可以用了一个包含32个元素（因为int型数值为32位）的数组来记录每一个位出现的次数，最后对每位对n进行取余操作，并通过位移操作将剩余的数字拼起来。对于Python需要注意的是要考虑负数的情况，因为Python默认数值都是可以无限大的，而不是根据首位是否为0来区分正负数，需要手动分辨一下。

## AC源码

```

class Solution(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        one, two, three = 0, 0, 0
        for num in nums:
            # calculate the count of the each bit
            three = two & num
            two = two | one & num
            one = one | num
            # clear the count for the bit which has achieved thr
ee
            one = one & ~three
            two = two & ~three
        return one

    def singleNumber_normal(self, nums):
        result = 0
        for i in range(32):
            count = 0
            for num in nums:
                count += (num >> i) & 1
            rem = count % 3
            # deal with the negative situation
            if i == 31 and rem:
                result -= 1 << 31
            else:
                result |= rem << i
        return result

if __name__ == "__main__":
    assert Solution().singleNumber([1, 1, 1, 2, 3, 3, 3]) == 2
    assert Solution().singleNumber([-2, -2, 1, 1, -3, 1, -3, -3,
-4, -2]) == -4
    assert Solution().singleNumber_normal([1, 1, 1, 2, 3, 3, 3])
== 2
    assert Solution().singleNumber_normal([-2, -2, 1, 1, -3, 1,

```

```
-3, -3, -4, -2]) == -4
```



欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Copy List with Random Pointer

---

## 原题

一个链表中的每一个节点都有一个额外的随机指针，指向链表中的任意节点或空节点。对这个链表进行深拷贝。

注意点：

- 无

例子：

输入：略

输出：略

## 解题思路

比较容易想到的思路就是先不管新添加的指针，把链表当做普通链表先进行拷贝。因为新添加的指针可以指向任意的节点，所以可以将原节点和拷贝的节点建立映射关系，这样就可以找到任意节点对应的拷贝节点了。然后再次遍历原链表，对任意指针进行复制。

链表的拷贝其实可以看做两个步骤，一个是节点数据的拷贝，另一个是节点关系的拷贝。我们也可以先把所有的节点进行拷贝，并存入字典中。然后遍历链表并拷贝两个指针。因为任意指针可能指向空指针，所以在字典中添加一个空指针项。

## AC源码

```
# Definition for singly-linked list with a random pointer.
class RandomListNode(object):
    def __init__(self, x):
        self.label = x
        self.next = None
        self.random = None

class Solution(object):
    def copyRandomList(self, head):
        """
        :type head: RandomListNode
        :rtype: RandomListNode
        """
        if not head:
            return None
        visited = dict()
        node = head
        while node:
            visited[node] = RandomListNode(node.label)
            node = node.next

        visited[None] = None
        node = head
        while node:
            visited[node].next = visited[node.next]
            visited[node].random = visited[node.random]
            node = node.next
        return visited[head]

if __name__ == "__main__":
    None
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。



## LeetCode 解题之 Word Break

---

### 原题

给定一个目标字符串和一组字符串，判断目标字符串能否拆分成数个字符串，这些字符串都在给定的那组字符串中。

注意点：

- 无

例子：

输入: `s = "leetcode", wordDict = {"leet", "code"}`

输出: `True`

### 解题思路

采用动态规划的方法解决，`dp[i]`表示字符串`s[:i]`能否拆分成符合要求的子字符串。

我们可以看出，如果`s[j:i]`在给定的字符串组中，且`dp[j]`为`True`（即字符串`s[:j]`能够拆分成符合要求的子字符串），那么此时`dp[i]`也就为`True`了。按照这种递推关系，我们就可以判断目标字符串能否成功拆分。

### AC源码

```
class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: Set[str]
        :rtype: bool
        """
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True
        for i in range(n):
            for j in range(i, -1, -1):
                if dp[j] and s[j:i + 1] in wordDict:
                    dp[i + 1] = True
                    break
        return dp[n]

if __name__ == "__main__":
    assert Solution().wordBreak("leetcode", {"leet", "code"}) ==
    True
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Word Break II

## 原题

给定一个目标字符串和一组单词，将目标字符串进行拆分，要求拆分出的部分在那个单词组中，拆分后的单词用空格隔开，给出所有可能的拆分情况。

注意点：

- 无

例子：

输入: `s = "catsanddog"`, `dict = ["cat", "cats", "and", "sand", "dog"]`

输出: `["cats and dog", "cat sand dog"]`

## 解题思路

采用深度优先遍历的策略，我们把字符串分为前后两个部分，如果前半部分在单词组中，那么我们就只要递归拆分它的后半部分。那么怎么将字符串分为前后两个部分呢，我们可以直接依次把单词组中的单词与当前字符串的头部进行比较，如果相同，则递归后半部分。

光采用dfs效率低下，因为会有很多重复的情况，假设字符串为 `abcd...`，且我们的单词组中有 `{a, b, c, ab, bc, abc}`，那么以d开头的字符串就要被拆分四次。为了避免这样的情况，可以用过哈希的方法来缓存之前已经计算出来的结果。我们通过一个字典来缓存字符串和它对应的拆分结果。

## AC源码

```
import collections

class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: Set[str]
        :rtype: List[str]
        """
        dic = collections.defaultdict(list)

        def dfs(s):
            if not s:
                return [None]
            if s in dic:
                return dic[s]
            res = []
            for word in wordDict:
                n = len(word)
                if s[:n] == word:
                    for r in dfs(s[n:]):
                        if r:
                            res.append(word + " " + r)
                        else:
                            res.append(word)
            dic[s] = res
            return res

        return dfs(s)

if __name__ == "__main__":
    assert Solution().wordBreak("catsanddog", {"cat", "cats", "and", "sand", "dog"}) == ['cat sand dog', 'cats and dog']
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。





# LeetCode 解题之 Linked List Cycle

## 原题

判断一个链表中是否存在着一个环，能否在不申请额外空间的前提下完成？

注意点：

- 无

例子：

输入：

```
1->2->3
  |  |
  5<-4
```

输出: True

## 解题思路

沿着链表不断遍历下去，如果遇到空节点就说明该链表不存在环。但如果存在环，这样的遍历就会进入死循环。在环上前进会不断地绕圈子，我们让两个速度不同的指针绕着环前进，那么早晚速度快的那个将追上速度慢的，所以如果速度快的追上了速度慢的，那么该链表就存在环，循环终止。

## AC源码

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

class Solution(object):
    def hasCycle(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Linked List Cycle II

## 原题

如果给定的单向链表中存在环，则返回环起始的位置，否则返回为空。最好不要申请额外的空间。

注意点：

- 不要修改链表

例子:

输入:

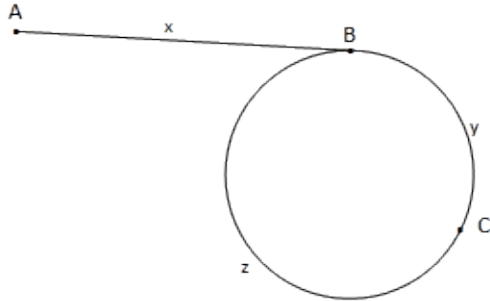
```
1->2->3
  |  |
  5<-4
```

输出: 2

## 解题思路

在[Linked List Cycle](#)中，我们通过双指针方法来判断链表中是否存在环。在此基础上，我们来找出环的起始节点。如下图所示，假设链表的起始节点为A，环的起始节点为B，快慢指针在C处相遇。因为快指针的速度是慢指针的两倍，所以在相同时间内，它走过的路程是慢指针的两倍，而快指针走过的路程是 $(x+y+z+y)$ ，而慢指针走过的路程是 $(x+y)$ ，根据关系我们可以得到  $x+y+z+y = 2(x+y)$ ，也就是说  $x=z$ 。此时快慢指针在C处，头指针在A处，而它们到B的距离相等，那么只要有两个指针分别从点A和点C出以相同的速度前进就会在点B处相遇，也就是找到了环的起始节点。

注：上面的情况是理想情况，实际上在点C相遇时，快指针可能已经绕着环走了好几圈了，如AB很长，而环很小的情况。假设走了 $n$ 圈，此时等式为  $x+y+n(y+z) = 2(x+y)$ ，即  $x=(n-1)(y+z)+z$ ，而从点C绕 $n-1$ 圈后再走 $z$ 的距离还是会跟从点A出发的指针在点B相遇。



## AC源码

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

class Solution(object):
    def detectCycle(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                node = head
                while node != slow:
                    node = node.next
                    slow = slow.next
                return node
        return None

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Reorder List

---

## 原题

将单向链表 $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$ 转化为 $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$ 的形式，也就是从头部取一个节点，从尾部取一个节点，直到将原链表转化成新的链表。

注意点：

- 不要申请额外的空间
- 不要修改节点的数值

例子：

输入: {1,2,3,4}

输出: {1,4,2,3}

## 解题思路

由于是一个单向链表，从尾部不断取元素比较困难，所以我们要将链表反转，又因为是从两头取元素，所以我们只需要反转后半段链表。我们先通过快慢指针来获得链表的中间节点，并将链表截断。接着翻转后半段链表，最后依次从两个链表中提取元素进行连接。需要注意的是，在截断时注意哪个链表比较长，合并的时候不要遗漏元素。

## AC源码

```
class Solution(object):
    def reorderList(self, head):
        """
        :type head: ListNode
        :rtype: void Do not return anything, modify head in-place instead.
        """
        if not head:
            return
        # split
        fast = slow = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        head1, head2 = head, slow.next
        slow.next = None
        # reverse
        cur, pre = head2, None
        while cur:
            nex = cur.next
            cur.next = pre
            pre = cur
            cur = nex
        # merge
        cur1, cur2 = head1, pre
        while cur2:
            nex1, nex2 = cur1.next, cur2.next
            cur1.next = cur2
            cur2.next = nex1
            cur1, cur2 = nex1, nex2

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。





# LeetCode 解题之 Binary Tree Preorder Traversal

## 原题

采用非递归的方法进行二叉树的前序遍历。

注意点：

- 无

例子：

输入：

```
  1
   \
    2
   /
  3
```

输出: [1,2,3]

## 解题思路

二叉树进行前序遍历时，首先访问根结点然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树。可以看出采用递归的方法非常简单，如果不用递归，我们可以通过一个栈来辅助遍历。在先序遍历中，访问完根节点，我们接着遍历它的左子树，它的右子树要等左子树遍历完成后再遍历，所以我们先把它存起来。而左子树的头节点（区别于根节点）也会有它的右子树，这棵右子树需要比之前的右子树先遍历（因为它是根节点的左子树中的），所以存储采用栈的结构。当遍历到某一个节点没有左子树后，我们从栈中取出右子树节点继续遍历，直到遍历完整棵树。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        stack = []
        result = []
        while root or stack:
            if not root:
                root = stack.pop()
                result.append(root.val)
            if root.right:
                stack.append(root.right)
            root = root.left
        return result

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Binary Tree Postorder Traversal

## 原题

采用非递归的方法进行二叉树的后序遍历。

注意点：

- 无

例子：

输入：

```
  1
   \
    2
   /
  3
```

输出: [3,2,1]

## 解题思路

二叉树进行后序遍历时，先后序遍历左子树，再后序遍历右子树，最后访问该节点。也就是说第一次遍历到一个节点的时候，我们不将其加入到结果中，只有当它的左右子树都遍历完后，我们将该节点加入到结果中。跟先序遍历中一样，我们也通过栈来解决，把接下去要访问的节点压入栈中。由于现在每个节点都要遍历两次，我们给节点添加一个标志位，如果一个节点还没有访问过，我们给的标志为 `visit`，表示下一次遇到它只是第一次访问它，在访问它之后，我们把它的标志改为 `get` 并再次压栈，表示下一次遇到它要访问它的值。同时还要将它的右子树和左子树分别压栈，表示要后续遍历左子树和右子树。对于第二次访问的节点，将其加入结果中。

## AC源码

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    def postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        if not root:
            return []
        result = []
        stack = [(root, 'visit')]
        while stack:
            node, label = stack.pop()
            if label == 'visit':
                stack.append((node, 'get'))
                if node.right:
                    stack.append((node.right, 'visit'))
                if node.left:
                    stack.append((node.left, 'visit'))
            else:
                result.append(node.val)
        return result

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode解题之LRU Cache

## 原题

设计并实现一个支持近期最少使用缓存的数据结构。要支持存和取两个操作。根据键取值时，如果键对应的值不存在，则返回-1。存键值对时，如果容量已经满了，要把最近最少使用的键值对去除后再插入。

注意点：

- 无

例子：

无

## 解题思路

首先明确最少使用缓存是指我们要缓存最近使用的数据，如果一个数据长时间没有使用，且又有新的数据加入，那么应该将最长时间没有使用的数据去除。为此我们可以通过一个双向链表完成这样的数据结构，表头表示最近使用过的数据，越接近表尾表示越久没有使用过。当要将旧数据删除时，我们只需要将链表尾部的节点去除，并在头部插入新的节点。而更新节点时，我们只需要将原来的节点删除，改变节点的内容，再插入到链表头部。而最简单的插入，即还没有达到容量上限时，我们只要头部直接插入。

我们知道链表的查找操作速度较慢，为了提高查找的速度，我们可以通过一个键值对的字典来记录数据。查找时先判断是否在字典中，如果在则需要更新节点的使用情况并返回结果，如果不在则直接返回-1。这样插入数据时也可以通过字典来判断是首次插入还是数据的更新。

## AC源码

```
class LRUCache(object):
```

```

class Node(object):
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev, self.next = None, None

def __init__(self, capacity):
    """
    :type capacity: int
    """
    self.capacity, self.size = capacity, 0
    self.dic = {}
    self.head, self.tail = self.Node(-1, -1), self.Node(-1,
-1)
    self.head.next, self.tail.prev = self.tail, self.head

def __remove(self, node):
    node.prev.next = node.next
    node.next.prev = node.prev
    node.prev, node.next = None, None

def __insert(self, node):
    node.prev, node.next = self.head, self.head.next
    self.head.next.prev = node
    self.head.next = node

def get(self, key):
    """
    :rtype: int
    """
    if key not in self.dic:
        return -1
    node = self.dic[key]
    self.__remove(node)
    self.__insert(node)
    return node.value

def set(self, key, value):
    """
    :type key: int

```

```
:type value: int
:rtype: nothing
"""
    if key in self.dic:
        node = self.dic[key]
        self.__remove(node)
        node.value = value
        self.__insert(node)
    else:
        if self.size == self.capacity:
            discard = self.tail.prev
            self.__remove(discard)
            del self.dic[discard.key]
            self.size -= 1
        node = self.Node(key, value)
        self.dic[key] = node
        self.__insert(node)
        self.size += 1

if __name__ == "__main__":
    lru_cache = LRUCache(3)
    lru_cache.set(1, 1)
    lru_cache.set(2, 2)
    lru_cache.set(3, 3)
    assert lru_cache.get(0) == -1
    assert lru_cache.get(1) == 1
    lru_cache.set(1, 10)
    assert lru_cache.get(1) == 10
    lru_cache.set(4, 4)
    assert lru_cache.get(2) == -1
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode解题之Insertion Sort List

---

## 原题

通过插入排序的方法排序一个链表。

注意点：

- 无

例子：

无

## 解题思路

数组的插入排序很简单，将元素依次放入已经排好序的数组中的正确位置。链表与数组的操作稍微有些不同，数组是将比要插入的元素大的数值往后移，直到遇到比该元素小的值，就把该元素放在比它小的值后面。但单链表只能从头开始判断新的节点该插入到哪里。链表也可以根据尾节点进行优化，如果要插入的节点比尾节点还大的话，就不用从头开始找插入的位置了。插入时要注意链表插入操作，Python写链表相关操作还是很方便的。

## AC源码

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

class Solution(object):
    def insertionSortList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(-1)
        cur = dummy
        while head:
            # check if it is needed to reset the cur pointer
            if cur and cur.val > head.val:
                cur = dummy
            # find the place to insert
            while cur.next and cur.next.val < head.val:
                cur = cur.next
            # insert and sort the next element
            cur.next, cur.next.next, head = head, cur.next, head
            head = head.next
        return dummy.next

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之Max Points on a Line

## 原题

在一个平面上有 $n$ 个点，求一条直线最多能够经过多少个这些点。

注意点：

- 无

例子：

无

## 解题思路

不同的两点能够确定一条直线，所以要将这些点两两组合来得到相应的直线。而要判断其他的点是否在该直线上，只需要计算要判断的点与两点中的一个点A计算出的斜率是否与该直线的斜率相同。需要注意的是如果点A和要判断的点的横坐标相同，那么可能这两个点是同一个或者斜率不存在。

有些情况可以不需要重复计算，我们可以给点规定一个顺序。例如点A、B、C、D都在一直线上，当我们用A和B组成一条直线时，我们可以判断出B和C也在该直线上，此时直线上有4个点。而当点B和点C组合时，其实还是之前已经检测过的那条直线，所以之前的点A没必要重复检测了。而后面的点要继续检测，因为BC可能是一条全新的直线，之后的点可能出现在该直线上。但B之前的点是不可能出现在该直线上的，否则B和那个点就已经组成过该直线，这与假设该直线是全新的直线相矛盾。

## AC源码

```
# Definition for a point.
class Point(object):
    def __init__(self, a=0, b=0):
```

```
self.x = a
self.y = b

class Solution(object):
    def maxPoints(self, points):
        """
        :type points: List[Point]
        :rtype: int
        """
        n = len(points)
        slope_map = {}
        result = 0
        for i in range(n):
            slope_map.clear()
            same, vertical = 1, 0
            slope_max = 0
            for j in range(i + 1, n):
                dx, dy = points[i].x - points[j].x, points[i].y
                - points[j].y

                if dx == dy == 0:
                    same += 1
                elif dx == 0:
                    vertical += 1
                else:
                    slope = float(dy) / float(dx)
                    slope_map[slope] = slope_map.get(slope, 0) +
1
                    slope_max = max(slope_max, slope_map[slope])
            result = max(result, max(slope_max, vertical) + same
        )

        return result

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



# LeetCode 解题之 Evaluate Reverse Polish Notation

## 原题

对表达式的后缀形式（也称为逆波兰表达式）进行计算并返回结果。操作符只有加、减、乘、除四种，操作数为一个整数或者一个表达式。

注意点：

- 无

例子：

输入: `tokens = ["2", "1", "+", "3", "*"]`

输出: 9

## 解题思路

后缀表达式的形式为 操作数1，操作数2，操作符，也就是操作符要进行计算操作的两个数（或者表达式）在它的前方，所以在遍历列表的时候，我们要将前面的操作符压入栈中，当遇到操作符的时候，我们将它对应的操作数弹出并进行计算，计算结果可能是其他操作数的操作数，它原来是一个表达式，我们将该表达式的值计算出来了，所以应该把那个值继续压栈，遍历完整个列表的时候，计算结束。这里特别要注意的是除法操作，因为给的表达式都是合法的，所以不用考虑除数为零的情况，但这里的除法操作是针对整数的，会对结果进行去尾操作。对负数与整数的除法操作也与Python自带的计算方式不同，Python计算 `-1//2` 结果为-1，而在这里应该为0，所以要进行特殊的处理。

## AC源码

```
class Solution(object):
    def evalRPN(self, tokens):
        """
        :type tokens: List[str]
        :rtype: int
        """
        stack = []
        for token in tokens:
            if token not in ("+", "-", "*", "/"):
                stack.append(int(token))
            else:
                second = stack.pop()
                first = stack.pop()
                if token == "+":
                    stack.append(first + second)
                elif token == "-":
                    stack.append(first - second)
                elif token == "*":
                    stack.append(first * second)
                else:
                    if first * second < 0:
                        stack.append(-(abs(first) // abs(second)))
                    else:
                        stack.append(first // second)
        return stack.pop()

if __name__ == "__main__":
    assert Solution().evalRPN(["2", "1", "+", "3", "*"]) == 9
    assert Solution().evalRPN(["4", "13", "5", "/", "+"]) == 6
    assert Solution().evalRPN(["10", "6", "9", "3", "+", "-11",
                                "*", "/", "*", "17", "+", "5", "+"]) == 22
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。





## LeetCode解题之Reverse Words in a String

---

### 原题

将一个字符串依照单词的力度进行反转。例如将"the sky is blue"转化为"blue is sky the"。

注意点：

- 原始字符串中可能首位有空格，结果不要有这些空格
- 原始字符串单词间可能有多个空格，结果单词间只需有一个空格

例子：

输入: s = "the sky is blue"

输出: "blue is sky the"

### 解题思路

用Python解决这个问题有些特殊，Python中的字符串是不可变的数据类型，而比较pythonic的写法是直接一行代码，先把字符串按空格分开，反转后用空格连接起来。

从算法层面讲，这道题希望做到原地完成字符串的反转。我们可以通过以下几步做到：

- 将整个字符串直接反转
- 遍历字符串单独将每个单词反转，注意反转的同时可以通过移位来除去不必要的空格

### AC源码

```
class Solution(object):
    def reverseWords(self, s):
        """
        :type s: str
        :rtype: str
        """
        return " ".join(s.split()[::-1])

if __name__ == "__main__":
    assert Solution().reverseWords("the sky is blue ") == "blue
is sky the"
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Maximum Product Subarray

---

## 原题

在一个数组中找出一个子数组，使得子数组中的数的乘积最大。

注意点：

- 数字可能为负数
- 给定的数组不为空

例子：

输入: `nums = [2,3,-2,4]`

输出: 6

## 解题思路

比较典型的动态规划题目，需要注意负数乘以负数为正数，所以要同时记录最大局部最优解和最小局部最优解。递推关系式为：

```
temp = positive
positive = max(num, positive * num, negative * num)
negative = min(num, temp * num, negative * num)
```

变量命名有点问题，`positive`指局部最大乘积（不一定是正数），`negative`指局部最小乘积（也不一定是负数）。

## AC源码

```
class Solution(object):
    def maxProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        positive, negative = nums[0], nums[0]
        result = nums[0]
        for num in nums[1:]:
            positive, negative = max(num, positive * num, negative * num), min(num,
                                         positive * num, negative * num)
            result = max(result, positive)
        return result

if __name__ == "__main__":
    assert Solution().maxProduct([2, 3, -2, 4]) == 6
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Min Stack

## 原题

实现一个栈，这个栈除了普通的压栈、弹出、获取栈顶元素外，还要能够在获得栈中的最小元素，且这些操作的时间复杂度为 $O(1)$ 。

注意点：

- 这里的弹出操作只需要去除栈顶元素，没有返回值

例子：

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();    --> Returns -3.
minStack.pop();
minStack.top();        --> Returns 0.
minStack.getMin();    --> Returns -2.
```

## 解题思路

因为仍要有普通栈的功能，所以我们可以内部用一个列表来实现普通的栈，但又需要在常量时间内返回栈内的最小元素，所以要用另外的数据结构来存当前的最小元素。栈中的最小元素会在压栈和弹出两个操作中发生改变，如果仅用一个变量来表示当前栈中的最小元素，压栈时更新该变量没有问题，但弹出时，如果弹出的就是最小的元素，那剩下栈中最小的元素需要 $O(n)$ 的时间重新找出，不符合题目要求。我们可以通过另外一个栈来存储当前栈中的最小元素。在压栈时，如果最小栈的栈顶元素大于等于压入的元素，那么要对最小栈也进行压栈操作。而在弹出时，如果栈中弹出的元素与最小栈栈顶的元素相等，那我们也要对最小栈进行弹出，最小栈在弹出后，栈顶元素仍然是当前栈中最小的元素。

## AC源码

```
class MinStack(object):
    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.minStack = []

    def push(self, x):
        """
        :type x: int
        :rtype: nothing
        """
        self.stack.append(x)
        if not self.minStack:
            self.minStack.append(x)
        else:
            if self.minStack[-1] >= x:
                self.minStack.append(x)

    def pop(self):
        """
        :rtype: nothing
        """
        if self.stack:
            if self.minStack[-1] == self.stack[-1]:
                self.minStack.pop()
            self.stack.pop()

    def top(self):
        """
        :rtype: int
        """
        if self.stack:
            return self.stack[-1]

    def getMin(self):
```

```
        """
        :rtype: int
        """
        if self.minStack:
            return self.minStack[-1]

if __name__ == "__main__":
    minStack = MinStack()
    minStack.push(4)
    minStack.push(5)
    minStack.push(1)
    minStack.push(3)
    assert minStack.getMin() == 1
    minStack.pop()
    minStack.pop()
    assert minStack.top() == 5
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Intersection of Two Linked Lists

## 原题

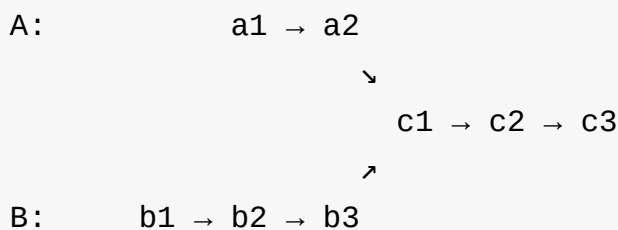
找出两个单向链表是在哪个节点开始合二为一的。

注意点：

- 如果没有交集，那么返回None
- 返回结果时要保证链表还是原来的结构
- 链表中没有环形结构
- 最好时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$

例子：

输入：



输出：c1

## 解题思路

最容易想到的方法是双指针，在交点之前，两个链表的长度不同，如果我们知道他们的长度差，那么只要在长的链表上先进他们相差的节点数，然后两个指针同时各自在两个链表上前进，那么当他们相遇的时候就是第一个相交的节点。而那个差就是两个链表的长度差，因为它们的后半部分是相同的。



上面是通过让长链表上的指针先走的方法，其实我们也可以在链表前加长，使得它们在交点之前的长度相等，在上面的例子中，我们可以在链表A之前加上一个链表B，在链表B之前加上一个链表A，这时两条链表的总长度相同，都是原来两条链表的长度和，那么在它们交点前的链表长度也是相等的。让两个指针同时前进直到相交即可。

还可以通过复用[Linked List Cycle II](#)的代码来实现，我们将链表A的尾部连接到它的头部，这道题就变成找出链表中环的起始位置了。不过要记得将链表恢复原样。

## AC源码

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

class Solution(object):
    def getIntersectionNode(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """
        nodeA, nodeB = headA, headB
        while nodeA != nodeB:
            nodeA = nodeA.next if nodeA else headB
            nodeB = nodeB.next if nodeB else headA
        return nodeA

    def getIntersectionNode_diff(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """

        def get_length(node):
            length = 0
```

```
        while node:
            node = node.next
            length += 1
        return length

len1 = get_length(headA)
len2 = get_length(headB)
if len1 > len2:
    for __ in range(len1 - len2):
        headA = headA.next
else:
    for __ in range(len2 - len1):
        headB = headB.next
while headA:
    if headA == headB:
        return headA
    headA = headA.next
    headB = headB.next
return None

if __name__ == "__main__":
    None
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Find Peak Element

## 原题

比左右两个元素大的元素我们称为顶点元素，在给定的一个数组中，没有连续的两个元素是相等的。找出这个数组中某一个顶点元素的下标，如果有多个，给出其中任意一个即可。可以默认在给定的数组两端还各有一个无穷小的元素，即数组[1]的顶点元素下标为0。

注意点：

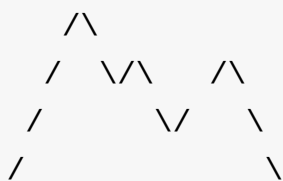
- 将时间复杂度控制为 $\log(n)$

例子：

输入: `nums = [1, 2, 3, 1]`

输出: 2

## 解题思路



从上图可以看出，一条上升的边和一条下降的边之间至少夹着一个顶点元素。由于左右两边各有一个无穷小的元素，所以起始的时候最左边的边是上升的，最右边的边是下降的。要求时间复杂度为 $\log(n)$ ，我们可以通过二分搜索来判断。我们取中点和它后面的一个点，如果这两个点构成的边是上升的，那我们就把中点左边的点抛弃掉，这时候仍然满足最左边的边是上升的，最右边的边是下降的；如果两个点构成的边是下降的，那么该把中点右边的点抛弃掉，这样仍然满足上面的要求，保证左右两个点之间至少有一个顶点元素。至于如何在两点中选择新的左右节点，我

们要尽可能使新的左右节点靠近顶点元素，因为最终的终止条件是左右节点重合。所以选择新的左节点时，应该选中点的后一个节点，而选新的右节点时，选择中点。

## AC源码

```
class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (right + left) // 2
            if nums[mid] < nums[mid + 1]:
                left = mid + 1
            else:
                right = mid
        return left

if __name__ == "__main__":
    assert Solution().findPeakElement([1, 2, 3, 1]) == 2
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Fraction to Recurring Decimal

---

## 原题

将一个分数用小数的形式表示，如果小数部分存在循环，将循环的部分放在圆括号内来表示无限循环。

注意点：

- 无

例子：

输入: numerator = 1, denominator = 2

输出: "0.5"

输入: numerator = 2, denominator = 3

输出: "0.(6)"

## 解题思路

能够除尽的很简单，就是模拟一个除法操作，对于无限循环小数，如果哪一次的余数在之前已经出现过了，那么继续算下去就会出现循环，这时候就可以停止运算，并将这两位之间的数添加到括号中。整数部分不存在循环问题，所以可以先单独计算，符号也可以先确定。因为是判断是否有重复的余数，对于能够除尽的情况，重复的余数是除尽后的0，所以末尾会多个"(0)"，要去掉。而能够整除的情况，要把小数点也去掉。

## AC源码

```

class Solution(object):
    def fractionToDecimal(self, numerator, denominator):
        """
        :type numerator: int
        :type denominator: int
        :rtype: str
        """
        sign = '-' if numerator * denominator < 0 else ''
        quotient, remainder = divmod(abs(numerator), abs(denominator))
        result_list = [sign, str(quotient), '.']
        remainders = []
        while remainder not in remainders:
            remainders.append(remainder)
            quotient, remainder = divmod(remainder * 10, abs(denominator))
            result_list.append(str(quotient))

            idx = remainders.index(remainder)
            result_list.insert(idx + 3, '(')
            result_list.append(')')
            result = ''.join(result_list).replace('(0)', '').rstrip('.')

        return result

if __name__ == "__main__":
    assert Solution().fractionToDecimal(1, 2) == '0.5'
    assert Solution().fractionToDecimal(2, 1) == '2'
    assert Solution().fractionToDecimal(2, 3) == '0.(6)'

```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Excel Sheet Column Title

---

## 原题

在Excel中，列名的表示形式为A,B,C...AA,AB...，给定一个正整数，将其转换为对应的列名。

注意点：

- 无

例子:

输入:  $n = 1$

输出: 'A'

输入:  $n = 28$

输出: 'AB'

## 解题思路

比较简单的一道题，相当于进制转换，转换的基数为26。注意一下高地位的问题，后转换出来的字母应该放在前面，还有所有字母都是大写。

## AC源码

```
class Solution(object):
    def convertToTitle(self, n):
        """
        :type n: int
        :rtype: str
        """
        result = []
        base = ord('A')
        while n:
            n, r = divmod(n - 1, 26)
            result.append(chr(base + r))
        return ''.join(result[::-1])

if __name__ == "__main__":
    assert Solution().convertToTitle(1) == 'A'
    assert Solution().convertToTitle(28) == 'AB'
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。



## LeetCode解题之Majority Element

---

### 原题

给定一个非空的数组，其中某一数值的数量超过数组长度的一半，请找出这个值。

注意点：

- 无

例子：

输入: `nums = [1, 2, 2, 3, 3, 3, 3]`

输出: 3

### 解题思路

可以通过哈希表记录每个数值出现的次数并找出出现次数最多的数值。不过有一个比较巧妙的思路是，就是不断将数组中两两不同的数同时去除，极端情况是每个其他数值都与一个数量最多的数同时去除，即使这样，最后剩下的数值也是那个数量超过总数一半的数值。具体实现来看，我们可以把需要抵消的数值存起来，并记录它的数量，这个数量表示它还能抵消掉几个不同的数值。遇到与它不同的数值就减一，遇到相同的就加一。如果数量变为0，那说明这个数暂时已经完全去除了。我们可以用下一个数值继续与其它数值相抵消。遍历完整个数组后，还没抵消掉的数值就是要求的结果。

### AC源码

```
class Solution(object):
    def majorityElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = None
        count = 0
        for num in nums:
            if count == 0:
                result = num
            if result == num:
                count += 1
            else:
                count -= 1
        return result

if __name__ == "__main__":
    assert Solution().majorityElement([1, 2, 2, 3, 3, 3, 3]) == 3

    assert Solution().majorityElement([3, 3, 3, 3, 1, 1, 2]) == 3
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Excel Sheet Column Number

---

## 原题

在Excel中，列名的表示形式为A,B,C...AA,AB...，给定一个Excel的列名，将其转化为数字，其中A表示1，其后以此类推。

注意点：

- 无

例子:

输入: `s = 'A'`

输出: 1

输入: `s = 'AB'`

输出: 28

## 解题思路

相当于将一个二十六进制的数字转换为十进制，不过这里的二十六进制比较特殊，不是由1,2,3...A,B,...Q这些数字表示，而是A,B,C...Z来表示。但原理是一样的，同一个字母往左移一位，它表示的数字就变为原来的26倍。将每个字母根据它所在的位置计算出它表示的值即可。

## AC源码

```
class Solution(object):
    def titleToNumber(self, s):
        """
        :type s: str
        :rtype: int
        """
        base = ord('A') - 1
        n = len(s)
        result = 0
        for i in range(n):
            result += (ord(s[n - 1 - i]) - base) * pow(26, i)
        return result

if __name__ == "__main__":
    assert Solution().titleToNumber('A') == 1
    assert Solution().titleToNumber('AB') == 28
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Factorial Trailing Zeroes

---

## 原题

求 $n$ 的阶乘末尾有几个零。

注意点：

- 将时间复杂度控制为 $\log(n)$

例子：

输入:  $n = 5$

输出: 1

## 解题思路

通过因数分解知道，10是由2和5相乘得到的，而在 $n$ 的阶乘中，因子2的数目总是比5多的，所以最终末尾有几个零取决于其中有几个5。1到 $n$ 中能够整除5的数中有一个5，能整除25的数有2个5（且其中一个在整除5中已经计算过）...所以只要将 $n$ 不断除以5后的结果相加，就可以得到因子中所有5的数目，也就得到了最终末尾零的数目。

## AC源码

```
class Solution(object):
    def trailingZeroes(self, n):
        """
        :type n: int
        :rtype: int
        """
        count = 0
        while n:
            n //= 5
            count += n
        return count

if __name__ == "__main__":
    assert Solution().trailingZeroes(25) == 6
```

欢迎查看我的[Github \(https://github.com/gavinfish/LeetCode-Python\)](https://github.com/gavinfish/LeetCode-Python) 来获得相关源码。

# LeetCode 解题之 Binary Search Tree Iterator

## 原题

在一棵二叉搜索树上实现一个迭代器，初始化参数为该二叉搜索树的根节点。当调用迭代器的 `next()` 方法时，返回该二叉搜索树中最小的元素。

注意点：

- `next()` 和 `hasNext()` 操作的平均时间复杂度为  $O(1)$ ，空间复杂度为  $O(n)$ ， $n$  是二叉树的高度

例子：

- 无

## 解题思路

要找到二叉搜索树中的最小节点，应该从根节点递归遍历左节点，直到遍历的节点没有左节点，那么该节点就是二叉树中的最小节点。现在已经有二叉搜索树中没有访问过的最小节点了，那么当访问了该节点后，剩余没有访问的树中最小的节点在哪里呢？如果该节点有右子树，那么在它的右子树中（又回到了找一棵二叉搜索树的最小元素，不过这棵二叉搜索树变小了）；如果没有右子树，那么就是它的父节点。为了能够快速定位到父节点，我们可以用栈将遍历路径暂存起来，当进行 `next()` 操作时，我们弹出栈顶元素并进行访问，如果它有右子树的话就遍历它的右子树；如果没有右子树，当下次出栈操作时就是访问当前节点的父节点了。

`hasNext()`和`next()`要连用，如`i.hasNext(): v.append(i.next())`，否则会抛出出栈异常，测试用例提供了这项保证

## AC源码

```
# Definition for a binary tree node
# class TreeNode(object):
```

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

class BSTIterator(object):
    def __init__(self, root):
        """
        :type root: TreeNode
        """
        self.stack = []
        self._pushLeft(root)

    def hasNext(self):
        """
        :rtype: bool
        """
        return self.stack

    def next(self):
        """
        :rtype: int
        """
        node = self.stack.pop()
        self._pushLeft(node.right)
        return node.val

    def _pushLeft(self, node):
        while node:
            self.stack.append(node)
            node = node.left

# Your BSTIterator will be called like this:
# i, v = BSTIterator(root), []
# while i.hasNext(): v.append(i.next())

if __name__ == "__main__":
    None
```



欢迎查看我的 [Github](https://github.com/gavinfish/LeetCode-Python) (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

## LeetCode 解题之Dungeon Game

### 原题

公主被困在地牢（可以理解为矩阵或者二维数组）的右下角，骑士从左上角出发去救公主。骑士拥有一定的初始血量，如果途中他的血量少于等于0，那么营救失败。地牢中的每个格子都有一个数字，负数表示骑士收到了伤害，正数（可能为0）表示骑士回复了一定的血量。现在规定骑士只能向右或者向下前进，求骑士的初始血量至少为多少才能救出公主。

注意点：

- 骑士的血量没有上限规定
- 每格地牢都有数字，包括起点和终点。

例子:

输入:

-2 (K)		-3		3
-5		-10		1
10		30		-5 (P)

输出: 7 (右->右->下->下)

### 解题思路

一看就是典型的动态规划问题，不过这道题从左上角开始递推不好推，因为要考虑两个因素，一是剩余的血越多越好（血量多可以使得以后的路程中血量更加高一点），行进过程中血量最低值越高越好（这个直接影响到结果），这两个因素都要考虑，且相互影响。其实我们可以从右下角开始推，dp表示如果从该格子出发，最少要多少初始血量。此时的递归关系为：

```
dp[j][i] = max(min(dp[j + 1][i], dp[j][i + 1]) - dungeon[j][i],
1)
```

我们来梳理一下这个关系，在(j,i)位置出发最少的初始血量  $dp[j][i]$ ，而从它右边的格子出发最少的初始血量为  $dp[j][i+1]$ ，从下边的格子出发的最少血量为  $dp[j+1][i]$ 。为了使  $dp[j][i]$  尽可能小，我们要使得  $dp[j][i] + \text{dungeon}[j][i] = \min(dp[j + 1][i], dp[j][i + 1])$ 。同时  $dp[j][i]$  又必须大于0。

## AC源码

```
class Solution(object):
    def calculateMinimumHP(self, dungeon):
        """
        :type dungeon: List[List[int]]
        :rtype: int
        """
        n = len(dungeon)
        m = len(dungeon[0])
        dp = [[0 for __ in range(m)] for __ in range(n)]
        dp[-1][-1] = 1 if dungeon[-1][-1] > 0 else 1 - dungeon[-1]
        ][-1]
        for i in range(m - 2, -1, -1):
            dp[-1][i] = max(1, dp[-1][i + 1] - dungeon[-1][i])
        for j in range(n - 2, -1, -1):
            dp[j][-1] = max(1, dp[j + 1][-1] - dungeon[j][-1])
        for j in range(n - 2, -1, -1):
            for i in range(m - 2, -1, -1):
                dp[j][i] = max(min(dp[j + 1][i], dp[j][i + 1]) -
dungeon[j][i], 1)
            return dp[0][0]

if __name__ == "__main__":
    assert Solution().calculateMinimumHP([[-2, -3, 3], [-5, -10,
1], [10, 30, -5]]) == 7
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode 解题之 Largest Number

---

## 原题

给定一组非负整数，将这些整数拼接成最大的数字。由于返回的数字可能非常大，直接返回字符串。

注意点：

- 无

例子：

输入: `nums = [3, 30, 34, 5, 9]`

输出: `'9534330'`

## 解题思路

要将这些数字拼接起来，其实就是要规定好这些数字出现的先后顺序，所以我们要对这些数字进行排序。直接比较数字的大小是行不通的，如12和121，虽然 $121 > 12$ ，但是 $12121 > 12112$ 。应有的排序规则应该是如果 $AB > BA$ ，那么我们规定 $A > B$ ，也就是最后拼接时A放在B之前。这个方案看着有点道理，而且也是正确的，但要证明按照这种排序得到的数字是最大的比较困难。

以下证明参考自[@19thhell](#)。主要要说名两点，一是这种比较关系具有传递性，二是按照这样的拼接顺序得到的数是最大的。

首先，通过证明传递性可以帮我得到所有数的大小关系：

为了方便表示，定义  $f(X) = 10^{(\lg X + 1)}$ ， $(\lg X + 1)$  表示  $X$  有多少位，所以  $AB = f(B)A + B$

假设  $A < \cdot B$ ,  $B < \cdot C$  ( $< \cdot$  是为了与一般的小于号区分)

即  $AB < BA$ ,  $BC < CB$

因为  $AB < BA$

$$f(B)A + B < f(A)B + A$$

$$(f(B) - 1)A < (f(A) - 1)B$$

$$A < B \cdot (f(A) - 1) / (f(B) - 1) \quad (1)$$

因为  $BC < CB$

$$f(C)B + C < f(B)C + B$$

$$(f(C) - 1)B < (f(B) - 1)C$$

$$B < C \cdot (f(B) - 1) / (f(C) - 1) \quad (2)$$

结合 (1), (2),

$$A < C \cdot (f(A) - 1) / (f(C) - 1)$$

$$(f(C) - 1)A < (f(A) - 1)C$$

$$f(C)A + C < f(A)C + A$$

$$AC < CA$$

即得到了  $A < \cdot C$

根据传递性可以得到按照上面的比较规则排序后的数字满足  $A1 \cdot > A2 \cdot > \dots \cdot > A(n-1)$

下面证明  $A1A2\dots A(n-1)$  是这些数能够拼接成的最大的数字。

如果  $A < B$ ，那么我们容易得到  $CAD < CBD$  (这里是一般的大小关系) (3)

因为  $A1 \cdot > A2 \cdot > \dots \cdot > A(n-1)$

所以  $A1A_j > A_jA1$ ，再根据 (3) 有以下的大小关系

$$(A1B)C\dots N > (BA1)C\dots N$$

$$B(A1C)\dots N > B(CA1)\dots N \rightarrow A1BC\dots N > BCA1\dots N$$

...

$$\dots \rightarrow A1BC\dots N > BCA\dots NA1$$

也就是说  $A1$  放在第一个位置相对于把它放在其他的位置时，拼接出来的数字大。同理我们也可以推断出  $A2$  应该放在第二位.....

综上所述，我们通过将数字按照上述方法排序后再拼接就可以得到最大的数字。还有需要注意的是可能参数是多个0，此时字符串拼接的结果是一串0，不符合常规的表达方法，要将其改为0。

## AC源码

```
from functools import cmp_to_key

class Solution:
    # @param {integer[]} nums
    # @return {string}
    def largestNumber(self, nums):
        sorted_nums = sorted(map(str, nums), key=cmp_to_key(lambda x, y: int(y + x) - int(x + y)))
        result = ''.join(sorted_nums).lstrip('0')
        return result or '0'

if __name__ == "__main__":
    assert Solution().largestNumber([3, 30, 34, 5, 9]) == '9534330'
    assert Solution().largestNumber([0, 0]) == '0'
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。

# LeetCode解题之Rotate Array

---

## 原题

将一个数组中的数字右旋 $k$ 位，即所有的数字向后移 $k$ 位，末尾的数字移到开头。

注意点：

- 使用尽可能多的方法来解决
- 提供一种只需要 $O(1)$ 空间的解法

例子：

输入: `nums = [1, 2, 3, 4, 5, 6, 7]`, `k = 3`

输出: `[5, 6, 7, 1, 2, 3, 4]`

## 解题思路

解法一，记下最后一个数字，其他的数字向后移一位，最后把记下的数字放在开头，如此进行 $k$ 次。这个解法空间复杂度为 $O(1)$ ，时间复杂度 $O(kn)$ 。

解法二，申请一个等大的数组，将移位后的结果存在新申请的数组中。这个解法空间复杂度为 $O(n)$ ，时间复杂度为 $O(n)$ 。

解法三，右旋 $k$ 个数字有个等价操作，就是将前 $k$ 个数字翻转，将剩下的数字也翻转，最后将整个数组翻转。此时数组与右旋 $k$ 个数字的结果相同。这个解法空间复杂度为 $O(1)$ ，时间复杂度为 $O(n)$ 。

注意， $k$ 可能比数组的长度 $n$ 还大，我们可以将 $k$ 对 $n$ 取余数，因为右旋 $n$ 个数字相当于没有变化，可以减少计算量。

## AC源码



```
class Solution(object):
    def rotate(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: void Do not return anything, modify nums in-place instead.
        """

        def reverse(nums, start, end):
            while start < end:
                nums[start], nums[end] = nums[end], nums[start]
                start += 1
                end -= 1

        n = len(nums)
        k %= n
        reverse(nums, 0, n - k - 1)
        reverse(nums, n - k, n - 1)
        reverse(nums, 0, n - 1)

if __name__ == "__main__":
    nums = [1, 2, 3, 4, 5, 6, 7]
    Solution().rotate(nums, 10)
    assert nums == [5, 6, 7, 1, 2, 3, 4]
```

欢迎查看我的Github (<https://github.com/gavinfish/LeetCode-Python>) 来获得相关源码。