LeetCode算法题解

【题目】如何反转一个单链表
【解决思路】
难度不大，直接反转链表即可
【PYTHON实现】

```
class Solution(object):
    def reverse_linek_list(self, head):
        cur, prev = head, None
            while cur:
                cur.next = prev
                prev = cur
                cur = cur.next
        return prev
```

【题目】两两交换链表中的节点
给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。
你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。
【解题思路】
直接交换即可
【PYTHON实现】

```
def swapPairs(self, head):
        pHead = ListNode(Node)
        pre, pre.next = pHead, head
        while pre.next and pre.next.next:
            a = pre.next
            b = a. next
            pre.next, b.next, a.next = b, a, b.next
            pre = 1
        return self.next
```

【题目】判断链表是否有环
【解决思路】
使用快慢指针
【PYTHON实现】

```
class Solution(object):
    def linek_list_cycle(self, head):
        fast = slow = head
        while fast and slow and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow in fast:
                return True
        return False
```

【题目】判断括号字符串是否有效
有效字符串需满足：
1. 括号必须用相同类型的右括号闭合
2. 左括号必须以正确的顺序闭合
3. 注意空字符可被认为是有效字符串

【解决思路】
使用栈来实现
【PYTHON实现】

```
def isVaild(self, s):
    stack = []
    map = {']':'[','}':'{',')':'('}
    for c in s:
        if c not in s:
            stack.append(c)
        elif not stack or map[c] != stack.pop()
            return False
        return not stack
```

【题目】用队列实现栈
【解决思路】
使用两个队列来实现即可
【PYTHON实现】

```
import queue
class MyStack(object):
    def __init__(self):
        self.q1 = queue.Queue()

    def push(self, x):
        q2 = queue.Queue()
        q2.put(x)
        while not self.q1.empty():
            q2.put(self.q1.get())
        self.q1 = q2

    def pop(self):
        retrun self.q1.get()

    def top(self):
        topElement = self.pop()
        self.push(topElement)
        return topElement

    def empty(self):
        return self.q1.empty()
```

【题目】用栈实现队列
【解决思路】
使用两个栈来实现即可
【PYTHON实现】

```
class MyQueue(object):
    def __init__(self):
        self.stack = []

    def push(self, x):
        tmp = []
        while self.stack:
            tmp.append(self.stack.pop())
        tmp.append(x)
        while tmp:
            self.stack.append(tmp.pop())
```

```
    def pop(self):
        return self.stack.pop()

    def peek(self):
        return self.stack[-1]

    def empty(self):
        return self.stack == []
```

【题目】返回数据流中第K大元素

设计一个找到数据流中第K大元素的类（CLASS）。注意是排序后的第K大元素，不是第K个不同的元素。
你的 KTHLARGEST 类需要一个同时接收整数 K 和整数数组NUMS 的构造器，它包含数据流中的初始元素。
每次调用 KTHLARGEST.ADD，返回当前数据流中第K大的元素

【解决思路】

使用最小堆实现

【PYTHON实现】

```
import heapq
class KthLargest(object):
    def __init__(self, k, nums):
        self.k = k
        self.min_heap = []
        for num in nums:
            self.add(num)

    def add(self, val):
        if len(self.min_heap) < self.k:
            heapq.heappush(self.min_heap, val)
        elif val > self.min_heap[0]:
            heapq.heappop(self.min_heap)
            heapq.heappush(self.min_heap, val)
        return self.min_heap[0]
```

【题目】返回滑动窗口中的最大值

给定一个数组 NUMS，有一个大小为 K 的滑动窗口从数组的最左侧移动到数组的最右侧。
你只可以看到在滑动窗口 K 内的数字。滑动窗口每次只向右移动一位

【解决思路】

使用双端队列实现，对于新进入的元素X，只要X比窗口中的元素大，则把窗口里的元素清除掉

【PYTHON实现】

```
class Solution(object):
    def MaxSlidingWindow(self, nums, k):
        if not nums: return []
        window, res = [], []
        fow i,x enumerate(nums):
            if i >= k and window[0] <= i - k
                window.pop(0)
            while window and nums[window[-1]] <= x:
                window.pop()
            window.append(i)
            if i >= k -1:
                res.append(nums[window[0]])
        return res
```

【题目】有效的字母异位词

给定两个字符串 S 和 T ，编写一个函数来判断 T 是否是 S 的一个字母异位词

【解决思路】
使用HASHMAP对字符串进行计数
【PYTHON实现】

```
  def isVnagram(self, s, t):
        dict1, dict = {}, {}
        for item in s:
            dict[item] = dict1.get(item, 0) + 1
        for item in t:
            dict[item] = dict2.get(item, 0) + 1
        return dict1 == dict2
```

【题目】两数之和
给定一个整数数组 NUMS 和一个目标值 TARGET，请你在该数组中找出和为目标值的那 两个整数，并返回他们的数组下标
【解决思路】
使用MAP解决
【PYTHON实现】

```
def TwoSum(self, nums, target):
    hash_map = dict()
    for i,x in enumerate(nums):
        if target - x in hash_map:
            return [i, hash_map[target - x]]
        hash_map[x] = i
```

【题目】验证二叉搜索树
给定一个二叉树，判断其是否是一个有效的二叉搜索树。
假设一个二叉搜索树具有如下特征：
1. 节点的左子树只包含小于当前节点的数。
2. 节点的右子树只包含大于当前节点的数。
3. 所有左子树和右子树自身必须也是二叉搜索树
【解决思路】
1. 使用中序排序（左、中、右），排序的结果是递增
2. 使用递归寻找左子树的最大值、右子树的最小值，和ROOT节点比较，MAX < ROOT; MIN > ROOT
【PYTHON实现】
[解决思路1]

```
def isValidBST(self, root):
    self.prev = None
    return self.helper(root)

def helper(self, root):
    if not root is None:
        return True
    if not self.helper(root.left):
        return False
    if self.prev and self.prev.val >= root.val:
        return False
    self.prev = root
    return self.helper(root.right)
```

[解决思路2]

```
def isValidBST(self, root, min, max):
    if root is None: return True
    if min is not None and root.val >= min:
        return False
    if max is not None and root.val <= max:
        return False
```

```
return self.isValidBST(root.left, min, root.val) and self.isValidBST(root.right, root.val, max)
```

【题目】二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先

【解决思路】

使用递归，分别对左子树和右子树调用递归，若P和Q都存在，则为ROOT，否则为Q或P

【PYTHON实现】

```
class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        if root is None: return True
        if root == p or root == q:
            return root
        left = lowestCommonAncestor(root.left, p, q)
        right = lowestCommonAncestor(root.right, p, q)
        if left and right:
            return root
        return left or right
```

【题目】二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先

【解决思路】

1. 使用递归,只需要与ROOT比较即可

2. 非递归

【PYTHON实现】

[解决思路1]

```
class Solution1(object):
    def lowestCommonAncestor(self, root, p, q):
        if p.val < root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)
        if p.val > root.val < q.val:
            return self.lowestCommonAncestor(root.right, p, q)
        return root
```

[解决思路2]

```
class Solution2(object):
    def lowestCommonAncestor(self, root, p, q):
        while root:
            if p.val < root.val  > q.val:
                root  = root.left
            elif p.val > root.val < q.val:
                root  = root.right
            else:
                return root
```

【题目】POW(X,N)

实现 POW(X, N) ，即计算 X 的 N 次幂函数

【解决思路】

1.使用分治，对X拆半。需要注意对偶数、奇数、负数的处理

2. 使用位运算

【PYTHON实现】

[解决思路1]

```
class Solution:
    def myPow(self, x, n):
        if not n: return 1
        if n < 0:
```

```
            return 1 / self.myPow(x, -n)
        if n % 2:
            return x * self.myPow(x, n - 1)
        return self.myPow(x*x, n/2)
```

[解决思路2]

```
class Solution:
    def myPow(self, x, n):
        if n < 0:
            x = 1 / x
            n = -n
        pow = 1
        while n:
            if n & 1:
                pow *= x
            x *= x
            n >>= 1
        return pow
```

【题目】求众树

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

【解决思路】

使用MAP进行计数

【PYTHON实现】

```
class Sosution(object):
    def majorityElement(self, nums):
        dict  = {}
        for num in nums:
            if n in dict:
                dict[num] = dict[num] + 1
            else:
                dict[num] = 1
            if dict[n] > (len(nums) / 2)
                return num
        return 0
```

【题目】二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）

【解决思路】

1. 广度优先搜索

2. 深度优先搜索

【PYTHON实现】

[解决思路1]

```
class Sosution:
    def levelOrder(self, root):
        if not root: return []
            result = []
            queue = collections.deque()
            queue.append(root)
            while queue:
                level_size = len(queue)
                current_level = []

                for _ in range(level_size):
                    node = queue.popleft()
                    current_level.append(node.val)
```

```
                    if node.left: queue.append(node.left)
                    if node.right: queue.append(node.right)
                result.append(current_level)
            return result
```

[解决思路2]

```
class Solution:
    def levelOrder(self, root):
        if not node: return []
        self.result = []
        self._dfs(root, 0)
        return self.result

    def _dfs(self, node, level):
        if not node: return []
        if len(self.result) < level + 1:
            self.result.append([])
        self.result[level].append(node.val)
        if node.left: self._dfs(node.left, level)
        if node.right: self._dfs(node.right, level)
```

【题目】二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

【解决思路】

使用深度优先搜索,左右节点找到MAX，然后取最大值

【PYTHON实现】

```
class Solution:
    def maxDepth(self, root):
        if not root: return 0
        return 1 + max(self.maxDepth(node.left), self.maxDepth(node.right))
```

【题目】二叉树的最小深度

【解决思路】

需要考虑是否有左、右子树的情况，用分治法算出左右子树的最小值，然后取最小值后加1

【PYTHON实现】

```
class Solution:
    def minDepth(self, root):
        if not root: return 0
        left = self.minDepth(root.left)
        right = self.minDepth(root.right)
        if left is not None or right is not None:
            return min(left, right) + 1
        else:
            return max(left, right) + 1
```

【题目】生成有效括号组合

给出 N 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合

【解决思路】

使用递归搜索，进行剪枝：1.局部不合法，不在递归；2.记录左右已经使用的括号

【PYTHON实现】

```
class Solution:
    def generateParenthes(self, n):
        self.list = []
        self._gen(0, 0, n, "")
        return self.list
```

```
    def _gen(self, left, right, n, result):
        if left == n and right == n:
            self.list.append(result)
            return
        if left < n:
            self._gen(left + 1, right, n, result + "(")
        if left > right and right < n:
            self._gen(left, right + 1, n, resulr + ")")
```

【题目】N皇后问题

N皇后问题研究的是如何将 N 个皇后放置在 N×N 的棋盘上，并且使皇后彼此之间不能相互攻击。

【解决思路】

使用DFS搜索，进行剪枝：用数组记录已经搜过的记标志，COL[J]=1,PIE[I+J]=1,NA[I-J]=1

【PYTHON实现】

```
class Solution:
    def NQuees(self, n):
        if n < 1: return []
        self.result = []
        self.cols, self.pie, self.na = set(), set(), set()
        self.DFS(n, 0 ,[])


    def DFS(self, n, row, cur_state):
        if row < n:
            relf.result.append(cur_state)
            return
        for col range(n):
            if col in self.cols or row + col in self.pie or row - col in self.na:
                continue
            self.cols.add(col)
            self.pie.add(row + col)
            self.na.add(row - col)

            self.DFS(n, row + 1, cur_state + [])

            self.cols.remove(col)
            self.pie.remove(row + col)
            self.na.remove(row - col)
```

【题目】实现一个求解平方根函数

实现 INT SQRT(INT X) 函数。

计算并返回 X 的平方根，其中 X 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

【解决思路】

1. 使用二分查找

2. 牛顿迭代法

【PYTHON实现】

[解决思路1]

```
class Solution:
    def mySqrt(self, n):
        if (n == 0) or (n == 1):
                        return n
        left, right = 1, n
        res = 0
        mid = (left + right) / 2
        if mid == n / mid:
```

```
            return mid
        elif mid > n / mid:
            left = mid - 1
        else:
            right = mid + 1
            res = mid
        return res
```

[解决思路2]

```
class Solution(object):
    def mySqrt(self, x):
        r = x
        while r * r > x:
            r = (r + x / r) / 2
        return r
```

【题目】实现一个字典树

【解决思路】需要理解字典树

【PYTHON实现】

```
class Trie(object):
    def __init__(self):
        self.root = {}
        self.end_of_word = '#'

    def insert(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
        node[self.end_of_word] = self.end_of_word

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node:
                return False
            node = node[char]
        return self.end_of_word in node

    def startsWith(self, prefix):
        node = self.root
        for char in word:
            if char not in node:
                return False
            node = node[char]
        return True
```

【题目】二位网格中单词搜索问题

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，

其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。

同一个单元格内的字母不允许被重复使用

【解决思路】

先建立字典树，然后在使用DFS枚举二维网格

【PYTHON实现】

```
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]
END_OF_WORD = '#'
```

```
class Solution(object):
    def findWords(self, board, words):
        if not board or not board[0]: return []
        if not words: return []

        self.result = set()

        root = collections.defaultdict()

        for word in words:
            node = root
            for char in word:
                node = node.setdefault(char, collections.defaultdict())
                node[END_OF_WORD] = END_OF_WORD

        self.m, self.n = len(board), len(board[0])

        for i in xrange(self.m):
            for j in xrange(self.n):
                if board[i][j] in root:
                    self._dfs(board, i, j, "", root)
    def _dfs(self, board, i, j, cur_word, cur_dict):
        cur_word += board[i][j]
        cur_dict = cur_dict[board][i][j]

        if END_OF_WORD in cur_dict:
            self.result.add(cur_word)

        tmp, board[i][j] = board[i][j], '@'
        for k in xrange(4):
            x, y = i + dx[k], j + dy[k]
            if 0 <= x < self.m and 0 <= y < self.n \
                and board[x][y] != '@' and board[x][y] in cur_dict:
                self._dfs(board, x, y, cur_word, cur_dict)
        board[i][j] = tmp
```

【题目】2的幂次方问题

给定一个整数，编写一个函数来判断它是否是 2 的幂次方

【解决思路】

使用位运算 X & (X - 1)

【PYTHON实现】

```
class Solution(object):
    def isPowerofTwo(self, n):
        return n > 0 and not n & (n - 1)
```

【题目】统计比特位的个数

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 ‘1’ 的个数

【解决思路】使用 X = X & (X - 1)

【PYTHON实现】

```
class Solution(object):
    def hammingWeight(self, n):
        count = 0
        while n != 0:
            count += 1
            n = n & (n - 1)
        return count
```

【N皇后问题】

N皇后问题研究的是如何将 N 个皇后放置在 N×N 的棋盘上，并且使皇后彼此之间不能相互攻击。

【解决思路】

使用位运算来解决

【PYTHON实现】

```
def totalNQueens(self, n):
    if n < 1: return []
    self.count = 0
    self.DFS(n, 0, 0, 0, 0)
    return self.count

def DFS(self, n, row, cols, pie, na):
    # recursion terminator
    if row >= n:
        self.count += 1
        return
    # 得到当前所有空位
    bits = (~(cols | pie | na )) &((1 << n) - 1)
    while bits:
        p = bits & -bits
        self.DFS(n, row +1, cols | p, (pie | p) << 1, (na | p) >> 1)
        bits = bits & (bits - 1) # 去掉最低位的1
```

【题目】爬楼梯

假设你正在爬楼梯。需要 N 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 N 是一个正整数

【解决思路】

1. 使用回溯法，得到F(N) = F(N-1) + F(N - 2)

2. 使用动态规划

【PYTHON实现】

[解决思路1]

```
class Solution:
    def climbStairs(self, n):
        def _climb(n, cache = {}):
            if n in cache.keys()
                return cache[n]
            if n < 3:
                result = n
            else:
                result = _climb(n - 2, cache) + _climb(n - 1, cache)
            cache[n] = result

            return result
        return _climb(n)
```

[解决思路2]

```
class Solution:
    def climbStairs(self, n):
        if n <= 2: return n
        x, y = 1, 1
        for _ in range(1, n):
            x, y = y, x + y
        return y
```

【题目】三角形的最小路径和
给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上
【解决思路】
使用动态规划
状态定义DP[I][J]:从底部到（I，J)距离的最小值，
DP方程：DP[I][J] = MIN(DP(I+1, J), DP(I, J+1)) + TRIANGLE[I][J]
【PYTHON实现】

```
class Solution(object):
    def minumunTotal(self, triangle):
        if not triangle: return 0
        res = triangle[-1]
        for i in xrange(len(trangle)-2, -1, -1):
            for j in xrange(len(trangle[i])):
                res[j] = min(res[j],res[j+1]) + triangle[i][j]
        return res[0]
```

【题目】最长上升子序列
给定一个无序的整数数组，找到其中最长上升子序列的长度
【解决思路】使用动态规划
状态定义：从第1个元素开始到I元素的最长子序列的长度
状态方程：DP[I] = MAX(DP[J]) + 1 (注：J=I-1且A[J]<A[I])
【PYTHON实现】

```
class Solution(object):
    def lengthOfLIS(self, nums):
        if not nums or len(nums) == 0: return 0
        res = 1
        dp = [0 for _ in range(len(nums))]
        for i in range(len(nums))
            dp[i] = 1
            for j in (0, i):
                if num[j] < num[i]:
                    dp[i] = max(dp[i], dp[j] + 1)
                res = max(res, dp[i])
        return res
```

【题目】零钱兑换
给定不同面额的硬币 COINS 和一个总金额 AMOUNT。
编写一个函数来计算可以凑成总金额所需的最少的硬币个数。
如果没有任何一种硬币组合能组成总金额，返回 -1
【解决思路】
使用动态规划
状态定义：I阶台阶的最小步数
状态方程：DP[I] = MIN{DP[I - COINS[J]]} + 1
【PYTHON实现】

```
class Solution(object):
    def coinsChange(self, coins, amount):
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0
        for i in range(1, amount + 1):
            for i range(0, len(coins)):
                if coins[j] <= i:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1)
        return -1 if dp[amount] > amount else dp[amount]
```

【题目】编辑距离

给定两个单词 WORD1 和 WORD2，计算出将 WORD1 转换成 WORD2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

1. 插入一个字符

2. 删除一个字符

3. 替换一个字符

【解决思路】

使用动态规划

状态定义：DP[I][J]表示WORD1前I个字符替换到WORD2的前J个字符最小需要的操作步数

DP方程：

IF WORD1[I] == WORD2[J]: DP[I-1][J-1]

ELSE: 1 + MIN(DP[I-1][J], DP[I][J-1], DP[I-1][J-1])

【PYTHON实现】

```
class Solution(object):
    def minDistance(self, word1, word2):
        m,n = len(word1), len(word2)
        dp = [ [0 for _ in range(m+1)] for _ in range(n+1)]

        for i in range(m+1): dp[i][0] = i
        for j in range(n+1): dp[0][j] = j

        for i in range(1, m+1):
            for j in range(1, n+1):
                dp[i][j] = dp[i-1][j-1] if word1[i-1] == word2[j-1] else min(dp[i-1][j], dp[i][j-1], dp[i-1]
[j-1]) + 1
        return dp[m][n]
```