Leetcode(python版) 栈中等题

71简化路径

题目：

给定一个文档 (Unix-style) 的完全路径，请进行路径简化。

例如，  
**path** = "/home/", => "/home"  
**path** = "/a/./b/../../c/", => "/c"

**边界情况:**

* 你是否考虑了 路径 = "/../" 的情况？  
  在这种情况下，你需返回 "/" 。
* 此外，路径中也可能包含多个斜杠 '/' ，如 "/home//foo/" 。  
  在这种情况下，你可忽略多余的斜杠，返回 "/home/foo" 。

思路：看到这种来来回回，增增删删的题，一般都想到用栈。

我们把字符串按照/分割之后就得到了每个文件的目录，然后判断路径是添加还是向上层进行返回。这个题很简单了。

代码：

1. **class** Solution(object):
2. **def** simplifyPath(self, path):
3. """
4. :type path: str
5. :rtype: str
6. """
7. stack=list()
8. dirs=path.split('/')
9. **for** dir **in** dirs:
10. **if** **not** dir **or** dir=='.':
11. **continue**
12. **if** dir=='..':
13. **if** stack:
14. stack.pop()
15. **else**:
16. stack.append(dir)
17. **return** '/'+'/'.join(stack)

94二叉树的中序遍历

题目：

给定一个二叉树，返回它的*中序*遍历。

**示例:**

**输入:** [1,null,2,3]

1

\

2

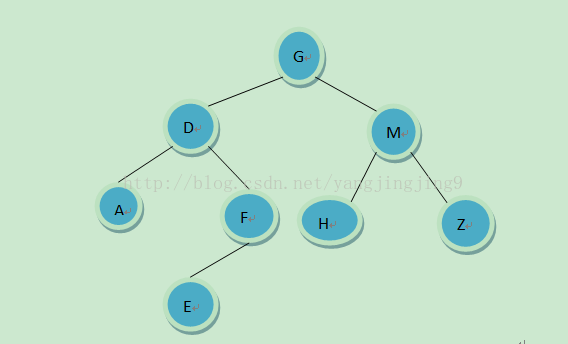
/

3

**输出:** [1,3,2]

**进阶:** 递归算法很简单，你可以通过迭代算法完成吗？

思路一：用迭代的方法，非递归



l 根结点G入栈，若入栈的结点存在左子树，则依次入栈G/D/A，直至A发现其左子树为空，停止入栈，此时栈stack = [G,D,A]。

l A出栈，并对A进行遍历，发现A没有右子树，根据中序遍历，需要遍历A的根节点D，D出栈，D存在右孩子，将其右孩子F入栈，F有左子树E，E入栈，此时stack = [G,F,E]，res=[A,D]

l E出栈，并对E进行遍历，发现没有右子树，根据中序遍历，需要遍历E的根结点F，F出栈，此时栈为stack = [G],res = [A,D, E, F]

l G出栈，并遍历G，G有右子树，将右子树M入栈，此时栈为stack = [M],res = [A,D, E, F,G]

l 右子树根结点M，按照中序遍历规则重复以上步骤：

M存在左子树H入栈，stack = [M,H],res= [A,D, E, F,G]

H出栈，遍历H，发现H没有右子树，根据中序遍历，需要遍历H的根结点M，M有右子树Z，Z入栈，此时stack = [Z],res = [A,D, E, F,G,H,M]

Z出栈，遍历Z，发现Z没有右子树，此时stack= [],res = [A,D, E, F,G,H,M,Z]

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** inorderTraversal(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. res=[]
15. self.iterative\_inorder(root,res)
16. **print**(res)
17. **return** res
18. **def** iterative\_inorder(self,root,res):
19. stack=[]
20. **while** root **or** stack:
21. **if** root:
22. stack.append(root)
23. root=root.left
24. **else**:
25. root=stack.pop()
26. res.append(root.val)
27. root=root.right
28. **return** res

思路二：利用递归的方法

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** inorderTraversal(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. res=[]
15. self.recursive\_inorder(root,res)
16. **print**(res)
17. **return** res
18. **def** recursive\_inorder(self,root,res):
19. **if** root:
20. self.recursive\_inorder(root.left,res)
21. res.append(root.val)
22. self.recursive\_inorder(root.right,res)

103二叉树的锯齿形层次遍历

题目：

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：  
给定二叉树 [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回锯齿形层次遍历如下：

[

[3],

[20,9],

[15,7]

]

思路：还是利用队列，但是要在res中加入templist时加了一步判断flag是否为负的步骤，如果为负，表示是偶数次遍历应该从右往左，将templist更新为逆序

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** zigzagLevelOrder(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[List[int]]
13. """
14. **if** **not** root:
15. **return** []
16. queue=[root]
17. res=[]
18. flag=1
19. **while** queue:
20. templist=[]
21. length=len(queue)
22. **for** i **in** range(length):
23. temp=queue.pop(0)
24. templist.append(temp.val)
25. **if** temp.left:
26. queue.append(temp.left)
27. **if** temp.right:
28. queue.append(temp.right)
29. **if** flag==-1:
30. templist=templist[::-1]
31. res.append(templist)
32. flag \*=-1
33. **return** res

144二叉树的前序遍历

题目：

给定一个二叉树，返回它的 *前序*遍历。

**示例:**

**输入:** [1,null,2,3]

1

\

2

/

3

**输出:** [1,2,3]

**进阶:** 递归算法很简单，你可以通过迭代算法完成吗？

思路：非递归

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** preorderTraversal(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. result = list()
15. **if** root == None:
16. **return** result
18. stack = list()
19. stack.append(root)
20. **while** len(stack) != 0:
21. top = stack.pop()
22. **if** top.right != None:
23. stack.append(top.right)
24. **if** top.left != None:
25. stack.append(top.left)
26. result.append(top.val)
28. **return** result

150逆波兰表达式求值

题目：

根据[逆波兰表示法](https://baike.baidu.com/item/%E9%80%86%E6%B3%A2%E5%85%B0%E5%BC%8F/128437" \t "_blank)，求表达式的值。

有效的运算符包括 +, -, \*, / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

**说明：**

* 整数除法只保留整数部分。
* 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

**示例 1：**

**输入:** ["2", "1", "+", "3", "\*"]

**输出:** 9

**解释:** ((2 + 1) \* 3) = 9

**示例 2：**

**输入:** ["4", "13", "5", "/", "+"]

**输出:** 6

**解释:** (4 + (13 / 5)) = 6

**示例 3：**

**输入:** ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]

**输出:** 22

**解释:**

((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

思路：建立一个栈，遇到数字就压入栈中，遇到运算符就弹出两个数字计算后再压回栈中，直到结束

代码：

1. **class** Solution(object):
2. evalRPN(self, tokens):
3. """
4. :type tokens: List[str]
5. :rtype: int
6. """
7. ans = []
8. **for** i **in** tokens:
9. **if** i != '/' **and** i != '\*' **and** i != '+' **and** i != '-':
10. ans.append(int(i))
11. **else**:
12. tmp1 = ans.pop()
13. tmp2 = ans.pop()
14. **if** i == '/':
15. **if** tmp1\*tmp2 < 0:
16. ans.append(-((-tmp2) // tmp1))
17. **else**:
18. ans.append(tmp2/tmp1)
19. **if** i == '\*':
20. ans.append(tmp2\*tmp1)
21. **if** i == '+':
22. ans.append(tmp2 + tmp1)
23. **if** i == '-':
24. ans.append(tmp2 - tmp1)
25. **return** ans[0]

173二叉搜索树迭代器

题目：实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。

调用 next() 将返回二叉搜索树中的下一个最小的数。

注意:next() 和hasNext() 操作的时间复杂度是O(1)，并使用 *O(h)*内存，其中 *h*是树的高度。

思路：一般地，对时间要求比较严格的，我们可以使用空间进行补偿。所以使用一个栈，在初始化的过程中，就使用中序遍历，把BST的中序遍历是有序的这个特点用上。再定义hasnext()和next()就很容易了。Stack中序遍历的代码可以画个图推演一遍就可以理解了。

代码：

1. # Definition for a  binary tree node
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** BSTIterator(object):
9. **def** \_\_init\_\_(self, root):
10. """
11. :type root: TreeNode
12. """
13. self.stack=[]
14. self.inOrder(root)
16. **def** inOrder(self, root):
17. **if** **not** root:
18. **return**
19. self.inOrder(root.right)
20. self.stack.append(root.val)
21. self.inOrder(root.left)
23. **def** hasNext(self):
24. """
25. :rtype: bool
26. """
27. **return** len(self.stack)>0
29. **def** next(self):
30. """
31. :rtype: int
32. """
33. **return** self.stack.pop()
35. # Your BSTIterator will be called like this:
36. # i, v = BSTIterator(root), []
37. # while i.hasNext(): v.append(i.next())

331验证二叉树的前序序列化

题目：

序列化二叉树的一种方法是使用前序遍历。当我们遇到一个非空节点时，我们可以记录下这个节点的值。如果它是一个空节点，我们可以使用一个标记值记录，例如 #。

\_9\_

/ \

3 2

/ \ / \

4 1 # 6

/ \ / \ / \

# # # # # #

例如，上面的二叉树可以被序列化为字符串 "9,3,4,#,#,1,#,#,2,#,6,#,#"，其中 # 代表一个空节点。

给定一串以逗号分隔的序列，验证它是否是正确的二叉树的前序序列化。编写一个在不重构树的条件下的可行算法。

每个以逗号分隔的字符或为一个整数或为一个表示 null 指针的 '#' 。

你可以认为输入格式总是有效的，例如它永远不会包含两个连续的逗号，比如 "1,,3" 。

**示例 1:**

**输入:** "9,3,4,#,#,1,#,#,2,#,6,#,#"

**输出:** true

**示例 2:**

**输入:** "1,#"

**输出:** false

**示例 3:**

**输入:** "9,#,#,1"

**输出:** false

思路：我们的思路应该是这样的：判断一个二叉树是否合法的情况，那么应该是个递归或者循环问题。那么解决问题的思路是从顶部向下分析还是从底下向顶部分析呢？正确的结果应该是从二叉树的底部向上进行分析，因为我们可以通过#号判断是否是空节点，然后判断最底下的叶子节点是否含有两个空孩子的方式，循环向上解决这个问题。

所以这个题的思路就很明了了：用一个栈，从字符串的左侧向右依次进栈，如果满足栈的后三位是数字，#，#的模式时，说明可以构成合法的叶子节点，把这个叶子节点换成#号，代表空节点，然后继续遍历。最后应该只剩下一个#，那么就是一个合法的二叉树。

代码：

1. **class** Solution(object):
2. def isValidSerialization(self, preorder):
3. """
4. :type preorder: str
5. :rtype: bool
6. """
7. stack=[]
8. **for** node in preorder.split(','):
9. stack.append(node)
10. **while** len(stack)>=3 **and** stack[-1]==stack[-2]=='#' **and** stack[-3]!='#':
11. stack.pop()
12. stack.pop()
13. stack.pop()
14. stack.append('#')
15. **return** len(stack)==1 **and** stack.pop()=='#'

思路二：

这个方法还是第一次使用：看出度和入度的差。

我们知道一个树（甚至图），所有节点的入度之和等于出度之和。那么可以根据这个条件进行有效性的判断。

对于二叉树，我们把空的地方也作为叶子节点（如题目中的#），那么有

所有的非空节点提供2个出度和1个入度（根除外）

所有的空节点但提供0个出度和1个入度

我们在遍历的时候，计算diff = outdegree – indegree. 当一个节点出现的时候，diff – 1，因为它提供一个入度；当节点不是#的时候，diff+2(提供两个出度) 如果序列式合法的，那么遍历过程中diff >=0 且最后结果为0.

这里解释一下为什么diff的初始化为1.因为，我们加入一个非空节点时，都会先减去一个入度，再加上两个出度。但是由于根节点没有父节点，所以其入度为0，出度为2.因此diff初始化为1，是为了再加入根节点的时候，先减去一个入度，再加上两个出度，正好应该是2.

代码：

1. **class** Solution(object):
2. **def** isValidSerialization(self, preorder):
3. """
4. :type preorder: str
5. :rtype: bool
6. """
7. nodes = preorder.split(',')
8. diff = 1
9. **for** node **in** nodes:
10. diff -= 1
11. **if** diff < 0:
12. **return** False
13. **if** node != '#':
14. diff += 2
15. **return** diff == 0

341扁平化嵌套列表迭代器

题目：给定一个嵌套的整型列表。设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的项或者为一个整数，或者是另一个列表。

**示例 1:**

**输入:** [[1,1],2,[1,1]]

**输出:** [1,1,2,1,1]

**解释:** 通过重复调用 *next* 直到 *hasNex*t 返回false，*next*返回的元素的顺序应该是: [1,1,2,1,1]。

**示例 2:**

**输入:** [1,[4,[6]]]

**输出:** [1,4,6]

**解释:** 通过重复调用 *next*直到 *hasNex*t 返回false，*next*返回的元素的顺序应该是: [1,4,6]。

思路：需要我们设计一个数据结构保存嵌套数组的每个元素，一般选择列表，但为了速度快，我们选择了队列。

重点是利用递归把整个嵌套的列表迭代器给压平。注意，题目已经给了我们它的数据结构，而不是普通的list。所以我们必须用他的函数。题目中虽然是多重嵌套，但是归根到底，对于每层的嵌套都是一个一维数组而已。因此，不要想复杂，直接循环该一维数组，如果是整数，添加到队列中，如果是嵌套的列表则继续解嵌套。。最后的结果是有按照从左到右有序的，这个可以放心哈

代码：

1. # """
2. # This is the interface that allows for creating nested lists.
3. # You should not implement it, or speculate about its implementation
4. # """
5. #class NestedInteger(object):
6. #    def isInteger(self):
7. #        """
8. #        @return True if this NestedInteger holds a single integer, rather than a nested list.
9. #        :rtype bool
10. #        """
11. #
12. #    def getInteger(self):
13. #        """
14. #        @return the single integer that this NestedInteger holds, if it holds a single integer
15. #        Return None if this NestedInteger holds a nested list
16. #        :rtype int
17. #        """
18. #
19. #    def getList(self):
20. #        """
21. #        @return the nested list that this NestedInteger holds, if it holds a nested list
22. #        Return None if this NestedInteger holds a single integer
23. #        :rtype List[NestedInteger]
24. #        """
26. **class** NestedIterator(object):
28. **def** \_\_init\_\_(self, nestedList):
29. """
30. Initialize your data structure here.
31. :type nestedList: List[NestedInteger]
32. """
33. self.queue=collections.deque()
34. **def** getAll(nests):
35. **for** nest **in** nests:
36. **if** nest.isInteger():
37. self.queue.append(nest.getInteger())
38. **else**:
39. getAll(nest.getList())
40. getAll(nestedList)
42. **def** next(self):
43. """
44. :rtype: int
45. """
46. **return** self.queue.popleft()
48. **def** hasNext(self):
49. """
50. :rtype: bool
51. """
52. **return** len(self.queue)
54. # Your NestedIterator object will be instantiated and called as such:
55. # i, v = NestedIterator(nestedList), []
56. # while i.hasNext(): v.append(i.next())

385迷你语法分析器

题目：给定一个用字符串表示的整数的嵌套列表，实现一个解析它的语法分析器。

列表中的每个元素只可能是整数或整数嵌套列表

**提示：**你可以假定这些字符串都是格式良好的：

* 字符串非空
* 字符串不包含空格
* 字符串只包含数字0-9, [, - ,, ]

**示例 1：**

给定 s = "324",

你应该返回一个 NestedInteger 对象，其中只包含整数值 324。

**示例 2：**

给定 s = "[123,[456,[789]]]",

返回一个 NestedInteger 对象包含一个有两个元素的嵌套列表：

1. 一个 integer 包含值 123

2. 一个包含两个元素的嵌套列表：

i. 一个 integer 包含值 456

ii. 一个包含一个元素的嵌套列表

a. 一个 integer 包含值 78

思路：直接eval就能把字符串转成一个数组，而且这个数组的每个元素已经转成了int。

代码：

1. # """
2. # This is the interface that allows for creating nested lists.
3. # You should not implement it, or speculate about its implementation
4. # """
5. #class NestedInteger(object):
6. #    def \_\_init\_\_(self, value=None):
7. #        """
8. #        If value is not specified, initializes an empty list.
9. #        Otherwise initializes a single integer equal to value.
10. #        """
11. #
12. #    def isInteger(self):
13. #        """
14. #        @return True if this NestedInteger holds a single integer, rather than a nested list.
15. #        :rtype bool
16. #        """
17. #
18. #    def add(self, elem):
19. #        """
20. #        Set this NestedInteger to hold a nested list and adds a nested integer elem to it.
21. #        :rtype void
22. #        """
23. #
24. #    def setInteger(self, value):
25. #        """
26. #        Set this NestedInteger to hold a single integer equal to value.
27. #        :rtype void
28. #        """
29. #
30. #    def getInteger(self):
31. #        """
32. #        @return the single integer that this NestedInteger holds, if it holds a single integer
33. #        Return None if this NestedInteger holds a nested list
34. #        :rtype int
35. #        """
36. #
37. #    def getList(self):
38. #        """
39. #        @return the nested list that this NestedInteger holds, if it holds a nested list
40. #        Return None if this NestedInteger holds a single integer
41. #        :rtype List[NestedInteger]
42. #        """
44. **class** Solution(object):
45. **def** deserialize(self, s):
46. """
47. :type s: str
48. :rtype: NestedInteger
49. """
50. **def** getNumber(nums):
51. **if** isinstance(nums, int):
52. **return** NestedInteger(nums)
53. lst = NestedInteger()
54. **for** num **in** nums:
55. lst.add(getNumber(num))
56. **return** lst
57. **return** getNumber(eval(s))

394字符串解码

题目：给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded\_string]，表示其中方括号内部的 *encoded\_string* 正好重复 *k* 次。注意 *k* 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 *k* ，例如不会出现像 3a 或 2[4] 的输入。

**示例:**

s = "3[a]2[bc]", 返回 "aaabcbc".

s = "3[a2[c]]", 返回 "accaccacc".

s = "2[abc]3[cd]ef", 返回 "abcabccdcdcdef".

* 思路：思路：使用两个栈分别存储数字和字母
* 注意1: 数字是多位的话，要处理后入数字栈
* 注意2: 出栈时过程中产生的组合后的字符串要继续入字母栈
* 注意3: 记得字母出栈的时候字符要逆序组合成字符串
* 注意4: 不用字符串而用字母栈的原因是字符串的 join 效率会比字符串加法高一些

代码：

1. **class** Solution(object):
2. **def** decodeString(self, s):
3. """
4. :type s: str
5. :rtype: str
6. """
7. nums,chars=[],[]
8. i,length=0,len(s)
9. **while** i<length:
10. j=i+1
11. **if** s[i].isdigit():
12. num=int(s[i])
13. **while** j<length:
14. **if** s[j].isdigit():
15. num=num\*10+int(s[j])
16. j+=1
17. **else**:
18. **break**
19. nums.append(num)
20. **elif** s[i]=='[' **or** s[i].isalpha():  #把[和字母压入栈中
21. chars.append(s[i])
22. **else**:
23. t,tmpc=chars.pop(),[]
24. **while** t!='[':
25. tmpc.append(t)  #把字母依次取出来，但此时的顺序是反的，所以接下来要做个调转
26. t=chars.pop()
27. tchars=nums.pop() \*''.join(tmpc[::-1])
28. chars.append(tchars)
29. i=j
30. **return** ''.join(chars)

402移除K位数字

题目：给定一个以字符串表示的非负整数 *num*，移除这个数中的 *k* 位数字，使得剩下的数字最小。

**注意:**

* *num* 的长度小于 10002 且 ≥ *k。*
* *num* 不会包含任何前导零。

**示例 1 :**

输入: num = "1432219", k = 3

输出: "1219"

解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。

**示例 2 :**

输入: num = "10200", k = 1

输出: "200"

解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。

示例 **3 :**

输入: num = "10", k = 2

输出: "0"

解释: 从原数字移除所有的数字，剩余为空就是0。

思路：使用一个栈作为辅助，遍历数字字符串，当当前的字符比栈最后的字符小的时候，说明要把栈的最后的这个字符删除掉。为什么呢？你想，把栈最后的字符删除掉，然后用现在的字符进行替换，是不是数字比以前的那种情况更小了？所以同样的道理，做一个while循环！这个很重要，可是我没有想到。在每一个数字处理的时候，都要做一个循环，使得栈里面最后的数字比当前数字大的都弹出去。

最后，如果K还没用完，那要删除哪里的字符呢？毋庸置疑肯定是最后的字符，因为前面的字符都是小字符。

1. 代码：**class** Solution(object):
2. **def** removeKdigits(self, num, k):
3. """
4. :type num: str
5. :type k: int
6. :rtype: str
7. """
8. **if** len(num)<=k:
9. **return** '0'
10. stack=[]
11. **for** n **in** num:
12. **while** stack **and** k **and** int(stack[-1])>int(n):
13. stack.pop()
14. k-=1
15. stack.append(n)
16. **while** k:
17. stack.pop()
18. k-=1
19. **if** **not** stack:
20. **return** '0'
21. **return** str(int("".join(stack)))

456 132模式

题目：给定一个整数序列：a1, a2, ..., an，一个132模式的子序列 a**i**, a**j**, a**k** 被定义为：当 **i** < **j** < **k** 时，a**i** < a**k** < a**j**。设计一个算法，当给定有 n 个数字的序列时，验证这个序列中是否含有132模式的子序列。

**注意：**n 的值小于15000。

**示例1:**

**输入:** [1, 2, 3, 4]

**输出:** False

**解释:** 序列中不存在132模式的子序列。

**示例 2:**

**输入:** [3, 1, 4, 2]

**输出:** True

**解释:** 序列中有 1 个132模式的子序列： [1, 4, 2].

**示例 3:**

**输入:** [-1, 3, 2, 0]

**输出:** True

**解释:** 序列中有 3 个132模式的的子序列: [-1, 3, 2], [-1, 3, 0] 和 [-1, 2, 0].

思路：下面这种方法利用栈来做，既简洁又高效，思路是我们维护一个栈和一个变量third，其中third就是第三个数字，也是pattern

132中的2，栈里面按顺序放所有大于third的数字，也是pattern

132中的3，那么我们在遍历的时候，如果当前数字小于third，即pattern

132中的1找到了，我们直接返回true即可，因为已经找到了，注意我们应该从后往前遍历数组。

如果当前数字大于栈顶元素，那么我们按顺序将栈顶数字取出，赋值给third，然后将该数字压入栈，这样保证了栈里的元素仍然都是大于third的，我们想要的顺序依旧存在，进一步来说，栈里存放的都是可以维持second > third的second值，其中的任何一个值都是大于当前的third值，如果有更大的值进来，那就等于形成了一个更优的second > third的这样一个组合，并且这时弹出的third值比以前的third值更大，为什么要保证third值更大，因为这样才可以更容易的满足当前的值first比third值小这个条件。

代码：

1. **class** Solution(object):
2. **def** find132pattern(self, nums):
3. """
4. :type nums: List[int]
5. :rtype: bool
6. """
7. **if** len(nums) <=2:
8. **return** False
9. third = float('-inf')
10. stack = []
11. **for** i **in** range(len(nums)-1, -1, -1):
12. **if** nums[i] < third:
13. **return** True
14. **else**:
15. **while** stack **and** stack[-1] < nums[i]:
16. third = stack.pop()
17. stack.append(nums[i])
18. **return** False

503下一个更大元素II

题目：给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

**示例 1:**

**输入:** [1,2,1]

**输出:** [2,-1,2]

**解释:** 第一个 1 的下一个更大的数是 2；

数字 2 找不到下一个更大的数；

第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

思路：在遍历数组的过程中，如果是往后遇到大的数，那就是第一个更大的数，如果一直遇到不断小的数，才会一直找不到，我们可以用一个栈来记录，遇到比栈顶小的数字就放入栈中，遇到比栈顶大的数字就说明这是栈顶数字的下一个更大的数，就将其放在结果数组的对应位置上，栈顶的元素出栈，继续比较新的栈顶的数，如果还是大，那么继续记录，出栈，直到栈顶的数比新数要小，那么就可以将新数入栈了。因为我们要将找到的更大的数放在对应位置上，所以栈中记录的应该是元素位置，而不是具体的数字，但比较的时候还是比较原来的数组中这个位置的数字，这一点要想清楚。此外，因为会出现循环寻找的情况，所以数组我们可能遍历两次。这个做法会快很多。

注意，栈里保存的是索引，一定不要忘记

代码：

1. **class** Solution(object):
2. **def** nextGreaterElements(self, nums):
3. """
4. :type nums: List[int]
5. :rtype: List[int]
6. """
7. res=[-1]\*len(nums)
8. stack=[]
9. **for** i **in** range(len(nums))\*2:
10. **while** stack **and** (nums[stack[-1]]<nums[i]):
11. res[stack.pop()]=nums[i]
12. stack.append(i)
13. **return** res

636函数的独占时间

题目：给出一个非抢占单线程CPU的 **n** 个函数运行日志，找到函数的独占时间。

每个函数都有一个唯一的 Id，从 **0** 到 **n-1**，函数可能会递归调用或者被其他函数调用。

日志是具有以下格式的字符串：function\_id：start\_or\_end：timestamp。例如："0:start:0" 表示函数 0 从 0 时刻开始运行。"0:end:0" 表示函数 0 在 0 时刻结束。

函数的独占时间定义是在该方法中花费的时间，调用其他函数花费的时间不算该函数的独占时间。你需要根据函数的 Id 有序地返回每个函数的独占时间。

**示例 1:**

**输入:**

n = 2

logs =

["0:start:0",

"1:start:2",

"1:end:5",

"0:end:6"]

**输出:**[3, 4]

**说明：**

函数 0 在时刻 0 开始，在执行了 2个时间单位结束于时刻 1。

现在函数 0 调用函数 1，函数 1 在时刻 2 开始，执行 4 个时间单位后结束于时刻 5。

函数 0 再次在时刻 6 开始执行，并在时刻 6 结束运行，从而执行了 1 个时间单位。

所以函数 0 总共的执行了 2 +1 =3 个时间单位，函数 1 总共执行了 4 个时间单位。

**说明：**

1. 输入的日志会根据时间戳排序，而不是根据日志Id排序。
2. 你的输出会根据函数Id排序，也就意味着你的输出数组中序号为 0 的元素相当于函数 0 的执行时间。
3. 两个函数不会在同时开始或结束。
4. 函数允许被递归调用，直到运行结束。
5. 1 <= n <= 100

思路：

代码：

1. **class** Solution(object):
2. **def** exclusiveTime(self, n, logs):
3. """
4. :type n: int
5. :type logs: List[str]
6. :rtype: List[int]
7. """
8. rList=[0]\*n
9. startList=[]
10. preId=int(logs[0].split(':')[0])
11. preKind=logs[0].split(':')[1]
12. preTime=int(logs[0].split(':')[2])
13. startList.append(preId)
14. **for** i **in** range(1,len(logs)):
15. Id=int(logs[i].split(':')[0])
16. Kind=logs[i].split(':')[1]
17. Time=int(logs[i].split(':')[2])
18. **if**(Kind=='start'):
19. startList.append(Id)
20. #被抢占，抢占前运行时间
21. #Id抢占资源，preId进入等待。这种情况要计算preId等待前已经运行的时间
22. **if**(preKind=='start'):
23. rList[preId]+=(Time-preTime)
24. #被抢占，抢占后运行时间
25. #preId结束后到Id开始前，有一段时间，这段时间其实就是startList上最近一个等待的线程在运行的时间，参考实例如下
26. #example2:n=2,logs=["0:start:0","0:start:2","0:end:5","1:start:7","1:end:7","0:end:8"]
27. #在"0:end:5","1:start:7"期间，有个时间片6，这个时间就是"0:start:0"运行的时间
28. **else**:
29. **if**(len(startList)>1):
30. rList[startList[-2]]+=(Time-preTime-1)
31. **else**:
32. **del** startList[-1]
33. #没有被抢占的线程运行时间
34. #由于是递归调用，所以这种情况下，必有Id=preId，参考实例如下
35. #example1:n=1,logs=["0:start:0","0:start:2","0:end:5","0:start:6","0:end:6","0:end:7"]
36. #在"0:start:2","0:end:5"，出现一个end，如果前一个是start，则必有Id=preId
37. **if**(preKind=='start'):
38. rList[Id]+=(Time-preTime+1)
39. #被抢占，抢占后运行时间
40. **else**:
41. rList[Id]+=(Time-preTime)
42. preId=Id
43. preKind=Kind
44. preTime=Time
45. **return** rList

735行星碰撞

题目：给定一个整数数组 asteroids，表示在同一行的行星。

对于数组中的每一个元素，其绝对值表示行星的大小，正负表示行星的移动方向（正表示向右移动，负表示向左移动）。每一颗行星以相同的速度移动。

找出碰撞后剩下的所有行星。碰撞规则：两个行星相互碰撞，较小的行星会爆炸。如果两颗行星大小相同，则两颗行星都会爆炸。两颗移动方向相同的行星，永远不会发生碰撞。

**示例 1:**

**输入:**

asteroids = [5, 10, -5]

**输出:** [5, 10]

**解释:**

10 和 -5 碰撞后只剩下 10。 5 和 10 永远不会发生碰撞。

**示例 2:**

**输入:**

asteroids = [8, -8]

**输出:** []

**解释:**

8 和 -8 碰撞后，两者都发生爆炸。

**示例 3:**

**输入:**

asteroids = [10, 2, -5]

**输出:** [10]

**解释:**

2 和 -5 发生碰撞后剩下 -5。10 和 -5 发生碰撞后剩下 10。

**示例 4:**

**输入:**

asteroids = [-2, -1, 1, 2]

**输出:** [-2, -1, 1, 2]

**解释:**

-2 和 -1 向左移动，而 1 和 2 向右移动。

由于移动方向相同的行星不会发生碰撞，所以最终没有行星发生碰撞。

**说明:**

* 数组 asteroids 的长度不超过 10000。
* 每一颗行星的大小都是非零整数，范围是 [-1000, 1000] 。

思路：当我们意识到，行星是两两之间互相作用的，那我们很容易就想到了用栈。因为栈能处理这样抵消和遗留的问题。

算法的思想是，从左到右遍历每个行星，并和栈顶数字相比较，当栈顶数字为正（向右），当前数字为负（向左）的时候，会发生碰撞。这时候，判断遗留下来的数字是多少，保存到ast里，如果ast为空代表啥都没有了，如果ast质量大于栈顶元素会留下来ast，否则留下pre。判断ast是否为空，不为空就把遗留下来的数字进栈就好了

代码：

1. **class** Solution(object):
2. **def** asteroidCollision(self, asteroids):
3. """
4. :type asteroids: List[int]
5. :rtype: List[int]
6. """
7. stack=[]
8. **for** ast **in** asteroids:
9. **while** stack **and** ast<0 **and** stack[-1]>=0:
10. pre=stack.pop()
11. **if** ast==-pre:
12. ast=None
13. **break**
14. **elif** -ast<pre:
15. ast=pre
16. **if** ast!=None:
17. stack.append(ast)
19. **return** stack

739每日温度

题目：根据每日气温列表，请重新生成一个列表，对应位置的输入是你需要再等待多久温度才会升高的天数。如果之后都不会升高，请输入 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

**提示：**气温列表长度的范围是 [1, 30000]。每个气温的值的都是 [30, 100] 范围内的整数。

思路：这个题难在找到下一个比当前气温高的位置和当前位置的差。注意到题目中温度变化范围只有60，而天数的变化范围有30000，所以对温度遍历是可以接受的，对天数遍历不可接受。

所以我们倒序遍历温度，保留每个温度的最新的天数位置，保存在字典中。对当前的温度，我们从字典中找所有比他大的温度的位置，保留最小值。如果没有比他大的，就写入0.

代码：

1. **class** Solution(object):
2. **def** dailyTemperatures(self, temperatures):
3. """
4. :type temperatures: List[int]
5. :rtype: List[int]
6. """
7. save={}
8. answer=[]
9. **for** day **in** range(len(temperatures)-1,-1,-1):
10. tmp=temperatures[day]
11. save[tmp]=day
12. larger=[]
13. **for** i **in** range(tmp+1,102):
14. **if** i **in** save:
15. larger.append(save[i]-day)
16. **if** larger:
17. answer.append(min(larger))
18. **else**:
19. answer.append(0)
21. **return** answer[::-1]

853车队

题目：N  辆车沿着一条车道驶向位于 target 英里之外的共同目的地。

每辆车 i 以恒定的速度 speed[i] （英里/小时），从初始位置 position[i] （英里） 沿车道驶向目的地。

一辆车永远不会超过前面的另一辆车，但它可以追上去，并与前车以相同的速度紧接着行驶。

此时，我们会忽略这两辆车之间的距离，也就是说，它们被假定处于相同的位置。

*车队*是一些由行驶在相同位置、具有相同速度的车组成的非空集合。注意，一辆车也可以是一个车队。

即便一辆车在目的地才赶上了一个车队，它们仍然会被视作是同一个车队。

会有多少车队到达目的地?

**示例：**

**输入：**target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

**输出：**3

**解释：**

从 10 和 8 开始的车会组成一个车队，它们在 12 处相遇。

从 0 处开始的车无法追上其它车，所以它自己就是一个车队。

从 5 和 3 开始的车会组成一个车队，它们在 6 处相遇。

请注意，在到达目的地之前没有其它车会遇到这些车队，所以答案是 3。

**提示：**

1. 0 <= N <= 10 ^ 4
2. 0 < target <= 10 ^ 6
3. 0 < speed[i] <= 10 ^ 6
4. 0 <= position[i] < target

思路：把车按照位置大小进行排序，计算出每个车在无阻拦的情况下到达终点的时间，如果后面的车到达终点所用的时间比前面车小，那么说明后车应该比前面的车先到。但是由于后车不能超过前车，所以这种情况下就会合并成一个车队，也就是说后车“消失了”。

然后像这种需要判断是否存在的题目一般都是用栈进行解决，对时间遍历，把哪些应该消失的车不进栈就行了

代码：（提交时有些案例通过不了）

1. **class** Solution(object):
2. **def** carFleet(self, target, position, speed):
3. """
4. :type target: int
5. :type position: List[int]
6. :type speed: List[int]
7. :rtype: int
8. """
9. cars = [(pos, spe) **for** pos, spe **in** zip(position, speed)]
10. sorted\_cars = sorted(cars)
11. times = [(target - pos) / spe **for** pos, spe **in** sorted\_cars]
12. stack = []
13. **for** time **in** times[::-1]:
14. **if** **not** stack:
15. stack.append(time)
16. **else**:
17. **if** time > stack[-1]:
18. stack.append(time)
19. **return** len(stack)

代码二：

1. **class** Solution(object):
2. **def** carFleet(self, target, position, speed):
3. """
4. :type target: int
5. :type position: List[int]
6. :type speed: List[int]
7. :rtype: int
8. """
9. **if** len(position) < 2: **return** len(position)
11. # Get a sorted list of (position, time-to-target)
12. ts = [(target-v)\*1.0/s **for** (v, s) **in** zip(position, speed)]
13. v = sorted(zip(position, ts))
15. # Count the number of slowest fleets
16. ret, slowest = 0, -1.0
17. **for** (\_, t) **in** reversed(v):
18. **if** t > slowest:
19. slowest = t
20. ret += 1
21. **return** ret

856括号的分数

题目：

给定一个平衡括号字符串 S，按下述规则计算该字符串的分数：

* () 得 1 分。
* AB 得 A + B 分，其中 A 和 B 是平衡括号字符串。
* (A) 得 2 \* A 分，其中 A 是平衡括号字符串。

**示例 1：**

**输入：** "()"

**输出：** 1

**示例 2：**

**输入：** "(())"

**输出：** 2

**示例 3：**

**输入：** "()()"

**输出：** 2

**示例 4：**

**输入：** "(()(()))"

**输出：** 6

**提示：**

1. S 是平衡括号字符串，且只含有 ( 和 ) 。
2. 2 <= S.length <= 50

思路：依次压入栈中，遇到），如果前一个为（，则pop (，然后压入1，如果是数字，则pop两个，然后将2\*数字，如果遇到数字，则看如果前一个也是数字，则相加后压入

代码：

880索引处的解码字符串

题目：

给定一个编码字符串 S。为了找出*解码*字符串并将其写入磁带，从编码字符串中每次读取一个字符，并采取以下步骤：

* 如果所读的字符是字母，则将该字母写在磁带上。
* 如果所读的字符是数字（例如 d），则整个当前磁带总共会被重复写 d-1 次。

现在，对于给定的编码字符串 S 和索引 K，查找并返回解码字符串中的第 K 个字母。

**示例 1：**

**输入：**S = "leet2code3", K = 10

**输出：**"o"

**解释：**

解码后的字符串为 "leetleetcodeleetleetcodeleetleetcode"。

字符串中的第 10 个字母是 "o"。

**示例 2：**

**输入：**S = "ha22", K = 5

**输出：**"h"

**解释：**

解码后的字符串为 "hahahaha"。第 5 个字母是 "h"。

**示例 3：**

**输入：**S = "a2345678999999999999999", K = 1

**输出：**"a"

**解释：**

解码后的字符串为 "a" 重复 8301530446056247680 次。第 1 个字母是 "a"。

思路：如果模拟题目中的操作进行解码，空间占用过大，一定会通不过。而且，题目只要求了求指定位置的字符，并没有要求所有的字符，所以全部解码没有必要。

参考了官方的解答。思路如下：

比如，对于一个解码了的字符串，appleappleappleappleappleapple，并且要求的索引K=24的话，那么结果和K=4是一样的。因为单词apple的size=5，重复了6次。所以第K个索引和第K%size个索引是一样的。

所以我们使用反向的计算，保持追踪解码字符串的size，如果解码字符串等于一个word重复了d次的时候，我们可以把K变化为K % (word.length).

算法：

首先，找出解码字符串的长度。然后，我们反向查找，保持追踪size，也就是对编码字符串S[0], S[1], ..., S[i]解码后的长度。

如果找到的是一个数字S[i]，那么意味着解码字符串S[0], S[1], ..., S[i-1]的长度应该是size / Integer(S[i])；否则应该是size-1.

1. 代码：**class** Solution(object):
2. **def** decodeAtIndex(self, S, K):
3. """
4. :type S: str
5. :type K: int
6. :rtype: str
7. """
8. size=0
9. **for** c **in** S:
10. **if** c.isdigit():
11. size \*=int(c)
12. **else**:
13. size +=1
14. **for** c **in** reversed(S):
15. K%=size
16. **if** K==0 **and** c.isalpha():
17. **return** c
18. **if** c.isdigit():
19. size /=int(c)
20. **else**:
21. size-=1

901股票价格跨度

题目：编写一个 StockSpanner 类，它收集某些股票的每日报价，并返回该股票当日价格的跨度。今天股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。

例如，如果未来7天股票的价格是 [100, 80, 60, 70, 60, 75, 85]，那么股票跨度将是 [1, 1, 1, 2, 1, 4, 6]。

**示例：**

**输入：**

["StockSpanner","next","next","next","next","next","next","next"],[[],[100],

[80],[60],[70],[60],[75],[85]]

**输出：**[null,1,1,1,2,1,4,6]

**解释：**

首先，初始化 S = StockSpanner()，然后：

S.next(100) 被调用并返回 1，

S.next(80) 被调用并返回 1，

S.next(60) 被调用并返回 1，

S.next(70) 被调用并返回 2，

S.next(60) 被调用并返回 1，

S.next(75) 被调用并返回 4，

S.next(85) 被调用并返回 6。

注意 (例如) S.next(75) 返回 4，因为截至今天的最后 4 个价格

(包括今天的价格 75) 小于或等于今天的价格。

**提示：**

1. 调用 StockSpanner.next(int price) 时，将有 1 <= price <= 10^5。
2. 每个测试用例最多可以调用  10000 次 StockSpanner.next。
3. 在所有测试用例中，最多调用 150000 次 StockSpanner.next。
4. 此问题的总时间限制减少了 50%。

思路：看了数值的范围，可以肯定这个题的时间复杂度必须在O(n)以内，也就是说平均每次next()方法调用的时候，必须在将近O(1)的时间内找到前面多少天的价格是小于等于今天的。

这个题的思路是使用一个栈，这个栈里保存的是当前的价格向后可以找多少天。注意这个栈里存放的内容是严格单调递减的，如果新来的数值大于了栈顶元素，那么就要把栈顶的元素给弹出去，直到当前元素小于栈顶才行。

这样做的好处就是，我们没必要保留较小的元素，只需要知道每个元素前面有几个比它小的数字就行了。因为我们在遍历的过程中，是在找比当前元素小的元素个数，栈顶保留的只有较大的元素和它前面出现的次数，那么就知道了前面比它小的元素个数

代码：

1. **class** StockSpanner(object):
3. **def** \_\_init\_\_(self):
4. self.a=[]
6. **def** next(self, price):
7. """
8. :type price: int
9. :rtype: int
10. """
11. res=1
12. **while** self.a **and** self.a[-1][0]<=price:
13. res += self.a.pop()[1]
14. self.a.append((price,res))
15. **return** res

18. # Your StockSpanner object will be instantiated and called as such:
19. # obj = StockSpanner()
20. # param\_1 = obj.next(price)

907子数组的最小值之和

题目：给定一个整数数组 A，找到 min(B) 的总和，其中 B 的范围为 A 的每个（连续）子数组。

由于答案可能很大，因此**返回答案模 10^9 + 7**。

**示例：**

**输入：**[3,1,2,4]

**输出：**17

**解释：**

**子数组为** [3]，[1]，[2]，[4]，[3,1]，[1,2]，[2,4]，[3,1,2]，[1,2,4]，[3,1,2,4]。

最小值为 3，1，2，4，1，1，2，1，1，1，和为 17。

**提示：**

1. 1 <= A <= 30000
2. 1 <= A[i] <= 30000

思路：对于A中的每一个值A[i]，寻找以它为最小值的子序列的数量。也就是说，我们需要寻找A[i]左侧第一个比它大的值的位置，以及右侧第一个比它大的值的位置。然后就可以计算对应的子序列的数量了。这个过程可以简化为使用一个栈。对于被某个数从栈中弹出的数而言，它右侧第一个比它小的数就是这个数。所以我们可以对所有被弹出的数得到左侧的区间范围和右侧的区间范围。我觉得这是一种非常聪明的做法

代码：