Leetcode（python版） 栈（困难）

42接雨水

题目：

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 **感谢 Marcos** 贡献此图。

**示例:**

**输入:** [0,1,0,2,1,0,1,3,2,1,2,1]

**输出:** 6

思路一：我们观察其中的一个位置，单独考虑这个位置的容量是多少？毫无疑问，我们只要知道，这个位置左边最高的那个边，和右边最高的那个边，两者取小的，然后再减去本身的大小，那么结果就是这个位置的容量。所以，（1）从左向右进行扫描，获取每个位置的左边最高的边。（2）从右向左进行扫描，获取每个位置的右边最高的边。（3）再遍历一边，计算出每个位置的容量，累加，即结果

代码：

1. **class** Solution(object):
2. **def** trap(self, height):
3. """
4. :type height: List[int]
5. :rtype: int
6. """
7. **if** **not** height:
8. **return** 0
9. n, res = len(height), 0
10. left\_max, right\_max = [0] \* n, [0] \* n
11. left\_max[0] = height[0]
12. **for** i **in** range(1, n):
13. left\_max[i] = max(height[i], left\_max[i - 1])
14. right\_max[n - 1] = height[n - 1]
15. **for** i **in** range(n-2, -1, -1):
16. right\_max[i] = max(height[i], right\_max[i + 1])
17. **for** i **in** range(1, n-1):
18. res += min(left\_max[i], right\_max[i]) - height[i]
19. **return** res

思路二：双指针

用双指针法，在左右两端分别设置指针left, right，并且用 left\_max， right\_max 记录走过之后的最长最高的边。例如，left\_max 就是指针left 左边最长的线，right\_max 同理。（1）如果 height[left] < height[right]，就是左指针指向的数字 < 右指针指向的数字（现在操作左指针）, 说明此时，左边已经走过的位置都小于 height[right]，通俗的讲，现在left指针位置的左右两边桶的短板，一定存在于左边，又因为 left\_max记录了当前left位置的左边最高的那个边，所以当前left位置的容量就是只需考虑left\_max ， height[left] 这个两个值的大小即可得出。

问题：为什么说左边已经走过的位置都小于 height[right]？因为两个指针的移动条件是，那个指针小就移动那一个。（2）如果height[left] >= height[right]，同理（1）。直到两个指针相遇。

代码（速度很快）：

1. **class** Solution(object):
2. **def** trap(self, height):
3. """
4. :type height: List[int]
5. :rtype: int
6. """
7. **if** **not** height:
8. **return** 0
9. left\_max = right\_max = res = 0
10. left, right = 0, len(height) - 1
11. **while** left < right:
12. **if** height[left] < height[right]:
13. **if** height[left] < left\_max:
14. res += left\_max - height[left]
15. **else**:
16. left\_max = height[left]
17. left += 1
18. **else**:
19. **if** height[right] < right\_max:
20. res += right\_max - height[right]
21. **else**:
22. right\_max = height[right]
23. right -= 1
24. **return** res

84柱状图中最大的矩形

题目：给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

**示例:**

**输入:** [2,1,5,6,2,3]

**输出:** 10

思路：如果已知height数组是升序的，应该怎么做？

比如1,2,5,7,8

那么就是(1\*5) vs. (2\*4) vs. (5\*3) vs. (7\*2) vs. (8\*1)

也就是max(height[i]\*(size-i))

2、使用栈的目的就是构造这样的升序序列，按照以上方法求解。

但是height本身不一定是升序的，应该怎样构建栈？

比如2,1,5,6,2,3

（1）2进栈。s={2}, result = 0

（2）1比2小，不满足升序条件，因此将2弹出，并记录当前结果为2\*1=2。

将2替换为1重新进栈。s={1,1}, result = 2

（3）5比1大，满足升序条件，进栈。s={1,1,5},result = 2

（4）6比5大，满足升序条件，进栈。s={1,1,5,6},result = 2

（5）2比6小，不满足升序条件，因此将6弹出，并记录当前结果为6\*1=6。s={1,1,5},result = 6

2比5小，不满足升序条件，因此将5弹出，并记录当前结果为5\*2=10（因为已经弹出的5,6是升序的）。s={1,1},result = 10

2比1大，将弹出的5,6替换为2重新进栈。s={1,1,2,2,2},result = 10

（6）3比2大，满足升序条件，进栈。s={1,1,2,2,2,3},result = 10

栈构建完成，满足升序条件，因此按照升序处理办法得到上述的max(height[i]\*(size-i))=max{3\*1, 2\*2, 2\*3, 2\*4, 1\*5, 1\*6}=8<10

代码：

1. **class** Solution(object):
2. **def** largestRectangleArea(self, heights):
3. """
4. :type heights: List[int]
5. :rtype: int
6. """
7. maxArea = 0
8. stack = []
9. i = 0
10. **while** i < len(heights):
11. **if** len(stack) == 0 **or** stack[-1] <= heights[i]:
12. stack.append(heights[i])
13. **else**:
14. count = 0
15. **while** len(stack) > 0 **and** stack[-1] > heights[i]:
16. count += 1
17. maxArea = max(maxArea,stack[-1]\*count)
18. stack.pop()#栈顶元素出栈
19. **while** count > 0:#将当前height入栈
20. count -= 1
21. stack.append(heights[i])
22. stack.append(heights[i])
23. i += 1
24. count = 1
25. **while** len(stack) != 0:
26. maxArea = max(maxArea, stack[-1]\*count)
27. stack.pop()
28. count += 1
29. **return** maxArea

85最大矩形

题目：给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

**示例:**

**输入:**

[

["1","0","1","0","0"],

["1","0","**1**","**1**","**1**"],

["1","1","**1**","**1**","**1**"],

["1","0","0","1","0"]

]

**输出:** 6

思路：如图所示，如果把每一行的1和它上面的1连在一起，那么就可以看成一个个站里的矩形方块。那么我们的最终目的就是找出最大面积的矩形方块，所以就是第84题的做法了，使用单调栈。需要注意的是，我们使用一个height数组，保存到某一层的第i个位置为止，能向上构成的矩形的高度。而且需要对每层都做一个寻找面积的操作，最终选择所有层中能够成矩形面积最大值。

时间复杂度是O(M(N+M))，空间复杂度是O(N)。

代码：

1. **class** Solution(object):
2. **def** maximalRectangle(self, matrix):
3. """
4. :type matrix: List[List[str]]
5. :rtype: int
6. """
7. **if** **not** matrix **or** **not** matrix[0]: **return** 0
8. M, N = len(matrix), len(matrix[0])
9. height = [0] \* N
10. res = 0
11. **for** row **in** matrix:
12. **for** i **in** range(N):
13. **if** row[i] == '0':
14. height[i] = 0
15. **else**:
16. height[i] += 1
17. res = max(res, self.maxRectangleArea(height))
18. **return** res
20. **def** maxRectangleArea(self, height):
21. **if** **not** height: **return** 0
22. res = 0
23. stack = list()
24. height.append(0)
25. **for** i **in** range(len(height)):
26. cur = height[i]
27. **while** stack **and** cur < height[stack[-1]]:
28. w = height[stack.pop()]
29. h = i **if** **not** stack **else** i - stack[-1] - 1
30. res = max(res, w \* h)
31. stack.append(i)
32. **return** res

145二叉树的后序遍历

题目：给定一个二叉树，返回它的 *后序*遍历。

**示例:**

**输入:** [1,null,2,3]

1

\

2

/

3

**输出:** [3,2,1]

**进阶:** 递归算法很简单，你可以通过迭代算法完成吗？

思路：

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** postorderTraversal(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. res, stack = [], [(1, root)]
15. **while** stack:
16. p = stack.pop()
17. **if** **not** p[1]:
18. **continue**
19. **if** p[0] == 0:
20. res.append(p[1].val)
21. **else**:
22. stack.extend([(0, p[1]), (1, p[1].right), (1, p[1].left)])
23. **return** res

224基本计算器

题目：实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式可以包含左括号 ( ，右括号 )，加号 + ，减号 -，**非负**整数和空格  。

**示例 1:**

**输入:** "1 + 1"

**输出:** 2

**示例 2:**

**输入:** " 2-1 + 2 "

**输出:** 3

**示例 3:**

**输入:** "(1+(4+5+2)-3)+(6+8)"

**输出:** 23

思路：

代码：

1. **class** Solution(object):
2. **def** calculate(self, s):
3. """
4. :type s: str
5. :rtype: int
6. """
7. s = "(" + s + ")"
8. stack = []
9. \_stack = []
10. i = 0
11. **while** i < len(s):
12. **if** s[i] == " ":
13. i += 1
14. **elif** s[i] == "(":
15. \_stack.append(len(stack))
16. i += 1
17. **elif** s[i] == ")":
18. start = \_stack.pop()
19. j = start
20. a = stack[j]
21. **while** j + 2 < len(stack):
22. ops = stack[j + 1]
23. **if** ops == "+":
24. a = a + stack[j + 2]
25. **elif** ops == "-":
26. a = a - stack[j + 2]
27. **else**:
28. **return** "invalid"
29. j += 2
30. k = len(stack) - start
31. **while** k > 0:
32. stack.pop()
33. k -= 1
34. stack.append(a)
35. i += 1
36. **elif** s[i] **in** "+-":
37. stack.append(s[i])
38. i += 1
39. **else**:
40. start = i
41. **while** i < len(s) **and** s[i] **not** **in** "-+() ":
42. i += 1
43. num = int(s[start:i])
44. stack.append(num)
45. **return** stack[0]

316去除重复字母

题目：给定一个仅包含小写字母的字符串，去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

**示例 1:**

**输入:** "bcabc"

**输出:** "abc"

**示例 2:**

**输入:** "cbacdcbc"

**输出:** "acdb"

思路：先用字典计算每个字母出现多少次，然后依次压入栈中，如果后来的字母比现在栈顶字母小，且栈顶字母后面还会出现，则弹出栈顶

代码：

1. **class** Solution(object):
2. **def** removeDuplicateLetters(self, s):
3. """
4. :type s: str
5. :rtype: str
6. """
7. remaining={}
8. **for** c **in** s:
9. remaining[c]=remaining.get(c, 0) + 1
11. in\_stack,stk=set(),[]
12. **for** c **in** s:
13. **if** c **not** **in** in\_stack:
14. **while** stk **and** stk[-1]>c **and** remaining[stk[-1]]:
15. in\_stack.remove(stk.pop())
16. stk+=c
17. in\_stack.add(c)
18. remaining[c]-=1
19. **return** "".join(stk)

591标签验证器

题目：给定一个表示代码片段的字符串，你需要实现一个验证器来解析这段代码，并返回它是否合法。合法的代码片段需要遵守以下的所有规则：

1. 代码必须被**合法的闭合标签**包围。否则，代码是无效的。
2. **闭合标签**（不一定合法）要严格符合格式：<TAG\_NAME>TAG\_CONTENT</TAG\_NAME>。其中，<TAG\_NAME>是起始标签，</TAG\_NAME>是结束标签。起始和结束标签中的 TAG\_NAME 应当相同。当且仅当 TAG\_NAME 和 TAG\_CONTENT 都是合法的，闭合标签才是**合法的**。
3. **合法的** TAG\_NAME 仅含有**大写字母**，长度在范围 [1,9] 之间。否则，该 TAG\_NAME 是**不合法的**。
4. **合法的** TAG\_CONTENT 可以包含其他**合法的闭合标签**，**cdata** （请参考规则7）和任意字符（注意参考规则1）**除了**不匹配的<、不匹配的起始和结束标签、不匹配的或带有不合法 TAG\_NAME 的闭合标签。否则，TAG\_CONTENT 是**不合法的**。
5. 一个起始标签，如果没有具有相同 TAG\_NAME 的结束标签与之匹配，是不合法的。反之亦然。不过，你也需要考虑标签嵌套的问题。
6. 一个<，如果你找不到一个后续的>与之匹配，是不合法的。并且当你找到一个<或</时，所有直到下一个>的前的字符，都应当被解析为 TAG\_NAME（不一定合法）。
7. cdata 有如下格式：<![CDATA[CDATA\_CONTENT]]>。CDATA\_CONTENT 的范围被定义成 <![CDATA[ 和**后续的第一个** ]]>之间的字符。
8. CDATA\_CONTENT 可以包含**任意字符**。cdata 的功能是阻止验证器解析CDATA\_CONTENT，所以即使其中有一些字符可以被解析为标签（无论合法还是不合法），也应该将它们视为**常规字符**。

**合法代码的例子:**

**输入:** "<DIV>This is the first line <![CDATA[<div>]]></DIV>"

**输出:** True

**解释:**

代码被包含在了闭合的标签内： <DIV> 和 </DIV> 。

TAG\_NAME 是合法的，TAG\_CONTENT 包含了一些字符和 cdata 。

即使 CDATA\_CONTENT 含有不匹配的起始标签和不合法的 TAG\_NAME，它应该被视为普通的文本，而不是标签。

所以 TAG\_CONTENT 是合法的，因此代码是合法的。最终返回True。

**输入:** "<DIV>>> ![cdata[]] <![CDATA[<div>]>]]>]]>>]</DIV>"

**输出:** True

**解释:**

我们首先将代码分割为： start\_tag|tag\_content|end\_tag 。

start\_tag -> **"<DIV>"**

end\_tag -> **"</DIV>"**

tag\_content 也可被分割为： text1|cdata|text2 。

text1 -> **">> ![cdata[]] "**

cdata -> **"<![CDATA[<div>]>]]>"** ，其中 CDATA\_CONTENT 为 **"<div>]>"**

text2 -> **"]]>>]"**

start\_tag **不**是 **"<DIV>>>"** 的原因参照规则 6 。

cdata **不**是 **"<![CDATA[<div>]>]]>]]>"** 的原因参照规则 7 。

**不合法代码的例子:**

**输入:** "<A> <B> </A> </B>"

**输出:** False

**解释:** 不合法。如果 "<A>" 是闭合的，那么 "<B>" 一定是不匹配的，反之亦然。

**输入:** "<DIV> div tag is not closed <DIV>"

**输出:** False

**输入:** "<DIV> unmatched < </DIV>"

**输出:** False

**输入:** "<DIV> closed tags with invalid tag name <b>123</b> </DIV>"

**输出:** False

**输入:** "<DIV> unmatched tags with invalid tag name </1234567890> and <CDATA[[]]> </DIV>"

**输出:** False

**输入:** "<DIV> unmatched start tag <B> and unmatched end tag </C> </DIV>"

**输出:** False

**注意:**

1. 为简明起见，你可以假设输入的代码（包括提到的**任意字符**）只包含数字, 字母, '<','>','/','!','[',']'和' '。

思路：

代码：

1. **class** Solution(object):
2. **def** isValid(self, code):
3. """
4. :type code: str
5. :rtype: bool
6. """
7. stack = []
8. i = 0
9. n = len(code)
10. **while**(i<n):
11. **if** stack == [] **and** i!=0:
12. **print** 'stack empty'
13. **return** False
14. **if** code[i:i+9] == "<![CDATA[":
15. i = code.find("]]>",i+9)
16. **if** i == -1:
17. **print** 'cdata'
18. **return** False
19. i+=2
20. i+=1
21. **continue**
22. **if** code[i:i+2] == "</":
23. j = code.find(">",i+2)
24. **if** j == -1:
25. **print** 'not find'
26. **return** False
27. **if** stack != [] **and** stack[-1] == code[i+2:j]:
28. stack.pop()
29. i = j
30. **else**:
31. **print** 'pop'
32. **return** False
33. #print i,n
34. i+=1
35. **continue**
36. **if** code[i] == "<":
37. j = code.find(">",i+1)
38. **if** j == -1 **or** j==i+1 **or** j-i>10:
39. **print** "length"
40. **return** False
41. **for** t **in** code[i+1:j]:
42. **if** ord(t) < ord('A') **or** ord(t) > ord('Z'):
43. **print** "upper"
44. **return** False
45. stack.append(code[i+1:j])
46. i = j
47. i += 1
48. #print stack
49. **if** stack != []:
50. **print** 'empty'
51. **return** False
52. **return** True

726原子的数量

题目：给定一个化学式formula（作为字符串），返回每种原子的数量。

原子总是以一个大写字母开始，接着跟随0个或任意个小写字母，表示原子的名字。

如果数量大于 1，原子后会跟着数字表示原子的数量。如果数量等于 1 则不会跟数字。例如，H2O 和 H2O2 是可行的，但 H1O2 这个表达是不可行的。

两个化学式连在一起是新的化学式。例如 H2O2He3Mg4 也是化学式。

一个括号中的化学式和数字（可选择性添加）也是化学式。例如 (H2O2) 和 (H2O2)3 是化学式。

给定一个化学式，输出所有原子的数量。格式为：第一个（按字典序）原子的名子，跟着它的数量（如果数量大于 1），然后是第二个原子的名字（按字典序），跟着它的数量（如果数量大于 1），以此类推。

**示例 1:**

**输入:**

formula = "H2O"

**输出:** "H2O"

**解释:**

原子的数量是 {'H': 2, 'O': 1}。

**示例 2:**

**输入:**

formula = "Mg(OH)2"

**输出:** "H2MgO2"

**解释:**

原子的数量是 {'H': 2, 'Mg': 1, 'O': 2}。

**示例 3:**

**输入:**

formula = "K4(ON(SO3)2)2"

**输出:** "K4N2O14S4"

**解释:**

原子的数量是 {'K': 4, 'N': 2, 'O': 14, 'S': 4}。

**注意:**

* 所有原子的第一个字母为大写，剩余字母都是小写。
* formula的长度在[1, 1000]之间。
* formula只包含字母、数字和圆括号，并且题目中给定的是合法的化学式。

思路：看到括号匹配，也会让人立马想到栈，其实DFS本身就是栈实现的，所以也完全可以用栈来解决。

方法是，左括号进栈，然后把字母依次进栈，当遇到右括号的时候，需要对栈进行退栈操作，这个时候要统计每个元素的次数，当退栈的时候遇到左括号，说明内部的分子团已经结束，那么把遇到的第一个左括号退栈，把内部的分子团的各个元素和其个数进栈。然后遍历就好了！这个方法的好处是，当最后遍历结束的时候，栈里面保存的只剩下了已经统计好了的各个元素和其个数的对应，每个元素只会出现一次，相当于已经做了元素的求和操作，最后只需要排序即可。

为了方便，我把分子式用括号包了起来，方便栈操作的判断。

这个题做了很久，主要是查一个bug，查了一个小时，感觉很诡异。其实仔细对比一下和上面DFS的解法，大同小异。区别是我用字母n保存了分子式的长度，然后下面退栈的for循环中又使用了n这个变量名称！！由于python不用声明变量，所以直接把外边的n覆盖掉了！！做法很简单，把内部for循环里的变量名改一下就好了！生气！！

最坏的时间复杂度是O(N！)，最优时间复杂度是O(N)，空间复杂度是O(Ｎ)。其中N是分子的长度

代码：

1. **class** Solution(object):
2. **def** countOfAtoms(self, formula):
3. """
4. :type formula: str
5. :rtype: str
6. """
7. stack = list()
8. formula = "(" +  formula + ")1"
9. i = 0
10. n = len(formula)
12. **while** i < n:
13. **if** i >= n: **continue**
14. **if** formula[i] == "(":
15. stack.append("(")
16. i += 1
17. **elif** formula[i] == ")":
18. parentNum = 0
19. i += 1
20. **while** i < n **and** formula[i].isdigit():
21. parentNum = 10 \* parentNum + int(formula[i])
22. i += 1
23. count = collections.Counter()
24. **while** stack[-1] != "(":
25. atom, atomNum = stack.pop()
26. count[atom] += atomNum \* parentNum
27. **if** stack[-1] == "(":
28. stack.pop()
29. **for** c, t **in** count.items(): # 刚开始把变量t写成了n!!错了很多次
30. stack.append((c, t))
31. **elif** formula[i].isalpha():
32. atom = formula[i]
33. atomNum = 0
34. i += 1
35. **while** i < n **and** formula[i].isalpha() **and** formula[i].islower():
36. atom += formula[i]
37. i += 1
38. **while** i < n **and** formula[i].isdigit():
39. atomNum = 10 \* atomNum + int(formula[i])
40. i += 1
41. atomNum = 1 **if** atomNum == 0 **else** atomNum
42. stack.append((atom, atomNum))

45. res = ""
46. **for** atoms **in** sorted(stack):
47. **if** atoms == "(":
48. **continue**
49. c, n = atoms
50. **if** n == 1:
51. res += c
52. **else**:
53. res += c + str(n)
54. **return** res

770基本计算器IV

题目：

思路：

代码：

895最大频率栈

题目：实现 FreqStack，模拟类似栈的数据结构的操作的一个类。

FreqStack 有两个函数：

* push(int x)，将整数 x 推入栈中。
* pop()，它**移除**并返回栈中出现最频繁的元素。
  + 如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。

**示例：**

**输入：**

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

**输出：**[null,null,null,null,null,null,null,5,7,5,4]

**解释：**

执行六次 .push 操作后，栈自底向上为 [5,7,5,7,4,5]。然后：

pop() -> 返回 5，因为 5 是出现频率最高的。

栈变成 [5,7,5,7,4]。

pop() -> 返回 7，因为 5 和 7 都是频率最高的，但 7 最接近栈顶。

栈变成 [5,7,5,4]。

pop() -> 返回 5 。

栈变成 [5,7,4]。

pop() -> 返回 4 。

栈变成 [5,7]。

**提示：**

* 对 FreqStack.push(int x) 的调用中 0 <= x <= 10^9。
* 如果栈的元素数目为零，则保证不会调用  FreqStack.pop()。
* 单个测试样例中，对 FreqStack.push 的总调用次数不会超过 10000。
* 单个测试样例中，对 FreqStack.pop 的总调用次数不会超过 10000。
* 所有测试样例中，对 FreqStack.push 和 FreqStack.pop 的总调用次数不会超过 150000。

思路：同时优化两个目标：出现的频率和出现的索引。所以天然想到用优先级队列。python的优先级队列是个最小堆，而我们要优化的目标是求最大，因此，使用负号即可。

使用m保存出现的次数，使用index保存索引，使用q表示堆。

把出现的次数和出现的索引取反进堆，这样每次弹出堆的时候都是把这两个目标优化了的。pop的时候要更新频率。当每次Pop的时候会把各个字符出现的频率恢复到它入堆前的样子（题目给出了如果同样的频率时，弹出最后push进去的数字）。当这个数字是最大频率数字，并且多次出现的时候，尽可能弹出靠最后的进入数字保证了提前进入堆的那些数字的频率是正确的。堆的平均时间复杂度是O(1)，空间复杂度是O(N)。

代码：

1. **class** FreqStack(object):
3. **def** \_\_init\_\_(self):
4. self.m = collections.defaultdict(int)
5. self.q=[]
6. self.index=0
8. **def** push(self, x):
9. """
10. :type x: int
11. :rtype: void
12. """
13. self.m[x]+=1
14. heapq.heappush(self.q,(-self.m[x],-self.index,x))
15. self.index +=1
17. **def** pop(self):
18. """
19. :rtype: int
20. """
21. val=heapq.heappop(self.q)[2]
22. self.m[val]-=1
23. **return** val

26. # Your FreqStack object will be instantiated and called as such:
27. # obj = FreqStack()
28. # obj.push(x)
29. # param\_2 = obj.pop()