第 15 篇

滑动窗口的最大值

问题

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}；针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

思路

详细思路可参照 https://blog.csdn.net/qq_41822235/article/details/82891762

基本思路，从左到右滑过每个窗口，在每个窗口里寻找最大值并记录下来。一点改进，诸如记住当前最大值的下标，到新窗口里，如果根据下标判断该最大值仍然在新窗口，则只比较新窗口比上一个窗口新加入的那1个元素和最大值的大小，如果不在的话则要找遍新窗口才能找出最大，可以减少一些比较次数。

上面方法需要比较的次数是 size x n，可以进一步减少比较次数。

滑动窗口可以用一个队列来记录，新值进入窗口则加入队列，旧值滑出窗口则移出队列。但这里采用可以从两端删除元素的队列，来减少比较次数。目的是，队列头部始终存储队列中最大的元素。

当新的元素到来时，首先判断队尾元素是否比它小，若小，则剔除当前队尾元素，一直重复下去，直到队尾元素比新元素大，或者队列空了。然后新值加到队尾。然后判断当前队头是否下标超出窗口，如果超出，则队头移出。完成上述操作后，当前队头就是当前窗口的最大值。

代码

按基本思路（size x n）

```
# -*- coding:utf-8 -*-
class Solution:
    def maxInWindows(self, num, size):
        # write code here
        length = len(num)
        if size <= 0 or size > length:
            return []
        i = 0
        ans =[max(num[:size])]
        while size + i < length:
            if num[i] < ans[-1]:
                ans.append(max([ans[-1], num[i+size]]))
            else:
```

```
                ans.append(max(num[i+1:i+size+1]))
            i += 1
        return ans
```

双向队列写法：

```
# -*- coding:utf-8 -*-
class Solution:
    def maxInWindows(self, num, size):
        # write code here
        ans = []
        length = len(num)
        if size <= 0 or size > length or not num:
            return []
        deque = []  # 队列里存的是下标
        for i in range(0, len(num)):
            while(deque and num[deque[-1]] < num[i]):
                deque.pop()
            deque.append(i)
            if i - deque[0] + 1 > size:
                deque.pop(0)
      if i >= k - 1:
              ans.append(num[deque[0]])
        return ans
```

矩阵中的路径

问题

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则之后不能再次进入这个格子。 例如 a b c e s f c s a d e e 这样的3 X 4 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

思路

使用回溯法。首先遍历全部数据，找到和字符串第一个字符相同的位置，以此为起点，上下左右前进寻找和下一个字符相同的位置行。如果某处上下左右都找不到，则回退到上一步。非常适合使用递归的思路，而且代码也比较容易懂，详见下面代码（这段没有自己写，直接搬运了一份）。

代码

```
# -*- coding:utf-8 -*-
class Solution:
    def hasPath(self, matrix, rows, cols, path):
        # write code here
```

```
        if not matrix and rows <= 0 and cols <= 0 and path == None:
            return False
        # 模拟的字符矩阵
        markmatrix = [0] * (rows * cols)
        pathlength = 0
        # 从第一个开始递归，当然第一二个字符可能并不位于字符串之上，所以有这样一个双层循环找起点用的，一旦找到第一个符合的字符串，就开始进入递归，
        # 返回的第一个return Ture就直接跳出循环了。
        for row in range(rows):
            for col in range(cols):
                if self.hasPathCore(matrix, rows, cols, row, col, path,
pathlength, markmatrix):
                    return True
        return False

    def hasPathCore(self, matrix, rows, cols, row, col, path, pathlength,
markmatrix):
        # 说明已经找到该路径，可以返回True
        if len(path) == pathlength:
            return True

        hasPath = False
        if row >= 0 and row < rows and col >= 0 and col < cols and matrix[row
* cols + col] == path[pathlength] and not \
                markmatrix[row * cols + col]:
            pathlength += 1
            markmatrix[row * cols + col] = True
            # 进行该值上下左右的递归
            hasPath = self.hasPathCore(matrix, rows, cols, row - 1, col, path,
pathlength, markmatrix) \
                      or self.hasPathCore(matrix, rows, cols, row, col - 1,
path, pathlength, markmatrix) \
                      or self.hasPathCore(matrix, rows, cols, row + 1, col,
path, pathlength, markmatrix) \
                      or self.hasPathCore(matrix, rows, cols, row, col + 1,
path, pathlength, markmatrix)

            # 对标记矩阵进行布尔值标记
            if not hasPath:
                pathlength -= 1
                markmatrix[row * cols + col] = False
        return hasPath
```

机器人的运动范围

问题

地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子?

思路

思路仍然是回溯法，但判断条件有了变化。

代码

```
# -*- coding:utf-8 -*-
class Solution:
    def movingCount(self, threshold, rows, cols):
        # write code here
        markmatrix = [False] * (rows * cols)
        count = self.GetNum(threshold, rows, cols, 0, 0, markmatrix)
        return count

    def GetNum(self, threshold, rows, cols, row, col, markmatrix):
        count = 0

        if self.GetSum(threshold, rows, cols, row, col, markmatrix):
            markmatrix[row * cols + col] = True
            count = 1 + self.GetNum(threshold, rows, cols, row - 1, col,
markmatrix) + \
                    self.GetNum(threshold, rows, cols, row, col - 1,
markmatrix) + \
                    self.GetNum(threshold, rows, cols, row + 1, col,
markmatrix) + \
                    self.GetNum(threshold, rows, cols, row, col + 1,
markmatrix)
        return count

    def GetSum(self, threshold, rows, cols, row, col, markmatrix):
        if row >= 0 and row < rows and col >= 0 and col < cols and
self.getDigit(row) + self.getDigit(
                col) <= threshold and not markmatrix[row * cols + col]:
            return True
        return False

    def getDigit(self, number):
        sumNum = 0
        while number > 0:
            sumNum += number % 10
            number = number // 10
        return sumNum
```

本文稿来自 https://github.com/dox1994/offer-coding-interviews-python，欢迎前来给个star🌟～
如有错误或遗漏欢迎issue～