算法总结-单调栈

原创 聿亍 聿亍 2019-12-21

栈是非常常用的数据结构，最重要的特性就是**先进后出**。比如在操作系统中，函数调用就是通过栈来实现的。同样栈在一些常用的算法中也有很多应用，比如深度优先搜索，字符串处理(尤其是表达式处理， 括号匹配)等。

这一篇要讲的是使用栈实现的一种更有效的数据结构，**单调栈**。首先讲一下单调栈的基本形式，基本题型以及一些复杂的应用，最后提及一点点类似的结构**单调队列**。

顾名思义，单调栈中维护着一个元素单调的序列，以升序序列为例，一般具有如下范式：

```
stack = []
for x in nums:
    while stack and stack[-1] > x:
        stack.pop()
    stack.append(x)
```

上述代码的思路就是在 x 入栈之前将栈中大于 x 的元素全部弹出，以此来维护栈中元素的(非严格)升序性，注意上述框架中我们存入栈中的是元素本身的值，但更多的情况下，我们会存入元素对应的下标，接下来题目可以看出这样做更加自然和便利。

我们先来看单调栈可以用来做什么?

> LeetCode 496. 给两个数组，求 A 中元素在 B 中的 nextGreaterElement。

我们可以直接暴力解这个题目，时间复杂度是 $O(mn)$ m, n 分布对应两个数组的长度，但使用单调栈可以在线性时间内求解。

先对第二个数组进行处理，使用单调栈维护一个降序的序列，每次我们看到一个比栈顶元素大的元素 x, 则弹出栈中所有比x小的元素，所有被弹出元素的 nextGreaterElement 就是 x 。然后使用一个 hash 表保存所有 nextGreaterElement 元素对。最后对第一个数组进行一般查找就可以了，代码如下。

```
class Solution_496:
    def nextGreaterElement(self, findNums, nums):
        stack, d = [], {}
        for x in nums:
            while stack and stack[-1] < x:
                d[stack.pop()] = x
```

```
            stack.append(x)
        return [d.get(x, -1) for x in findNums]
```

> LeetCode 503. 求循环数组内的元素的 nextGreaterElement。

这个题目跟上面差不多是一样的，只是现在是对一个循环数组，所以把数组拼接起来查找一遍或者循环两遍就可以了。

```
class Solution_503:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        # 做两遍循环也是一样的
        res = [-1] * len(nums)
        stack = []
        for i in range(2):
            for i, x in enumerate(nums):
                while stack and nums[stack[-1]] < x:
                    t = stack.pop()
                    res[t] = x
                stack.append(i)
        return res
```

由上面例子可以看出，单调栈适合用来求 **前一小元素**(Previous Less Element)，**后一小元素**(Next Less Element)，**前一大元素**(Previous Greater Element)，**后一大元素**(Next Greater Element)问题。并且利用这个性质，可以求解很多有趣的问题。

> LeetCode 907. 给定一个数组 A, 找出其所有非空子集的最小值之和。

显然对于长度为 `n` 的数组，直接暴力枚举所有子集的情况复杂度是 $O(2^n)$，这个题是关键在于对于 `A` 中的每一个元素都会在最后的结果中被累加至少一次（取决于这个元素是多少个子集的最小值）。设 `A[i]` 是 `f(i)` 个子集的最小值，则有 `ans = sum(A[i] * f(i))`，这样原问题转变为了求 `f(i)` 的问题。

由上面知道，单调栈可以用来求解前一小(PLE)和后一小(NLE)，我们可以利用这个性质求 `f(i)`。具体用一个例子来说明。

```
A = [2, 8, 7, 3, 4, 6, 9, 1]
#    ^        ^           ^
#    0        3           7
```

上述例子中对于元素 3 (index=3)，其前一个小的元素是 2 (index=0), 后一小的元素是 1 (index=7)。我们用 `left[i]` 记录 `A[i]` 到它的前一小元素的距离，`right[i]` 记录 `A[i]` 到它后一小元素距离。这里 `left[3] = 3` ，`right[3] = 4` 。

我们来看一下以 3 为最小元素的非空子集有那些。

```
#           3           # 只有 3
#        7, 3           # 只有左边
#     8, 7, 3
#           3, 4        # 只有右边
#           3, 4, 6
#           3, 4, 6, 9
#        7, 3, 4        # 左右都有
#        7, 3, 4, 6
#        7, 3, 4, 6, 9
#     8, 7, 3, 4
#     8, 7, 3, 4, 6
#     8, 7, 3, 4, 6, 9
```

对应共 12 个子集，其中:

- 只包含 3 的子集 `[3]` ，共 1 个，
- 包含 3 和到前一小之间的元素的子集有 `[7, 3]`, `[8, 7, 3]` 共 2 个
- 包含 3 和到后一小之间的元素的子集有 `[3, 4]`, `[3, 4, 6]`, `[3, 4, 6, 9]` , 共 3 个
- 即包含到前一小又包含到后一小的元素的子集有 6 个。

于是我们可以得到

```
f(i) = 1 + (left[i]-1) + (right[i]-1) + (left[i]-1) * (right[i]-1)
     = left[i] * right[i]
```

于是问题就可以解决了，值得注意的是，对于数组中元素有重复的情况，求前一小的时候是我们使用小于，求后一小使用小于等于，体现在代码中就是 left 和 right 的赋值一个在 stack 循环内，一个在循环外，代码如下。

```
class Solution_907:
    def sumSubarrayMins(self, A: List[int]) -> int:
        n = len(A)
        # 初始化
        left = [i + 1for i in range(n)]
        right = [n - i for i in range(n)]
        # 求到前一个小的元素的距离
        stack = []
        for i, x in enumerate(A):
            while stack and A[stack[-1]] > x:
                stack.pop()
```

```
        left[i] = i - stack[-1] if stack else i + 1
        stack.append(i)
    # 求到后一个小的元素的距离
    # 注意求前一个小的元素是严格的小于，而求后一个小于是小于等于，
    # 体现在上下两段代码中就是两端代码中就是对 left 和 right 赋值的过程，
    # 区别就是赋值在 stack 循环内外，而比较用到都是大于号（A[stack[-1]] > x）
    stack = []
    for i, x in enumerate(A):
        while stack and A[stack[-1]] > x:
            t = stack.pop()
            right[t] = i - t
        stack.append(i)

    ans = 0
    MOD = int(1e9 + 7)
    for x, l, r in zip(A, left, right):
        ans = (ans + x * l * r) % MOD
    return ans
```

注意上面代码中，求 left 和 right 的逻辑是一样的，仅仅是赋值的时候不同，所以可以将其合并到一个循环中。

1. right 的赋值是在 stack 的循环内进行的, 每次对 pop 出来下标对应的元素计算 right 所以不用改动。
2. left 的赋值在上面的解法中是按顺序进行，我们如果对每次 pop 出来下标对应元素计算 left 应该怎么算呢？很简单，栈式升序有序的，弹出后下一个就是栈顶就是弹出元素的前一小，当然如果栈为空了就是弹出元素的下标加一

最后，既然求 left, right 可以在 one-pass 做完，那么求最后结果也可以合并进来，最后的代码如下。

```
class Solution_907:
    def sumSubarrayMins(self, A: List[int]) -> int:

        ans = 0
        MOD = int(1e9 + 7)
        n = len(A)
        stack = []
        for i in range(n + 1):
            while stack and A[stack[-1]] > (A[i] if i < n else0):
                t = stack.pop()
                k = stack[-1] if stack else-1
                ans += A[t] * (i - t) * (t - k)  # right[t] = i - t, left[t] = t - k
            stack.append(i)
        return ans % MOD
```

> LeetCode 84. 求直方图中最大的矩形面积

这时经典题目，注意到可以用区间查询，所以这个题目可以使用线段树/树形数组来做，不过这里要用单调栈来做。注意到，这个题目其实是和上面题目几乎一样的。要求一个最大的矩形，自然是对于

一个 bar 找到前一小和后一小，然后这中间间距乘与矩形高度就是面积了。也即是，这道题目的结果 `ans = max(A[i] * (left + right - 1))` 。即把上面的求和过程变成求最大值了。代码如下。

```
class Solution_84:
    def largestRectangleArea(self, heights: 'List[int]') -> int:
        # 解法一，使用线段树
        pass

    def largestRectangleArea(self, heights: List[int]) -> int:
        # 这个解法和 907 题的 one-pass 解法几乎是一样的
        # 找出每个数的前一小元素与后一小元素，然后面积是左右之间的距离乘于当前块的高度
        stack = []
        ans = 0
        n = len(heights)
        heights.append(0)
        for i in range(n + 1):  # 注意循环次数多了 1 次
            # 循环多了 i==n 的情况，同时判断 height[stack[-1]] > 0 (heights 全为非负数)
            # 是将上面解法中 最后的判断 stack 内的合并到了一起。
            while stack and heights[stack[-1]] > heights[i]:
                t = stack.pop()
                k = stack[-1] if stack else-1
                # 下面 ans = max(ans, heights[t] * ((i - t) + (t - k) - 1)) 一句中没有化简求和
                ans = max(ans, heights[t] * ((i - t) + (t - k) - 1))
            stack.append(i)
        return ans
```

直方图矩形面积的题与下面一道题有着相似的题目形式，但是解法不太一致。

> LeetCode 42. 求最大的积雨面积

这个题目自然的思路就是使用 left 存左边的最高值，right 存右边最高值，然后判断每一个位置是否会积水以及累计积水量。这样至少需要两遍循环以及 $O(n)$ 的空间。最优的解是使用双指针法，可以做到 One-pass 以及常量空间。代码如下。

```
class Solution_42:
    def trap(self, height: List[int]) -> int:
        # two pointers, O(n) time, O(1) space
        # 不使用额外的数组空间维护左/右侧的最大值，而是使用双指针法来做
        lo, hi = 0, len(height) - 1
        res = 0
        lm = rm = 0# 左侧最大值和右侧的最大值
        while lo < hi:
            # height[lo] <= height[hi] 表示右边当前值高于或等于左边，这样我们如果左边 lm > height[
            if height[lo] <= height[hi]:
                lm = max(lm, height[lo])
                res += lm - height[lo]  # lm - height[lo] >= 0
                lo += 1
```

```
            # 同理，height[lo] > height[hi] 表示左边当前值高于右边，这样我们如果右边 rm > height[
            else:
                rm = max(rm, height[hi])
                res += rm - height[hi]
                hi -= 1
        return res
```

同样这个题目也可以使用单调栈来解，注意单调栈中元素都是有序的，使用栈维护降序序列，那么出栈的情况就是对应当前元素大于栈顶，而且下一个栈顶（如果有的话）是大于等于栈顶的，那么栈顶对应的位置有可能积雨（都大于的情况下），于是我们可以使用这个思路对每一块一层层的计算总的面积。代码如下。

```
class Solution_42:
    def trap(self, height: List[int]) -> int:
        stack = []
        res = 0
        for i, h in enumerate(height):
            # 使用 stack 维护降序序列，这个对于一块连着的积水时一层一层算的。
            while stack and height[stack[-1]] < h:
                t = stack.pop()
                if stack:
                    min_h = min(height[stack[-1]], h)
                    res += (min_h - height[t]) * (i - stack[-1] - 1)
            stack.append(i)
        return res
```

LeetCode 42. 是与 LeetCode 84. 做对比的话，LeetCode 85. 就是 84 题的扩展，85 题可以变形到 84 题求直方图面积。

> LeetCode 85. 给定一个只含有 "0", "1" 的矩阵，求出其中 "1" 的最大矩形面积

第一眼看到这个题目的时候很显然会考虑一个连通图问题，但实际上不是，这是一个 DP 和单调栈的题目。我们来看怎么把他变形到求直方图面积。

具体来说就是，我们将列中的 `'1'` 看作一个单位的小方格，然后维护一个 heights 数组，这个数组表示扫描到当前位置时累积的每一列的高度，因为要求的是矩形是一个连续区域，于是我们在遇到 `'0'` 时 reset 当前列的 heights 值。这一部分是 DP 过程。

$$height[j] = \begin{cases} = height[j] + 1 \ ; \text{if matrix[i][j]} == \text{'1'} \\ = 0 \ ; \text{if matrix[i][j]} == \text{'0'} \end{cases}$$

举个例子来说明，一个 matrix 如下：

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

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

扫描第一行后 `heights = [1, 0, 1, 0, 0]` 扫描第二行后 `heights = [2, 0, 2, 1, 1]` 扫描第三行后 `heights = [0, 1, 3, 2, 2]` 注意第 0 列的高度被重置为 0 扫描第四行后 `heights = [1, 0, 0, 3, 0]` 注意第 1, 2, 4 列高度被重置为 0

每扫描一个元素，我们都更新一次 heights(上面只列出来扫描完一行后的)，更新 heights 的同时我们可以解LeetCode 84. Largest Rectangle in Histogram 问题。最后得到的就是问题的解。代码如下。

```
class Solution_85:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        m = len(matrix)
        n = len(matrix[0]) if m else0
        if m == 0or n == 0:
            return0
        res = 0
        heights = [0] * (n + 1)
        for i in range(m):
            stack = []
            for j in range(n + 1):
                # 更新 heights
                if j < n:
                    heights[j] = heights[j] + 1 if matrix[i][j] == '1' else 0
                # 计算 面积
                while stack and heights[stack[-1]] >= heights[j]:
                    t = stack.pop()
                    k = stack[-1] if stack else -1
                    res = max(res, heights[t] * (j - k - 1))
                stack.append(j)
        return res
```

> LeetCode 402. 给定一个数（串表示）和 k, 删除串中 k 个字符得到最小的数字（串表示）。

这个题当初保研机试的时候的第四题，当时没有做出来，其实有个很简单的思路，k 次扫描，每次遇到前一个比后一个字符大，前一个大的字符删除，然后再重新开始扫描，这样每次删除一个字符，总共删除 k 个，时间复杂度是 $O(kn)$ 。然后使用单调栈可以在 $O(n)$ 时间内解这道题。

使用单调栈尽可能的维护一个尽可能的升序序列，即如果栈顶（对应前面的元素）比当前大的话，同时弹出数目少于 k 个的话，就弹出这个这个元素（对应删除这个字符），最终检查弹出的数目是否到 k 个，不到的话继续从栈顶弹出元素。最后就是要注意，数字串不能有前导零，所以要把前置 `'0'` 给删除，同时数字 `"0"` 要保留。

```
class Solution_402:
    def removeKdigits(self, num: str, k: int) -> str:
```

```
        stack = []
        cnt = 0
        for i, c in enumerate(num):
            while stack and ord(stack[-1]) > ord(c) and cnt < k:
                stack.pop()
                cnt += 1
            stack.append(c)
        while cnt < k and stack:
            stack.pop()
            cnt += 1
        res = ''.join(stack).lstrip('0')
        return res if res else'0'
```

> LeetCode 316. 给定一个字符串，删除重复的字符同时保持原相对顺序不变，得到字典最小的串

这个题是上面的最小数字的扩展版，这里进一步要求每一个字符只出现一次，并保持原有字符的相对顺序。所以我们在使用单调栈保留结果的同时，使用另一个表记录这个字符是否已经在栈中，确保每个字符至多出现一次，同时也要对每个字符计数，确保每个字符至少出现一次。

```
class Solution_316:
    def removeDuplicateLetters(self, s: str) -> str:
        # 使用栈维护一个字符的升序序列(尽可能的升序，因为要保持原本的相对顺序)
        stack = []
        cnt = {}
        seen = {}
        for c in s:
            cnt[c] = cnt.get(c, 0) + 1
            seen[c] = False
        for c in s:
            cnt[c] -= 1# cnt 表示后面还剩多少个每个字符
            if seen[c]:  # 栈里面已经有 c 了，则 c 一定处在一个相对升序序列中
                continue
            while stack and ord(stack[-1]) > ord(c) and cnt[stack[-1]] > 0:
                t = stack.pop()
                seen[t] = False
            stack.append(c)
            seen[c] = True
        return''.join(stack)
```

> LeetCode 321. 给定两个数组：A，B，从中选出 k 个数，组成的数字最大，要保持原数组中元素的相对顺序。

首先，从 A，B 中取出 k 个数，包含从 A 中取 i 个（0<=i<=min(len(A), k)），从 b 中选 (k-i) 个，然后将选出的两组数进行合并。选数的过程时选出尽可能大的数字序列，可以使用单调栈实现。使用单调栈维护降序序列，最后栈中前 k 个元素就是备选的集合，同时为了避免出现栈中没有元素的情况，我

们需要维护一个数字，表示还可以丢弃多少个数，它的初始值为数组长度减去要选出的数字数。代码如下，具体的写在注释里。

```
class Solution_321:
    def maxNumber(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:

        def pre(nums, k):
            drop = len(nums) - k  # number of num need to drop
            stack = []
            for x in nums:
                # 维护一个降序序列
                while stack and drop and stack[-1] < x:
                    stack.pop()
                    drop -= 1# 可以丢弃的数目减一
                stack.append(x)
            return stack[:k]

        def merge(a, b):
            # a, b 都是 list max(a, b).pop(0) 从比较大的比较结果中 pop 头一个元素
            return [max(a, b).pop(0) for _ in a + b]

        m, n = len(nums1), len(nums2)
        ans = []
        # 枚举所有的情况  (0<=i<=min(len(A), k))
        for i in range(k+1):
            if i <= m and k - i <= n:
                tmp = merge(pre(nums1, i), pre(nums2, k-i))
                ans = max(ans, tmp)
        return ans
```

上述代码中是参考别人的写出来的，尤其是 merge 函数，使用了 built-in 函数来大幅度简化的代码的复杂性，优雅。

> LeetCode 456. 找 132 pattern。132 pattern 是指存在 i<j<k 满足 A[i]<A[k]<A[j]

暴力方法可以直接 $O(n^3)$ 枚举，优化一点的暴力解是 j 的循环中维护左边的最小值作为 `A[i]` 这样复杂度可以降低到 $O(n^2)$ ，但是依然会超时。

综合使用前缀最小值数组和单调栈可以在 $O(n)$ 时间内解决这个问题。逆序扫描，维护一个升序序列，这样栈中的元素都是 `A[k]` 的候选，同时我们使用 `mins` 数组保存 `0~i` 的最小值，这样对于 `A[j] > mins[j]` 时，有可能出现 `A[k]` 满足情况，我们要在栈中找到 `A[k]` 即将栈中所有小于等于 `mins[j]` (对应 `A[i]`) 的元素都弹出, 如果栈中最后有元素（说明大于 `mins[j]` ）满足小于 `A[j]` 的话我们就找到了一组满足题意得 i,j,k。代码如下。

```
class Solution_456:
    def find132pattern(self, A: List[int]) -> bool:
        # O(n) 使用 stack 维护有序序列
```

```
    if len(A) < 3:
        returnFalse

    stack = []
    # mins 存储从 0 ~ i 区间的最小值
    mins = [0] * len(A)
    mins[0] = A[0]
    for i in range(1, len(A)):
        mins[i] = min(mins[i - 1], A[i])

    for j in range(len(A) - 1, -1, -1):  # 注意这里是反向循环的
        if A[j] > mins[j]:
            # 使用 stack 维护了一个严格的降序序列
            while stack and stack[-1] <= mins[j]:
                stack.pop()
            # 如果 stack 中还有元素的话满足 stack[-1] > mins[j]
            if stack and stack[-1] < A[j]:
                returnTrue
            stack.append(A[j])
    returnFalse
```

当然有单调栈的话对应也有单调队列（使用双端队列 Deque），单调队列与单调栈有着类似的结构，这里给出一个经典题目。

> LeetCode 239. 求滑动窗口内最大值

这个题目第一个思路就是暴力求解，枚举每一个窗口求取最大值，但这样的复杂度是 $O(kn)$。显然暴力法有很多的重复比较运算，因此我们可以优化一下暴力的过程来解决这个问题。

SlidingWindow

如上图所示，滑动串口的过程，我们每次滑动的过程实际上是丢弃最左边一个（红色），新加入右边一个（蓝色）。那么新窗口的的最大值只可能出现在两个位置，蓝色元素，绿色窗口内，同样的，前一个窗口内的最导致只可能出现两个位置，红色元素，绿色窗口内。因此我们有：

- 如果蓝色元素比前一个最大值大，那么蓝色元素是新串口的最大值
- 如果蓝色元素小于等于前一个最大，但是前一个最大值出现在绿色窗口内，那么新的窗口最大值与前一个相同。
- 其他情况我们无法判断，所以直接便利窗口计算最大值。

优化过后的代码如下。

```
class Solution_239:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # 对暴力解进行优化
        if len(nums) == 0:
            return []
```

```
        res = [max(nums[:k])]
        i = 1
        while i + k <= len(nums):
            prev = res[-1]
            if prev < nums[i + k - 1]:
                res.append(nums[i + k - 1])
            elif prev > nums[i - 1]:
                res.append(prev)
            else:
                res.append(max(nums[i:i + k]))
            i += 1
        return res
```

重点是要讲怎么用单调队列来解决这个问题，其实上面优化的解法中已经涉及到了队列的思想。我们来看看怎么用单调队列来做。首先，队列要维护的保持窗口的大小，也即是队列中的元素都是当前窗口内的（并不一定是当前窗口内所有元素）然后我们可以用单调队列维护一个降序序列，这样每个窗口内的最大值就是队首元素。

维护降序序列和单调栈的结构基本一致，利用双端队列的一端模拟栈来实现；维护窗口的部分使用双端队列另一端，将不是窗口内的元素出队。代码如下。

```
class Solution_239:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if len(nums) == 0:
            return []
        res = []
        que = collections.deque()
        for i, x in enumerate(nums):
            # 这里的结构和单调栈基本一样，用队尾模拟一个栈维护降序序列
            while que and nums[que[-1]] < x:
                que.pop()
            que.append(i)
            # 从队首出队维护窗口
            if que[0] == i - k:
                que.popleft()
            # 从第一个窗口开始保存结果
            if i >= k-1:
                res.append(nums[que[0]])
        return res
```

总结，单调栈（Monotone Stack），就是栈内元素都是单调递增或者单调递减的，有时候需要严格的单调递增或递减，需要根据具体的题目情况来定。每个元素只会入栈一次，所以单调栈维护的复杂度是 $O(n)$, 可以用来求元素向左遍历第一个比它小的元素（前一小元素），元素向右遍历第一个比它小的元素（后一小元素），元素向左遍历第一个比它大的元素（前一大元素），元素向右遍历第一个比它大的元素（后一大元素）问题。使用单调栈可以大幅度降低求解某些问题的复杂度。单调队列是与单调栈差不多的结构，使用队列来维护队列的元素的有序性。

本来想着这个月把 DP 题过一遍，然后总结写完的，然而 DP 太难了，根本肝不动。而且最近事情又多，刷题的时间不够用了。

多做题，多总结~（吐槽一下这个工具的代码高亮真的好难受）

文章已于2019-12-21修改