单调栈与单调队列在蓝桥杯中出现较少，大家只需要掌握基本的模板即可。

单调栈

1.基本思想

单调栈求解的基本问题

> 在一个线性数据结构中，为任意一个元素找左边和右边第一个比自己大/小的位置，要求 $O(n)$ 的复杂度

对于这类问题，基本解法很容易想到 $O(n^2)$ 的解法，不过这种解法明显太基础了，有优化的空间。单调栈就可以优化基本解法的时间复杂度。

时间复杂度

单调栈是$O(n)$的。

> 时间复杂度的证明：所有的元素只会进栈一次，而且一旦出栈后就不会再进来了。因此总的进出次数最多是 $2n$ 次，因此时间复杂度为 $O(n)$。

单调栈的性质

单调栈里的元素具有单调性:

1. 单调递减栈=>向栈生长的地方单调递减;
2. 单调递增栈=>向栈生长的地方单调递增

元素加入栈前，会在栈顶端把破坏栈单调性的元素都删除!

使用单调栈可以找到元素向左遍历第一个比他小的元素，也可以找到元素向左遍历第一个比他大的元素。

一般解题思路

1. 栈中不存放值，而是存放下标;
2. 在尝试去将一个元素加入栈前，先对它和栈顶元素进行比较，并按照单调性要求，把破坏栈单调性的栈顶元素都删除，直至满足单调性。

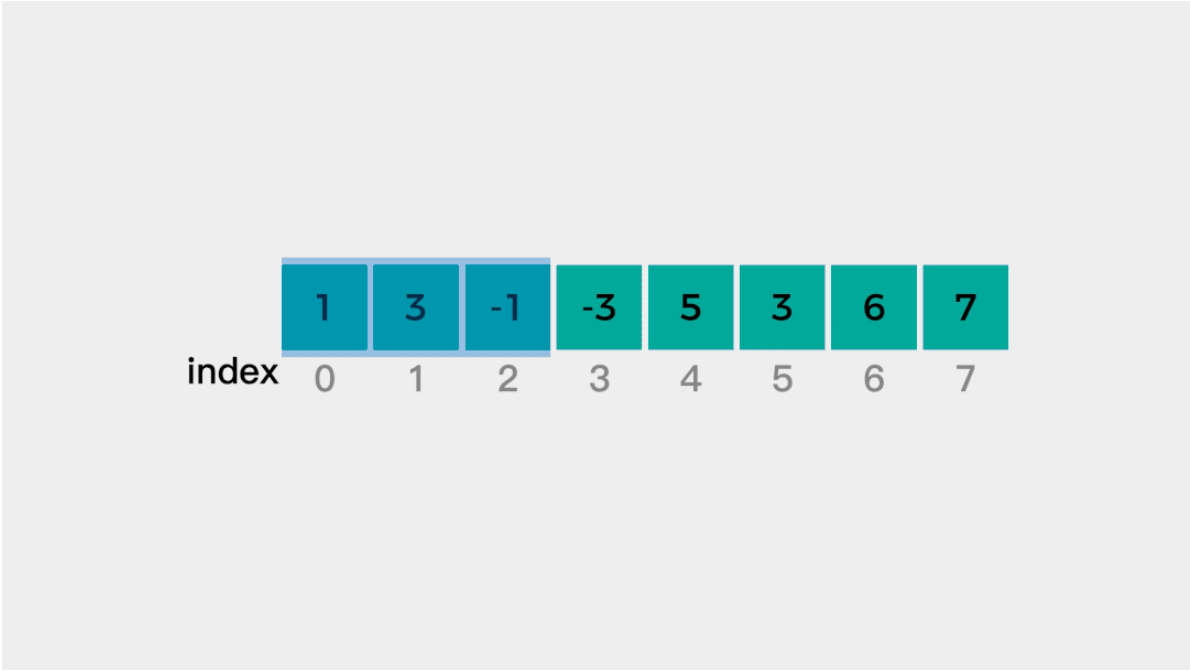
单调队列

问题引入

给定一个长度为 $n(n \leq 10^7)$ 的数组 $A[]$和长度为 $k(k \leq 10^5)$ 的窗口，这个窗口从数组左端向右端滑动，求每向前滑动一次后，窗口内元素的最小值，并输出它们。例如：

n = 8
k = 3

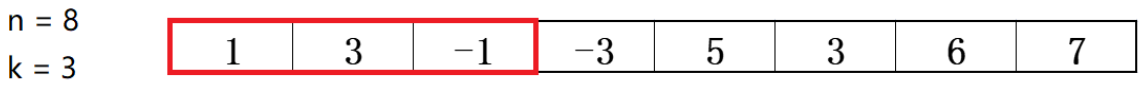
1	3	−1	−3	5	3	6	7



从暴力枚举到单调队列

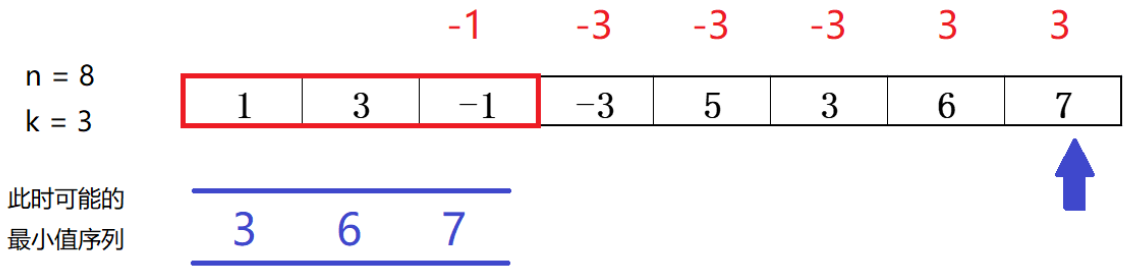
时间复杂度$O(nk)$，太高，如何优化?

考虑尽可能排除掉不可能是答案的值，第一层循环不能降低了，只能考虑降低第二层循环的次数。比如下图中，当处理到 -1 的时候，很明显，前面的 1 和 3 不会成为当前及之后的窗口中的最小值（因为 -1 比它们更靠后），可以直接排除掉了。



更一般地，如果 $A[i]$ 和 $A[j]$ 处在同一个窗口内，并且 $i < j$ 且 $A[i] \geq A[j]$，那么，必有 $A[i]$ 不可能是当前窗口内的最小值，也不可能是后续窗口内的最小值。即 $A[i]$ 是一个无用的数。

总结一下，我们发现，当处理 $A[j]$ 时，$A[j]$ 之前的所有比 $A[j]$ 大的元素都可以排除，不用考虑。由此，我们可以得出上面例子中的各个滑动窗口内的最小值如下：



最后，当滑动到 7 时，此时可能的解有三个：3,6,7。元素 -1 在处理到元素 3 时被排除掉，元素 -3 在处理元素 6 时被排除掉。

对于上面这种既有序，又单调的序列，我们称之为“单调队列”。

性质

单调队列中元素之间的关系具有单调性，分为 **单调递增队列** 和 **单调递减队列**。

1. 队列中的元素在原来的列表中的位置是由前往后的(随着循环顺序入队)。
2. 队列中的队首元素和队尾元素的距离应不超过问题中规定的区间长度（**区间性**）
3. 队列中元素的大小是单调递增或递减的（**单调性**）。

因此，

1. 为了保证队列的区间性，那么我们在将元素入队之前，应该将队列中会破坏区间性的（队首）元素删除掉。
2. 为了保证队列的单调性，那么我们在将元素入队之前，应该将队列中会破坏单调性的（队尾）元素删除掉。

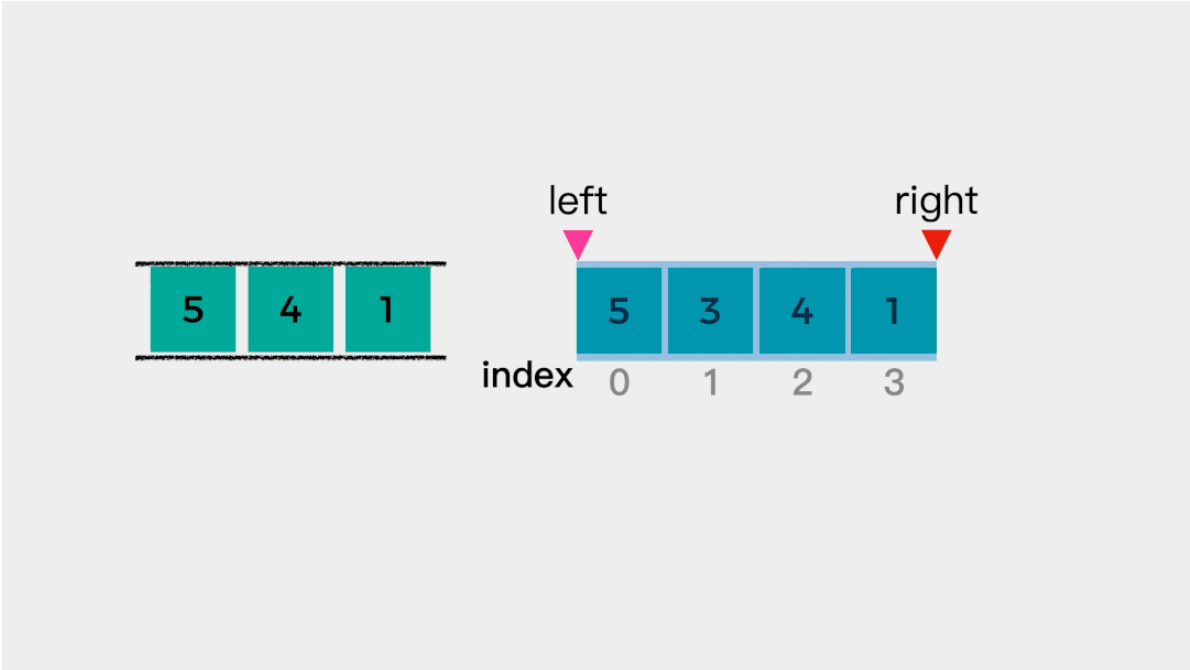
操作

单调队列和单调栈在实现上有一点是相同的： **当需要将一个元素加入队列或栈时，就将破坏队列或栈的单调性的元素删除。**区别是单调栈只在栈顶进出元素，而单调队列为了维持单调性，需要 **同时在队首和队尾弹出元素。**因此，单调队列实质上是一类 **双端队列（deque）。**

单调队列在加入一个元素时，我们通过一定操作维护队列的单调性。该队列特征的操作便是“后移一位”。顾名思义，后移一位就是指队列维护的区间往后移一位。这个操作中要做的事情是弹出队首与加入队尾。

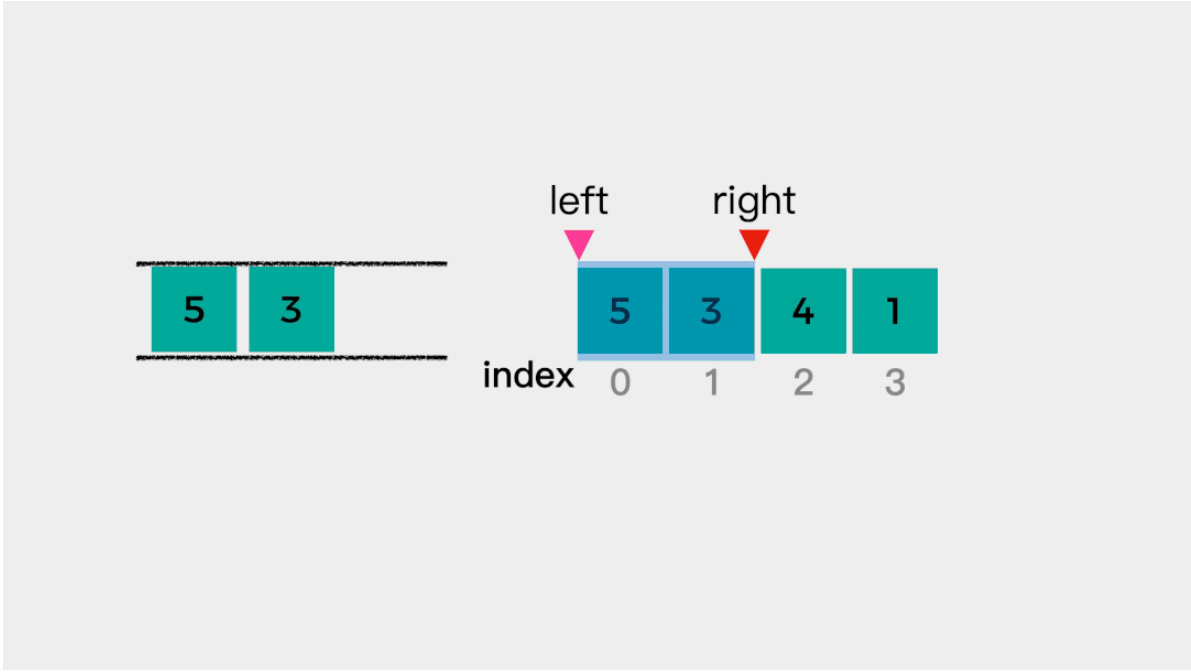
1. 弹出队首

对于插入的元素，我们检查它与队首元素的位置之差，当这个 **差大于队列定长时说明队首元素已在区间之外**，应当弹出。



2. 加入队尾

为了保证这个队列中元素的单调性，对于加入的元素，我们可能要删除一些队尾的元素。以单调递增队队列为例，插入时要 **弹出所有比要插入元素大的队尾元素，直到队尾元素不再比其大再插入。**此时插入元素时要记下插入元素在原序列中的位置，这个信息将会对弹出队首时发挥作用。



代码实现

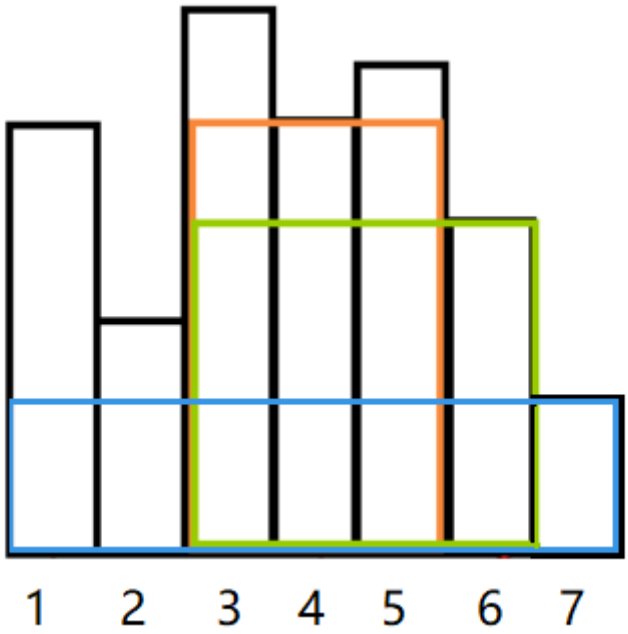
1. 手写队列

```
q = [0] * N
head, tail = 0, 0
for i in range(n):
    while head < tail and check_out(q[head], i):
        head += 1  # 队头滑出窗口区域就弹出队首
    while head < tail and check(q[tail - 1], i):
        tail -= 1  # 队尾不满足单调性就弹出队尾
    q[tail] = i  # 当前元素入队
    tail += 1
```

蓝桥杯真题

直方图中最大的矩形

依旧记录每个矩形可向两侧扩展的左右边界。但是在计算每个矩形可以扩展的左/右边界时，可以发现我们真正关心的是第一个比当前矮的矩形，因为它决定了可以左/右扩展的长度。而这类“求左边和右边第一个比自己大/小的位置”的问题，单调栈可以解决。并且时间复杂度是 $O(n)$ 的。



如上图所示，每个矩形高度从左向右依次是：`5,3,7,5,6,4,2`（代码中左右补两个高度 -1的矩形）在求每个矩形向左扩展时的最大长度时，我们维护一个单调栈即可。因为要找的是左侧第一个比当前高度矮的矩形，因此是一个单调递增栈。

矩形编号	当前矩形高度	当前的单调栈
0	-1	[]
1	5	[0]，向左长度为 1-0=1，向左最大面积为 5
2	3	[0]，向左长度为 2-0=2，向左最大面积为 6
3	7	[0,2]，向左长度为 3-2=1，向左最大面积为 7
4	5	[0,2]，向左长度为 4-2=2，向左最大面积为 10
5	6	[0,2,4]，向左长度为 5-4=1，向左最大面积为 6
6	4	[0,2]，向左长度为 6-2=4，向左最大面积为 16
7	2	[0] ，向左长度为 7-0=7，向左最大面积为 14

定义单调递增栈去统计每个矩形右侧第一个小于当前矩形的矩形的位置（求出向右扩展长度）。然后再定义一个数组保存向左能扩展到的大于当前高度的长度（当不满足单调栈特性时，求和就可以求出向左扩展的最大长度）。如此一来，对于栈顶元素的向左右扩展的最大面积就能求出来了。

```
def solve(ls: list) -> int:
    st, area = [], 0
    for i in range(len(ls)):
        while st and ls[i] < ls[st[-1]]:
            top = st.pop()
            area = max(area, ls[top] * (i - st[-1] - 1))
        st.append(i)
    return area


if __name__ == '__main__':
    res = []
```

```
    while True:
        ls = list(map(int, input().split()))
        if not ls[0]:
            break
        else:
            res.append(solve([0] + ls[1:] + [0]))
    for r in res:
        print(r)
```

习题

小蓝与赛马娘

这道题思路简单几乎是一个单调队列的板子题

“当一头马娘左边 D 距离内和右边 D 距离内都有身高至少是它的两倍的马娘，就会觉得不自在影响她的比赛发挥。”

这句话提示它是一个滑动窗口，窗口长度是 D，身高至少是两倍，那么只要窗口内的最大值大于等于当前高度两倍就可以判断为从某一侧拥挤了。

于是，只需要从左到右和从右到左都分别跑一遍单调队列滑动窗口，去判断每侧是否会拥挤，如果都拥挤，那么就答案加一。

```
from collections import deque

N = 50005

n, d = map(int, input().split())
a = [tuple(map(int, input().split())) for _ in range(n)]

a.sort()
x = [0] * (n + 1)
h = [0] * (n + 1)
for i in range(1, n + 1):
    x[i], h[i] = a[i - 1]

cl = [False] * (n + 1)
cr = [False] * (n + 1)

q = deque()
for i in range(1, n + 1):
    while q and h[q[-1]] < h[i]:
        q.pop()
    while q and x[i] - x[q[0]] > d:
        q.popleft()
    if q and h[q[0]] >= 2 * h[i]:
        cl[i] = True
    q.append(i)

q.clear()
for i in range(n, 0, -1):
    while q and h[q[-1]] < h[i]:
        q.pop()
    while q and x[q[0]] - x[i] > d:
        q.popleft()
    if q and h[q[0]] >= 2 * h[i]:
```

```
            cr[i] = True
        q.append(i)

ans = sum(cl[i] and cr[i] for i in range(1, n + 1))
print(ans)
```

小蓝与排队

仔细理解题意，题目可以转化为找当前数字向右查找的第一个大于他的数字之间有多少个数字，然后将每个结果累计就是答案。重点在于找到每个数字右边第一个大于他的数字的下标，如果知道对应的下标，就可以计算出之间的数字个数（距离）。具体可以用单调栈来处理。

从前往后遍历的单调递减栈

> 核心思想：从前往后计算，统计每个人可以挡住几个人的视野，以及对应的被挡住的人能看到的数量。

题中说，个子高的可以看到个子矮的，所以我们可以维护一个单调递减栈。然后在维护过程中对于当前的人如果发现单调栈的栈顶的人身高比当前的矮，就意味着栈顶的人被当前的人挡住了，因此栈顶的人能看到的人数也就确定了，如此处理即可。需要注意的是，可以在最后一个人后面插入一个极高的人，以方便处理没人能挡住视野的情况。具体见代码：

```
N = 100010
h = [0] * N
stk = [0] * N
top = 0
ans = 0

n = int(input())
input_line = list(map(int, input().split()))
for i in range(1, n + 1):
    h[i] = input_line[i - 1]
h[n + 1] = float('inf')

for i in range(1, n + 2):
    while top and h[stk[top]] <= h[i]:
        ans += i - stk[top] - 1  # stk[top] 能看到的人数
        top -= 1  # stk[top] 出栈，不再参与计算
    top += 1
    stk[top] = i

print(ans)
```

```
n = int(input())
h = [0] * 100005
ans = 0
s = []

input_line = input().split()
for i in range(1, n + 2):
    if i < n + 1:
        h[i] = int(input_line[i - 1])
    else:
        h[i] = float('inf')

    while s and h[s[-1]] <= h[i]:
```

```
        # s.top() 能看到 i 前面一个人
        ans += i - s[-1] - 1
        s.pop()  # s 统计完了，出去

    s.append(i)

print(ans)
```