Time complexity analysis for the average case (two pointer)

By MirShao07

For the Sliding Window Maximum problem, the Deque solution has a straightforward O(N) time complexity and O(K) space complexity. The two pointer solution, however, requires some careful analysis. The space complexity is O(1), as we only use two pointers and nothing more, but the time complexity is not so obvious. Best case scenario we have O(N), worst case scenario we have O(N^2), but what about the average case? Let's find out.

Step 1

Illustration:

```
    X         . . . . . . . . . . Y . . .
    |                             |
cur_max                  Y>=X, new_max
```

figure 1:
window_start_index is i

- Case 1: Within the next K elements, there exists an element Y>=X, then Y will be discovered before i reaches X, so findMax subroutine will not be called.

- Case 2: Within the next K elements, all elements are strictly smaller than X.

For every window, these two cases partition the entire sample space, so we only need to consider the probability of the two cases.

- prob(case2) = 0.5^K, as each number is equally likely to be bigger or smaller, so if all K elements are smaller than X, probability is 0.5^K. (For duplicates, we include them in the bigger group, as we want to discover the rightmost max);

- prob(case1) = 1-p(case2) = 1-0.5^K;

Comment: So we already see that as K gets larger, case 2 becomes very unlikely, whereas case 1 is very likely.

Now we need to compute the Expectation of the running time T(n).
E(T(n)) = prob(case1)*T(case1) + prob(case2)*T(case2).
T(case1) = ?   T(case2) = ?

Step 2

For Case 1, every update is O(1), as new max is always discovered before old max expires, so the findMax subroutine is not called. T(case1) = CN.

For Case 2, findMax will be called, when old max expires, and new max can only be found by traversing the current window. The subroutine itself runs in O(K), but how often do we call it? The key here is the POSITION of the SECOND MAX (or duplicate of cur_max) element within K spaces of old max.

Illustration:

```
    6         . . . . . . . . . . . . 4 . .
    |                                 |
cur_max                  second_max
```

figure 2: second_max is far from cur_max

```
    6         4 . . . . . . . . . . . . . .
    |         |
cur_max  second_max
```

figure 3: second_max is close to cur_max

The intuition here is that the further away second_max is from cur_max, the less often we call subroutine findMax.

e.g. As soon as 6 expires, we call findMax and discover 4, all the updates before i reaches 4 are O(1). Only when 4 expires do we call findMax.

As second_max is equally likely to appear in any position within the window, one extreme case is data in reverse order, in which case we have to call findMax EVERY TIME to do the update. For an input of N, we have to call findMax N times. (Assuming it's ALWAYS the worst case: entire array is in reverse order.)

Another extreme case is when second_max is at the Kth position after cur_max: the optimal case. In this case we call findMax N/K times. (Assuming it's ALWAYS the optimal case.)

We can visualise the input as a pulse shape array, separated by spikes, with each spike being the second_max, the advantage we gain by recording the index position is the length of the gap

between spikes. Obviously, the further away spikes are from each other, the more advantage we gain. The number of spikes is exactly the number of times we call findMax: that's how we discover each spike.

6……4……4……3…
figure 4: pulse shape

So the expected value of the frequency of calling findMax:
(We omit the ceiling function of each term in the summation as ceiling ultimately only adds a constant.)
E(# call findMax)
≈ 1/K * Sum(N + N/2 + N/3 + N/4 +…+ N/K)
= N/K * (1 + 1/2 + 1/3 + 1/4 +…+ 1/K)
≈ N/K * ($\ln K$ + Euler's constant)

Note: (1+1/2+1/3+…+1/K) is the Kth harmonic number.

Each time we call findMax, we have exactly K operations, therefore,
Average T(n) for Case 2 is:
E(T(case2)) = K * N/K * ($\ln K$ + Euler's constant) = N * ($\ln K$ + Euler's constant)

Step 3 Conclusion: O(N)

Combining step 1 and step 2, for the average case, we have:

E(T(n))
= prob(case1)*T(case1) + prob(case2)*T(case2)
= (1-0.5^K)*(CN) + (0.5^K) * N * ($\ln K$ + Euler's constant)

Do the big O analysis, this SEEMS to be O(N*logK), but if we look closer, we realise this interesting fact:

For the second term: (0.5^K) * N * ($\ln K$ + Euler's constant),
As K gets larger, 0.5^K gets smaller, logK gets larger,
0.5^K is an exponential function, whereas logK is a sublinear function,
By L'Hôspital's rule, asymptotically, exponential dominates sublinear.
Therefore (0.5^K) * logK is approaching zero, so the entire second term (which we took great pains to derive) is negligible in the big O sense.

For the average case of the two pointer method:

- **Time complexity is O(N),**
- **Space complexity is O(1).**

This is arguably better than the method with Deque or the more naive TreeMap, as it has better space complexity and incurs less overheads.

Step 4 Extension

We can also see this method (as well as the deque and treemap solution) as a viable online algorithm if we have an infinite input such as a stream of data to process, instead of a static array. As mentioned above, as the space complexity is O(1), the two pointer method scales much better than the deque or treemap method.