Sliding Window Maximum

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

This document is still a draft. This means that there may be typos, errors, or imprecisions. Please, read it critically and report me any correction or suggestion that may help to improve it.

1 Problem

The *Sliding window maximum* problem is as follows.

▶ **Problem 1** (Sliding window maximum). *Given an array $A[0, n-1]$ and an integer k, the goal is to find the maximum of each and every subarray (window) of A of size k.*

The most straightforward solution for this problem is reported in Figure 1. The idea is to process each of the $n - k + 1$ windows independently. For each window, we identify its largest element by scanning all its elements in time $\Theta(k)$. Thus, this brute force solution runs in $\Theta(nk)$ time.

The reason of the inefficiency of this solution is the fact that, when computing the maximum of a window, we completely ignore computations done for identifying maxima of the previous windows.

The design of a more efficient solution starts from two simple observations. Firstly, elements of a window can be represented as a multiset $\mathcal{M}$, so that, the answer for the window is the largest element in the multiset. Secondly, two consecutive windows differ in just two elements: the first element of the first window, which leaves the scene, and the last element of the second window, which enters. Thus, we can obtain the multiset of the new window from the multiset of the previous window by inserting and removing two elements. Thus, we need a data structure to represent multisets that supports three operations: insertion of a new element, deletion of an element, and reporting of the maximum element of the multiset. This way, we can let the window slide on the array and we can update and query the multiset to compute the answers. Now the question is: what's the best data structure supporting these operations? Balanced Binary Search Trees (BBST) supports any of these operations in $\Theta(\log |\mathcal{M}|)$, where $|\mathcal{M}|$ is the number of elements in the multiset (and it is optimal in the comparison model). This way, we can solve the problem in $\Theta(n \log k)$ time.

2 Linear time solution

Is it possible to do better than the previous solution? Well, as this section is titled *Linear time solution* you may suspect *"Yes, it is"*. But, why is it intuitively reasonable to think about an improvement? Well, a good point is to observe that the previous solution is able to do much more than what is needed to. If I ask you: what's the second largest element in

```cpp
template<typename T>
std::vector<T> SWM_brute_force(std::vector<T> const& A, size_t
    k) {
  std::vector<T> maxs;
  maxs.reserve(A.size()-k+1);

  for (int i = 0; i < A.size()-k+1; i++) {
     T max_value = A[i];
     for (int j = i; j < i+k; j++)
        max_value = max(max_value, A[j]);
     maxs.push_back(max_value);
  }
  return maxs;
}
```

Figure 1 Implementation of the brute force solution for Sliding window maximum problem

the window? No problem, the second largest element is the predecessor of the maximum and BBST supports also this operation in $\Theta(\log n)$ time. Actually, you would be able to report the top-x largest or smallest elements in $\Theta(x + \log n)$ time (How?). This is because the BBST is implicitly keeping all the elements of all the windows sorted. The fact that we can do much more that what is requested, it's an important signal to think that a faster solution could actually exist. Still, the title of this section is a stronger one.

Surprisingly, the better solution uses an elementary data structure: a queue. We actually require a Double-Ended Queue (*Deque*), which supports constant time insertion, removal and access at the front and the back of the queue. There are several ways to implement a Deque. The easiest way is probably with a bidirectional list.

The algorithm starts with an empty deque Q and with the window W that covers the positions in the range $\langle -k, -1\rangle$. That is, the window starts before the beginning of the array A. Then, we start sliding the window one position at the time and remove/insert elements from Q. We claim that the front of Q will be the element to report.

More preciselly, we repeat n times the following steps.

- Move the window one position to the right
- Remove from the head of Q the elements which are no longer in the window
- Insert the new element from the tail of Q and remove all the elements above it until we find a larger element
- Report the head of Q as the maximum in the current window

An implementation of the above pseudo-code is reported in Figure 2.

Figure 3 shows a running example of the solution. We conclude that `https://www.youtube.com/watch?v=4An1BrG2u_4`. Why?

Time complexity analysis and correctness

Before even thinking about the correctness of the solution, let us analyze its time complexity. Let us first use the standard approach. We have a loop that is repeated n times. What's the cost of an iteration? Looking at the implementation it should be clear that its cost is dominated by the cost (and, thus, number) of pop operations. However, in a iteration we

```cpp
#include <vector>
#include <deque>

template<typename T>
std::vector<T> SWM(std::vector<T> const& A, int k) {
        std::deque<int>  Q; // Q will store positions not
           values
  std::vector<T> maxs;
  maxs.reserve(A.size()-k+1);

  for (int i = 0; i < A.size(); ++i) {
    // Removes from front elements which are no longer in the
       window
    while ( (!Q.empty()) && Q.front() <= i-k)
        Q.pop_front();
    // Removes from back elements which are no longer useful,
       i.e., no greater than the current element
    while ( (!Q.empty()) && A[i] >= A[Q.back()])
        Q.pop_back();
    // Insert the new element
    Q.push_back(i);

    // Front is the maximum of the current window
    if(i >= k-1)
      maxs.push_back(A[Q.front()]);
  }
  return maxs;
}
```

Figure 2 Implementation of the linear time solution for Sliding window maximum problem

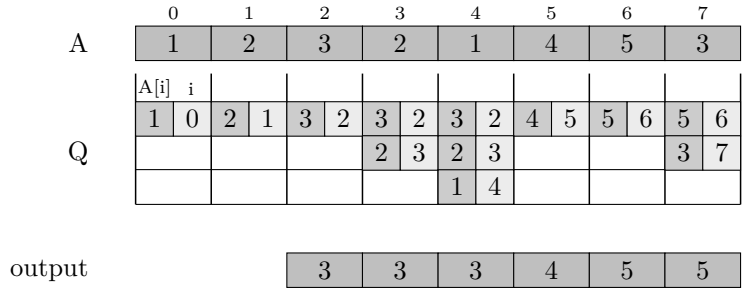

Figure 3 A running example of the linear time solution on the array $A = \{1, 2, 3, 1, 4, 5, 2, 3\}$ with $k = 3$. The deque Q stores a pair of values: a position i in A and the value at that position, i.e., $A[i]$.

may pop out all the elements in the deque. As far as we know there may be up to n elements in the deque and, thus, an iteration costs $O(n)$ time. So, the best we can conclude is that the algorithm runs in $O(n^2)$ time. Can't go too much far with this kind of analysis!

In fact, it is true that there may exist very costly iterations, but their are greatly amortized by several very cheap ones. Indeed, the overall number of pop operations cannot be larger than n as any element is not considered anymore by the algorithm as soon as it is removed from Q. Each of them costs constant time and, thus, the algorithm runs in linear time.

Let us prove the correctness of the algorithm. We first observe that elements in Q are sorted in decreasing order. This can be proved by induction on the number of iterations. The claim is true for the initial Q as it is empty. Given the queue after i iterations, by hypothesis it is sorted. The current iteration will only remove elements (no change in the ordering of the remaining elements) or insert the current element $A[i+1]$ as the tail of the queue just below the first element which is larger than it (if any). Thus, the queue remains sorted.

Sortedness of Q is a nice starting point, but we need now to introduce the definition of rightLeaders of the window. Given a window, an element is called rightLeader if and only if the element is larger than any other element of the window at its right.

As an example, consider the window of size 5 below.

The rightLeaders of this window are drawn in red.

We are now ready to prove a nice property of the elements in Q: *At every iteration, Q contains all and only the* rightLeader*s of the current window.*

This is quite easy to see. Firstly, any rightLeader cannot be removed from Q as all the subsequent elements are smaller than it. Secondly, any non rightLeader will be removed as soon as the next rightLeader will enter Q. Finally, any element outside the window cannot be in Q. By contradiction, let us assume that Q contains one of such element, say a. Let r be the largest rightLeader. On the one hand, a cannot be smaller than or equal to r, otherwise a would be removed when inserting r in Q. On the other hand, a cannot be larger than r, otherwise it would be in front of Q and removed by the loop at line 12.

The correctness of the algorithm is derived by combining the sortedness of Q with the fact that the largest rightLeader is the element to report.

3 Next larger element

Try to modify the previous solution to solve the *Next larger element* problem, which is as follows.

▶ **Problem 2** (Next larger element). *Given an array $A[0, n-1]$ having distinct elements, the goal is to find the next greater element for each element of the array in order of their appearance in the array.*