# An Optimal Algorithm for Sliding Window Order Statistics

**Pavel Raykov** ✉
Google Zürich, Switzerland

## Abstract

Assume there is a data stream of elements and a window of size $m$. Sliding window algorithms compute various statistic functions over the last $m$ elements of the data stream seen so far. The time complexity of a sliding window algorithm is measured as the time required to output an updated statistic function value every time a new element is read. For example, it is well known that computing the sliding window maximum/minimum has time complexity $O(1)$ while computing the sliding window median has time complexity $O(\log m)$. In this paper we close the gap between these two cases by (1) presenting an algorithm for computing the sliding window $k$-th smallest element in $O(\log k)$ time and (2) prove that this time complexity is optimal.

**2012 ACM Subject Classification** Theory of computation → Sorting and searching

**Keywords and phrases** sliding window, order statistics, median, selection algorithms

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2023.5

**Acknowledgements** We thank anonymous reviewers for their useful comments which led to an improved manuscript. We would also like to thank Mikhail Churakov, Christian Matt, and Dimitris Paparas for proofreading the paper.

## 1 Introduction

Selection and sliding window algorithms are considered to be among the classical computer science algorithms with numerous applications [7]. In this paper we consider the overlap between these two areas: what is the optimal sliding window algorithm for selecting the $k$-th smallest element? The sliding window average, median, minimum, and maximum algorithms have been well studied and have become a part of folklore. It is well known that for a data stream of elements and a window of size $m$, one can compute the sliding window average, minimum, and maximum in $O(1)$ time each time the sliding window moves [9, 25]. The situation is different for the median – the best possible algorithm can only compute the sliding window median in $O(\log m)$ time [11, 14, 24]. Motivated by the gap between the median and other order statistics algorithms, we study the sliding window algorithms for selecting the $k$-th smallest element for arbitrary $k$. In this paper we present the first algorithm computing the sliding window $k$-th smallest element in time $O(\log k)$ while using only $O(m)$ of memory storage. We also present a lower bound showing that this algorithm has the optimal running time complexity.

### 1.1 Prior Work

Algorithms for computing various statistics over a data stream play an important role in computer science and database processing in particular [4]. One can roughly divide these algorithms into two groups: exact and approximate ones. The research in the area of the approximate algorithms focuses on the problem where the whole data stream (or the sliding window) cannot fit into the memory and hence one seeks for a tradeoff between how much of the data need to be stored additionally while reading the input stream a limited number of times vs how accurate the output computed statistics can be [2, 3, 5, 8, 12, 18].


The exact algorithms can be further subdivided into those who operate on the whole input set and on the sliding window only. Once it was shown that the selection algorithms can be linear [6], the research in the area of the exact algorithms that work with the whole input set focused on algorithms minimizing the overall number of basic operations [22] and the algorithms performing well in practice [1]. There is also a corpus of the exact algorithms that do not only compute a statistical function over a fixed data set but also support range queries over it [17, 27].

The exact algorithms working with the sliding window include algorithms for standard aggregation functions like maximum, minimum, average, sum, count [9, 19, 26] and their monoid-compatible extensions [25]. The exact algorithms for computing quantiles over the sliding window are limited to the median filtering algorithms [14, 24]. These algorithms produce their output over the sliding window by maintaining a data structure holding the elements of the window and updating it accordingly whenever the window moves. Depending on the aggregation function all the known exact sliding window algorithms can update the sliding window in constant, logarithmic, or linear time in terms of the sliding window size.

This paper considers the latter model where the aggregation function outputs the $k$-th smallest element. For a window of size $m$ and the order statistic $k$, we present an algorithm that requires $O(m)$ memory to store the elements of the sliding window and can make updates to it in $O(\log k)$ time whenever the sliding window moves.

## 2 Notation and Tools

We define a sliding window algorithm as an algorithm that exposes a single interface update-window reading a new data stream element $v$ and outputting a statistic function over the last $m$ elements read. If fewer than $m$ elements have been read so far, the output is not defined. *The time complexity* of the algorithm is defined as the time complexity of a single invocation of update-window. *The space complexity* of the algorithm is defined as the amount of storage the algorithm utilises.

In the context of this paper we consider the sliding window algorithms that compute the $k$-th smallest element of the sliding window:

▶ **Definition 1.** *We say that a sliding window algorithm parameterised with integers $k$ and $m$ computes* the sliding window $k$-smallest element *if* update-window *returns the $k$-th smallest element among the last $m$ elements read by the algorithm.*

*The smallest element is indexed starting with* 1*, i.e., the* 1*-st smallest element corresponds to the window minimum, while the $m$-th smallest element corresponds to the window maximum.*

We will also assume that $k \leq m/2$ everywhere. To derive the same results for $k > m/2$, one needs to update all the algorithms to use a reverse order on the elements and output the $(m - k + 1)$-th smallest element.

Without loss of generality, we assume that the input data stream has unique elements only.[1] The easiest approach to achieve this is to enumerate each new data stream value $v$ with an increasing index $i$ and then operate on the tuples $(i, v)$ instead of the values only. Then, we can compare the tuples in the natural way by first looking at their values, and if they are the same, break the ties using the index entry. In the end, when producing the output one needs to drop the index entry of the tuples and output the value only.

[1] We use this assumption to simplify the machinery around the binary search trees which do not have a canonical multiset support. We believe that this requirement can be dropped at an expense of a more sophisticated analysis of handling the duplicate elements.

We employ a standard notion of arrays. Given an array $a$, we refer to an individual element in the array at index $i$ by writing $a[i]$; we start array indexing with 0 by default. We write $a[i, j]$ to denote the range of array elements $a[i], a[i + 1], \ldots, a[j]$. If $j < i$, then the expression $a[i, j]$ returns the empty set. Let k-smallest-set$(a[i, j])$ denote the set of the $k$ smallest elements in $a[i, j]$.

We assume there exists an implementation of AWBBS (augmented weight-balanced binary search) trees [20](e.g., based on red-black trees where each node is additionally *augmented* with the size of its subtree) that supports execution of the following operations on a tree with $n$ elements in $O(\log n)$ time:

- add: adds an element to the tree;
- remove: removes an element from the tree;
- max: outputs the maximum element of the tree;
- size: outputs the size of the tree;
- find-rank-one-tree: outputs an element of the tree with the given rank.

We also assume that the tree can be implemented using $O(n)$ space to store $n$ elements.

Given several separate trees we define the $k$-th smallest element between them to be the $k$-th smallest element among all the elements of the trees. Given three AWBBS trees of size at most $n$ we assume that there is a function computing the $k$-th smallest element between them in time $O(\log n)$. An example implementation of such a function find-rank-three-trees is given in Section A.

# 3 The Algorithm

## 3.1 Overview

The first trivial approach for the $k$-th smallest element sliding window algorithm is to maintain an AWBBS tree with $m$ elements of the sliding window. Then, at each invocation of update-window we just execute add for the newly added element, call remove on the element that falls outside of the sliding window, and then search for the element with rank $k$ in the tree by invoking find-rank-one-tree. The issue with this algorithm is that each operation takes time proportional to the size of the sliding window $O(\log m)$ and we would like to design a faster algorithm.

The first refinement of the trivial approach is to limit the size of the AWBBS tree by storing only the $k$ smallest elements of the sliding window in it. This will bring down the complexity of the elementary operations from $O(\log m)$ to $O(\log k)$ but it is not clear how to implement this approach. In particular, while adding a new element to such a tree is straightforward (we add a new element and then remove the maximum if the tree size exceeds $k$), removal of the elements that fall outside of the sliding window is not clear. If the removed element is one of the $k$ smallest elements we need to find the smallest element outside of the maintained tree that now needs to be added to it. This seems to require sorting the elements of the sliding window which again incurs $O(\log m)$ costs.

The second refinement is based on the following idea: while it does not seem to be possible to maintain a single AWBBS tree with the $k$ smallest elements of the sliding window, we can split the sliding window into separate blocks and maintain a separate AWBBS tree with the $k$ smallest elements of *each block* and not the whole sliding window. Then, finding the $k$-th smallest element of the sliding window now requires searching for an element with rank $k$ in multiple AWBBS trees which still can be done in $O(\log k)$ time (see Section A for an example of such an algorithm). Note that the "maintenance burden" of keeping the AWBBS tree with the $k$ smallest window elements is now distributed among multiple blocks. In particular,

one of the blocks will receive a new data stream element, while another one has the element falling outside of the sliding window and has to deal with its deletion. The trick is that while the window slides we will have enough time to prepare each block for deletions so that they also will incur only $O(\log k)$ costs once the deletion happens.

## 3.2 Description

We assume that the input data stream is split into consecutive blocks of size $m/2$.[2] At any moment of time the sliding window $W$ spans over three consecutive blocks $B_{left}, B_{middle}$ and $B_{right}$ as shown in the picture:

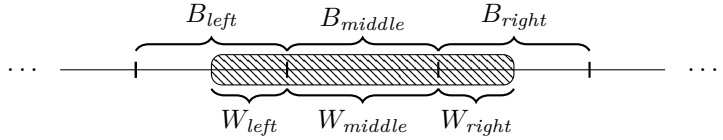


**Figure 1** Sliding window ▨ spanning over $B_{left}, B_{middle}, B_{right}$.

The sliding window then consists of the following three parts: $W_{left}$ intersecting with $B_{left}$, $W_{middle}$ equals to $B_{middle}$, and $W_{right}$ intersecting with $B_{right}$.[3] The proposed algorithm maintains AWBBS trees $T_{left}$, $T_{middle}$, and $T_{right}$ with the $k$ smallest elements of the $W_{left}, W_{middle}$, and $W_{right}$ parts, respectively. Computing the $k$-th smallest element of the sliding window is done by calling function find-rank-three-trees (see Section A for its description) over $T_{left}$, $T_{middle}$, and $T_{right}$.

The maintenance of the trees is done separately for each of the trees. $T_{right}$ is maintained in a straightforward way as we have already noticed before: we add a new element to it and then remove the maximum if the size of $T_{right}$ exceeds $k$. Nothing is done for maintaing $T_{middle}$: it is just assigned to $T_{right}$ every $m/2$ steps. The core of the algorithm lies in preparing a data for maintaining $T_{left}$. This data is computed based on $B_{middle}$ and then used for maintaining $T_{left}$ once the window slides to a point of time when the elements of $B_{middle}$ become $B_{left}$. We prepare a data structure which allows us to move from a tree containing k-smallest-set$(B_{middle}[0, m/2-1])$ to a tree containing k-smallest-set$(B_{middle}[1, m/2-1])$ (then to a tree containing k-smallest-set$(B_{middle}[2, m/2-1])$ and so on). The difficulty is that while we know that we need to remove $B_{middle}[0]$ from the k-smallest-set$(B_{middle}[0, m/2-1])$ , we do not know which element should be added back to it if $B_{middle}[0]$ was one of the $k$ smallest elements. We solve this by first learning how to construct k-smallest-set$(B_{middle}[0, m/2-1])$ from k-smallest-set$(B_{middle}[1, m/2-1])$. This can be done by the same procedure we described for $T_{right}$: first add $B_{middle}[0]$ to k-smallest-set$(B_{middle}[1, m/2-1])$ and then remove the maximum from this set if its size exceeds $k$. The key observation is that while doing this we can record the maximum $max$ that was removed. This allows us to "revert" the process in order to obtain k-smallest-set$(B_{middle}[1, m/2-1])$ from k-smallest-set$(B_{middle}[0, m/2-1])$: we add this $max$ and then remove $B_{middle}[0]$. Now we record these $max$'s during the preparation phase. The preparation costs are also distributed across the iterations so that the total complexity stays at $O(\log k)$ per invocation of update-window. The maintenance of $T_{left}$ is now done based on the $max$ array computed when $B_{left}$ was $B_{middle}$: we add the stored $max$ element and remove the element falling outside the window from $T_{left}$.

[2] Here and everywhere through paper we assume that $m$ is even to simplify the notation. If $m$ is odd one needs to update all the places where $m/2$ is used with $\lfloor m/2 \rfloor$ or $\lceil m/2 \rceil$ accordingly.

[3] We use a convention that if $W$ is shifted by a multiple of $m/2$ then $W_{left}$ is empty while $W_{right} = B_{right}$.

**Algorithm 1** The sliding window $k$-th smallest element algorithm.

**Input:** integers $k, m$

Initialize $T_{left}$, $T_{middle}$, $T_{right}$, and $T_{temp}$ as empty AWBBS trees.
Initialize incoming elements counter $j$ as 0.
Initialize $B_{left}, B_{middle}, B_{right}$ as arrays of size $m/2$.
Initialize $max$ and $max_{temp}$ as arrays of size $m/2 - k$.
**procedure** UPDATE-WINDOW($v$)
    ▷ General updates:
    $shift := j \bmod m/2$
    $j := j + 1$
    **if** $shift = 0$ **then**
        $max := max_{temp}$
        $T_{left} := T_{middle}$
        $T_{middle} := T_{right}$
        $B_{left} := B_{middle}$
        $B_{middle} := B_{right}$
        Set $T_{right}$ and $T_{temp}$ to empty tree and $B_{right}$ to a new array of size $m/2$
    **end if**
    $B_{right}[shift] := v$

    ▷ Maintaining $T_{right}$:
    $\mathsf{add}(T_{right}, v)$
    **if** $\mathsf{size}(T_{right}) > k$ **then**
        $\mathsf{remove}(T_{right}, \mathsf{max}(T_{right}))$
    **end if**

    ▷ Preparing $B_{middle}$:
    **if** $j > m/2$ **then**
        $\mathsf{add}(T_{temp}, B_{middle}[m/2 - 1 - shift])$
        **if** $shift \geq k$ **then**
            $max_{temp}[m/2 - 1 - shift] := \mathsf{max}(T_{temp})$
            $\mathsf{remove}(T_{temp}, max_{temp}[m/2 - 1 - shift])$
        **end if**
    **end if**

    ▷ Maintaining $T_{left}$:
    **if** $j > m$ **then**
        **if** $shift \leq m/2 - k - 1$ **then**
            $\mathsf{add}(T_{left}, max[shift])$
        **end if**
        $\mathsf{remove}(T_{left}, B_{left}[shift])$
    **end if**

    ▷ Producing the output (if $j \geq m$)
    Output $\mathsf{find\text{-}rank\text{-}three\text{-}trees}(T_{left}, T_{middle}, T_{right}, k)$
**end procedure**

▶ **Theorem 2.** *Algorithm 1 computes the sliding window $k$-th smallest element with time complexity $O(\log k)$ and space complexity $O(m)$.*

**Proof.** The algorithm splits the update-window method into four logical parts:

- (lines 6–17) The general part responsible for keeping the counters and proper wiring of the *left*,*middle*,*right* parts whenever we reach a block's boundary (when the *shift* variable is 0).
- (lines 19–23) The $T_{right}$ maintenance part.
- (lines 25–32) The $B_{middle}$ preparation part.
- (lines 34–40) The $T_{left}$ maintenance part.

The proof will show that at each iteration $T_{left}$, $T_{middle}$, and $T_{right}$ indeed contain the $k$ smallest elements of the corresponding window parts.

**Correctness.** We start by proving three lemmata about the state of $T_{left}$, $T_{middle}$, and $T_{right}$ in the update-window method.

▶ **Lemma 3.** *In the end of the method update-window at line 43 $T_{right}$ contains only k-smallest-set($B_{right}[0, shift]$).*

**Proof.** Consider an iteration where the right part of the sliding window is $B_{right}[0, shift]$. By construction we consecutively added elements $B_{right}[0], B_{right}[1], \ldots, B_{right}[shift]$ to $T_{right}$. Whenever the size of $T_{right}$ grew bigger than $k$ (line 21), we removed the largest element of $T_{right}$ (line 22) to bring the size of $T_{right}$ back to $k$. Hence $T_{right}$ contains only k-smallest-set($B_{right}[0, shift]$). ◀

▶ **Lemma 4.** *If $j > m/2$, in the end of the method update-window at line 43 $T_{middle}$ contains only k-smallest-set($B_{middle}[0, m/2 - 1]$).*

**Proof.** Note that $T_{middle}$ only changes at line 12 whenever the next $m/2$ elements have been consumed and *shift* equals to 0. Because of Lemma 3, we know that at such an iteration $T_{right}$ contains only k-smallest-set($B_{right}[0, m/2 - 1]$). Since $T_{middle}$ is assigned to $T_{right}$ at line 12 and $B_{middle}$ is assigned to $B_{right}$ at line 14, we conclude that $T_{middle}$ always contains only k-smallest-set($B_{middle}[0, m/2 - 1]$). ◀

▶ **Lemma 5.** *If $j > m$, in the end of the method update-window at line 43 $T_{left}$ contains only k-smallest-set($B_{left}[shift + 1, m/2 - 1]$) if $shift < m/2 - 1$ and is empty if shift is $m/2 - 1$.*

**Proof.** We start by proving a slightly different statement that at line 36 $T_{left}$ contains only k-smallest-set($B_{left}[shift, m/2 - 1]$). We prove this by induction on the *shift* variable. The base case of $shift = 0$ holds by construction since in this case $T_{left}$ has just been assigned to $T_{middle}$, while $B_{left}$ has been assigned to $B_{middle}$. Because of Lemma 4 it holds that $T_{middle}$ contains only k-smallest-set($B_{middle}[0, m/2 - 1]$), and hence $T_{left}$ contains only k-smallest-set($B_{left}[0, m/2 - 1]$) at line 36 when *shift* is 0. Assume now this holds for some $shift = t$ and we need to prove it for $shift = t + 1$. By the preparation phase at lines 25–32 $max[t]$ is the maximum element of k-smallest-set($B_{left}[t + 1, m/2 - 1]$) with the added $B_{left}[t]$. Hence, adding $max[t]$ to k-smallest-set($B_{left}[t, m/2 - 1]$) and removing $B_{left}[t]$ results in k-smallest-set($B_{left}[t + 1, m/2 - 1]$) which becomes the new value of $T_{left}$ after line 40 is completed. Finally, we observe that whenever *shift* becomes strictly bigger than $m/2 - k - 1$ then the size of the left part of the window becomes $\leq k$ and it is sufficient to always remove $B_{left}[shift]$ from $T_{left}$ in this part of the maintenance phase so that $T_{left}$ always contains k-smallest-set($B_{left}[shift, m/2 - 1]$).

The main lemma statement follows from observing that if at line 36 $T_{left}$ contains only k-smallest-set($B_{left}[shift, m/2-1]$), then at line 43 $T_{left}$ must contain only k-smallest-set($B_{left}[shift+1, m/2-1]$) for *shift* values $< m/2-1$ while for $shift = m/2-1$ the tree $T_{left}$ is empty. ◀

Now we have that $T_{left}$ consists of the $k$ smallest elements of $B_{left}[shift+1, m/2-1]$, $T_{middle}$ consists of the $k$ smallest elements of $B_{middle}[0, m/2-1]$, and $T_{right}$ consists of the $k$ smallest elements of $B_{right}[0, shift]$. Since find-rank-three-trees is correct we have that find-rank-three-trees will output a correct $k$-th smallest element of $T_{left}$, $T_{middle}$ and $T_{right}$ every time line 43 is executed.

**Complexity analysis.** In update-window we invoke max at most two times (lines 22, 29), add at most three times (lines 20, 27, 37), and remove at most three times (lines 22, 30, 39). While these operations take $O(\log k)$ time, all other operations are elementary (like assigning variables and objects) and take $O(1)$ time. Also, the specialized operation find-rank-three-trees takes $O(\log k)$ time (see Theorem 11). Hence, the time complexity of update-window is $O(\log k)$.

The space complexity is $O(m)$ since we have used $O(k)$ memory for the trees $T_{left}$, $T_{middle}$, $T_{right}$, $T_{temp}$; and $O(m)$ memory for storing the window parts $B_{left}, B_{middle}, B_{right}$ and auxiliary arrays $max$ and $max_{temp}$.

We also note that array assigning operations among $B_{left}, B_{middle}$ and $B_{right}$ can take $\Omega(m)$ time if implemented naively. Instead, whenever we assign arrays to each other we assign their pointers in $O(1)$ time and *do not* copy the array contents. Furthermore, we do not need to allocate a new memory for $B_{right}$ at line 15 – instead, it is sufficient to assign its pointer to the contents of $B_{left}$ which become unused otherwise. With this approach all array allocations happen once at lines 3 and 4 only. ◀

## 3.3 Discussing the Preparation Phase

Essentially, the preparation phase allows us to create a directly accessible view of every k-smallest-set($B_{middle}[i, m/2-1]$) for $i = 0, 1, \ldots, m-1$. First of all, we could have simplified the algorithm by running all the preparation steps at once whenever $B_{middle}$ becomes available and extract this functionality as a separate method. While the amortised time complexity of the algorithm would not change in this case, the worst case complexity would become $O(m \log k)$ instead of $O(\log k)$ which is undesirable. Second, instead of building an ad-hoc algorithm computing $max$ array used for "reverting" operations on AWBBS trees, we could have tried using the standard approach for persistent data structures [10] to build the sequence of AWBBS trees [21, 13, 23]. However, as explained in [17] such a generic approach would require copying the tree paths each time add/remove operation is invoked. This would lead to the suboptimal $O(m \log k)$ space complexity of the algorithm.

# 4 Lower Bounds

In order to prove lower bounds on the time complexity for the sliding window $k$-smallest element algorithms we give a reduction from them to sorting algorithms. Then, based on the well-known lower bounds for sorting algorithms we establish the lower bounds for the sliding window $k$-smallest element algorithms.

We give a reduction from computing the sliding window $k$-th smallest element to piecewise sorting. As opposed to the classical sorting piecewise sorting only sorts contiguous blocks of the input array without ensuring any order between the elements from different blocks. In

order to sort each contiguous block of size $k$ of an input array we will run a sliding window $k$-smallest element algorithm on this array with the caveat that when comparing elements from blocks $i$ and $j$ we use a custom comparison operator ensuring that all elements from block $i$ are smaller than the elements from block $j$ for $i < j$.

We note that our reduction is similar to the reductions that are used to prove the lower bounds on the sliding window median algorithms [11, 14, 17, 24]. While all these papers also present a reduction to sorting, the main difference to the reduction presented below is that all the previous reductions could only be adapted to work in a setting where $k$ is $\Theta(m)$ ($k = m/2$ is a concrete case of the median considered there), whereas the reduction presented in this paper works for arbitrary $k \leq m/2$.

▶ **Definition 6.** *An array $a$ is* $k$-piecewise sorted *if each contiguous $k$-size block $a[k \cdot i, k \cdot (i+1) - 1]$ is sorted.*

*An algorithm that takes an input array $a$ and outputs array $x$ which is $k$-piecewise sorted and each contiguous $k$-size block $x[k \cdot i, k \cdot (i+1) - 1]$ is a permutation of $a[k \cdot i, k \cdot (i+1) - 1]$ is called a* $k$-piecewise sorting algorithm.

**Algorithm 2** A reduction from sliding window $k$-smallest element to $k$-piecewise sorting.

**Input:** array of $n$ elements $a[0], a[1], \ldots, a[n-1]$

**Output:** array of $n$ elements $x[0], x[1], \ldots, x[n-1]$ where each consecutive $k$ size block $x[k \cdot i, k \cdot (i+1) - 1]$ is sorted and is a permutation of $a[k \cdot i, k \cdot (i+1) - 1]$.

1: Consider array $s$ of $n + m - 1$ elements where
   - ($k - 1$ prefix elements) $s[i] := (-\infty, -\infty)$ for $i = -1, -2, \ldots, -k + 1$;
   - ($n$ main elements) $s[i] := (\lfloor i/k \rfloor, a[i])$ for $i = 0, 1, \ldots, n - 1$;
   - ($m - k$ suffix elements) $s[i] := (+\infty, +\infty)$ for $i = n, n + 1, \ldots, n + m - k - 1$.

2: Define comparison operator on tuples in $s$ as: $(i, v) < (j, u)$ if $(i < j)$ or $(i = j$ and $v < u)$.

3: Invoke a sliding window $k$-th smallest algorithm on array $s$ (by feeding $s[-k + 1], s[-k + 2], \ldots, s[n + m - k - 1]$ one by one to its update-window interface). Denote the produced output as $y[0], y[1], \ldots, y[n - 1]$.

4: Let $x[i]$ be the second tuple entry of $y[i]$ for $i = 0, 1, \ldots, n - 1$. Output array $x$.

▶ **Lemma 7.** *In the regime where $m \geq 2k$, Algorithm 2 sorts each consecutive block of size $k$ of the input array in time $(n + m - 1) \cdot f(m, k) + O(n + m)$ where $f(m, k)$ is the time complexity of the underlying sliding window $k$-smallest element algorithm instantiated with the window size $m$ and the order statistics $k$.*

**Proof.** The core of the reduction lies in the construction of the intermediate array $s$ such that running a sliding window $k$-smallest element algorithm on it basically sorts the input array. The array $s$ elements are constructed at step 1 based on the input array elements $a$ with the feature that the elements from each contiguous block of size $k$ in $s$ are greater than the elements from the previous blocks. This is achieved by augmenting each element within the block with the block index and using the block index to compare the elements. Because the sliding window $k$-smallest element algorithm operates on windows of size $m$ and outputs the $k$-the smallest element, we also need to add $k - 1$ prefix elements in the beginning of $s$ and $m - k - 1$ suffix elements to its end, so that the sliding window algorithm operates correctly around the ends of the array.

**Correctness.** Consider any $k$-size contiguous block in the input array $a[k \cdot i, k \cdot (i+1) - 1]$. We claim that $y[k \cdot i + j]$ is the $(j+1)$-th smallest element in this array for $j = 0, 1, \ldots k-1$. By construction, $y[k \cdot i + j]$ corresponds to the $k$-th smallest element in the window $W = s[k \cdot (i-1) + j + 1, k \cdot (i-1) + j + m]$. We now define a partition of $W$ around the indices divisible by $k$ in blocks:

- we let the block $B_1$ contain all the window $W$ elements up to the $(k \cdot i)$-th index, i.e., $B_1 = s[k \cdot (i-1) + j + 1, k \cdot i - 1]$;
- all consecutive blocks have fixed size $k$, i.e., $B_2 = s[k \cdot i, k \cdot (i+1) - 1]$, $B_3 = s[k \cdot (i+1), k \cdot (i+2) - 1]$ and so on as long as we can take a full block of size $k$ within $W$.
- the last block contains the remaining elements of $W$.

Because $m \geq 2k$ we know that the second block $B_2 = s[k \cdot i, k \cdot (i+1) - 1]$ completely lies inside the sliding window $W = s[k \cdot (i-1) + j + 1, k \cdot (i-1) + j + m]$ and hence the window $W$ consists of at least $B_1$ and $B_2$.

By construction, elements in the earlier blocks are smaller than the elements in the successive blocks, e.g., all the elements in $B_1$ are smaller than the elements in $B_2$. This means that the $k$-th smallest element in $W$ is greater than all $k - j - 1$ elements in $B_1$, and is actually the $(j+1)$-th smallest element in $B_2$. The $(j+1)$-th smallest element in $B_2$ corresponds to the $(j+1)$-the smallest element in $a[k \cdot i, k \cdot (i+1) - 1]$.

**Complexity analysis.** The reduction takes $O(n+m)$ time to process elements at steps 1 and 4. Then, at step 3 the invocation of the underlying sliding window $k$-smallest element algorithm takes $f(m, k)$ time. Hence, the resulting time complexity is $(n+m-1) \cdot f(m, k) + O(n+m)$. ◀

We will now apply existing sorting lower bounds in the comparison model to obtain lower bounds for the sliding window $k$-th smallest element algorithms using the reduction we have just described in Lemma 7.

▶ **Theorem 8.** *Any algorithm computing the sliding window $k$-th smallest element has time complexity $\Omega(\log k)$ in the comparison model.*

**Proof.** Assume we have an algorithm that can solve sliding window $k$-th smallest element with time complexity $f(m, k)$. Take an arbitrary array of $n - m + 1$ elements where $m \leq n/2$. Then, based on the reduction in Lemma 7 we can $k$-piecewise sort this array in time $n \cdot f(m, k) + O(n)$. We know that $k$-piecewise sorting of $(n - m + 1)$-size array can only be done in $\Omega((n - m + 1) \cdot \log k)$ [15]. Given that $m \leq n/2$, we have that $n \cdot f(m, k) + O(n)$ must be in $\Omega(n \log k)$. This means that $f(m, k)$ is $\Omega(\log k)$. ◀

## 5 Conclusions and Future Work

In this paper we have presented the first optimal algorithm for finding the sliding window $k$-th smallest element with time complexity $O(\log k)$ and proved that this complexity is optimal. We note that the presented algorithm has fixed time complexity of $O(\log k)$ per newly read element which is independent of the window size $m$; it also requires only $O(m)$ memory storage.

The presented algorithm subsumes the existing results on the sliding window minimum/maximum algorithms [9, 25] (case $k = 1$) and median algorithms [14] (case $k = m/2$). One distinct feature of the specialized algorithms [9, 25, 14] is the possibility of their extreme concise representation which effectively fits into several lines of pseudocode. While the algorithm presented in this paper does share some ideas with these specialized algorithms (like maintaining an ordered tree of the window elements [14] and keeping only those window

elements that can potentially become the corresponding order statistics [9]), its pseudocode representation is still significantly larger. It would be interesting to investigate if all these sliding windows algorithms can be cast to a common framework with a simplified pseudocode representation.

## References


1 Andrei Alexandrescu. Fast deterministic selection. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.24.

2 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

3 Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In Catriel Beeri and Alin Deutsch, editors, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 286–296. ACM, 2004. doi:10.1145/1055558.1055598.

4 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16. ACM, 2002. doi:10.1145/543613.543615.

5 Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In Frank Neven, Catriel Beeri, and Tova Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 234–243. ACM, 2003. doi:10.1145/773153.773176.

6 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

8 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002. doi:10.1137/S0097539701398363.

9 Scott C. Douglas. Running max/min calculation using a pruned ordered list. *IEEE Trans. Signal Process.*, 44(11):2872–2877, 1996. doi:10.1109/78.542446.

10 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 109–121. ACM, 1986. doi:10.1145/12130.12142.

11 David Eppstein. Nontrivial algorithm for computing a sliding window median [online]. 2014. URL: https://cstheory.stackexchange.com/questions/21730/nontrivial-algorithm-for-computing-a-sliding-window-median.

12 Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 58–66. ACM, 2001. doi:10.1145/375663.375670.

13 Dana Jansens. Persistent binary search trees [online]. 2009. URL: https://cglab.ca/~dana/pbst/.

14 Martti Juhola, Jyrki Katajainen, and Timo Raita. Comparison of algorithms for standard median filtering. *IEEE Trans. Signal Process.*, 39(1):204–208, 1991. doi:10.1109/78.80784.

15 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973. URL: https://www.worldcat.org/oclc/310903895.

16 Georgiy Korneev. Designing an algorithm that searches for the kth largest element between two AVL trees [online]. 2016. URL: https://stackoverflow.com/questions/40473890/designing-an-algorithm-that-searches-for-the-kth-largest-element-between-two-avl.

17 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

18 Xuemin Lin, Hongjun Lu, Jian Xu, and Jeffrey Xu Yu. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 362–373. IEEE Computer Society, 2004. doi:10.1109/ICDE.2004.1320011.

19 Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. Scalable algorithms for large temporal aggregation. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 145–154. IEEE Computer Society, 2000. doi:10.1109/ICDE.2000.839401.

20 Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. In Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg, editors, *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 137–142. ACM, 1972. doi:10.1145/800152.804906.

21 Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999. doi:10.1017/s0956796899003494.

22 Mike Paterson. Progress in selection. In Rolf G. Karlsson and Andrzej Lingas, editors, *Algorithm Theory - SWAT '96, 5th Scandinavian Workshop on Algorithm Theory, Reykjavík, Iceland, July 3-5, 1996, Proceedings*, volume 1097 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 1996. doi:10.1007/3-540-61422-2_146.

23 Abhiroop Sarkar. Persistent red black trees in Haskell [online]. 2017. URL: https://abhiroop.github.io/Haskell-Red-Black-Tree/.

24 Jukka Suomela. Median filtering is equivalent to sorting. *CoRR*, abs/1406.1717, 2014. arXiv:1406.1717.

25 Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 66–77. ACM, 2017. doi:10.1145/3093742.3093925.

26 Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003. doi:10.1007/s00778-003-0107-z.

27 Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 128–136. ACM, 1982. doi:10.1145/800070.802185.


## A Finding the $k$-smallest Element in Three AWBBS Trees

We assume that each node of the input trees is *augmented* with its weight, i.e., contains the size of its subtree. Then, searching for the $k$-th smallest element in such a tree is straightforward: one traverses the tree from the root down to the leaves and at each step chooses the left or the right subtree depending on whether the $k$-th smallest element can

be there. The task of finding the $k$-th smallest element in multiple trees is similar: we simultaneously traverse all trees from their roots down to the leaves while choosing in which subtrees the $k$-th smallest element still can be. One needs to carefully analyze the traversal condition to make sure that the algorithm works in linear time in terms of the tree heights, since otherwise the incurred costs can lead to higher complexity (e.g., for two trees it can become quadratic [16]).

The algorithm that we present below can be seen as the fixed version of the algorithm from Lemma 2 of [17], where the condition 2 is corrected to continue the search instead of stopping. (Without this correction the algorithm of [17] won't work.)

## A.1 The Algorithm

We assume that there exists a function find-rank-one-tree which outputs the $k$-th smallest element in one tree in time $O(h)$ where $h$ is the height of the tree. At each step of our algorithm we maintain the remaining trees $T_1, T_2$ and $T_3$ where we continue to search for the $k$-th smallest element. Wlog, we assume that the root elements of the trees are ordered, i.e., the root $r_1$ of $T_1$ is smaller than the root $r_2$ of $T_2$ which is smaller than the root $r_3$ of $T_3$. Let $S$ denote the elements of the left subtrees $L_1$, $L_2$ and $L_3$. By construction, we know that the rank of $r_1$ in the union of $T_1$, $T_2$ and $T_3$ is at most $|S| + 1$, whereas the rank of $r_3$ in the union of $T_1$, $T_2$ and $T_3$ is at least $|S| + 3$. These inequalities allow us to identify the subtree where the element of rank $k$ *cannot* be: if $|S| + 3 > k$, then the $k$-th smallest element cannot be $r_3$ and cannot be in $R_3$; if $|S| + 3 \leq k$, then $r_1$ and elements in $L_1$ are all smaller than the $k$-th smallest element and hence can be discarded. We continue this traversal until one of the trees becomes empty, then we apply the same reasoning for the two remaining trees only. Finally, once we are left with a single tree we use find-rank-one-tree to output the $k$-th smallest element in the remaining tree.

**Algorithm 3** find-rank-three-trees finds the $k$-smallest element in three trees.

**Input:** trees $T_1, T_2, T_3$ and the target rank $k$.
**Output:** $k$-th smallest element in $T_1, T_2$ and $T_3$.

1: *target-rank* $:= k$
2: **while** True **do**
3: **if** all but one tree are empty **then**
4: return *target-rank*-th smallest element in the non-empty tree by calling find-rank-one-tree.
5: **end if**
6: Let $T_1, T_2, \ldots, T_\ell$ denote the non-empty trees left
7: Let $L_j, R_j, r_j$ be the left subtree, the right subtree, and the root of $T_j$, respectively
8: Wlog, assume $r_1 < r_2 < \cdots < r_\ell$ (otherwise, rename the trees)
9: **if** $\sum_{j=1}^{\ell} \mathsf{size}(L_j) + \ell >$ *target-rank* **then**
10: $T_\ell := L_\ell$
11: **else**
12: $T_1 := R_1$
13: *target-rank* $:=$ *target-rank* $- \mathsf{size}(L_1) - 1$
14: **end if**
15: **end while**

▶ **Lemma 9.** *Algorithm 3 finds the $k$-th smallest element in $T_1, T_2$ and $T_3$ in $O(h_1 + h_2 + h_3)$ time where $h_1, h_2$ and $h_3$ are the tree heights of $T_1, T_2$ and $T_3$, respectively.*

**Proof.**

**Correctness.** Let $T_j^i$ denote the state of the trees at the beginning of the while loop at step 2 at iteration $i$. That is, $T_1^0, T_2^0$ and $T_3^0$ represent the initial input trees, $T_1^1, T_2^1$ and $T_3^1$ represent the trees after one iteration, and so on. Similarly, let *target-rank*$^i$ denote the state of the target rank variable at the beginning of the $i$-th iteration. We prove correctness by validating the following invariant:

▷ Claim 10. The *target-rank*$^i$-th smallest element among $T_1^i, T_2^i$ and $T_3^i$ is the *target-rank*$^{i+1}$-th smallest element among $T_1^{i+1}, T_2^{i+1}$ and $T_3^{i+1}$.

Proof. Let $L_j^i, R_j^i, r_j^i$ denote the corresponding loop variables during the iteration $i$. Consider two cases:

- $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell >$ *target-rank*$^i$ : Because the rank of $r_\ell^i$ is at least $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell$, and $\ell \geq 2$ the element with rank *target-rank*$^i$ must be smaller than $r_\ell^i$ and all the elements in $R_\ell^i$. Hence, we can continue searching for the element with rank *target-rank*$^i$ in $T_1, \ldots, T_{\ell-1}$ and the left subtree of $T_\ell$ which is $L_\ell$. This is implemented in line 10.
- $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell \leq$ *target-rank*$^i$ : Because the rank of $r_1^i$ is at most $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + 1$, and $\ell \geq 2$ the element with rank *target-rank*$^i$ must be greater than $r_1^i$ and all the elements in $L_1^i$. Hence, we can continue searching for the element with rank *target-rank*$^{i+1}$ = *target-rank*$^i - \mathsf{size}(L_1^i) - 1$ in $T_2, T_3, \ldots, T_\ell$ and the right subtree of $T_1$ which is $R_1$. This is implemented in lines 12 and 13. ◁

As long as more than two trees $T_1^i, T_2^i, \ldots T_\ell^i$ are non-empty we search for the right element in them. Once all but one tree are empty we will output the element with the correct rank because find-rank-one-tree is correct.

**Complexity analysis.** Every time the while loop (lines 2 to 15) is executed either $T_1$'s, $T_2$'s, or $T_3$'s height is decreased by 1. Hence, the maximum number of loop executions is $h_1 + h_2 + h_3$. Furthermore, whenever only one of the trees is non empty the loop returns the output of the find-rank-one-tree call. One invocation of find-rank-one-tree takes time $O(h)$ where $h$ is the height of the non-empty tree during the invocation. By construction $h$ is smaller than $h_1, h_2$ and $h_3$. This means that the overall time complexity is $O(h_1 + h_2 + h_3)$. ◀

Instantiating Algorithm 5 in the setting where the binary search trees are weight-balanced we obtain the desired result.

▶ **Theorem 11.** *Algorithm 5 finds the $k$-th smallest element in $O(\log n)$ time where $T_1, T_2$ and $T_3$ are AWBBS trees of size at most $n$.*

**Proof.** The theorem follows by combining the result of Lemma 9 and the observation that the height of $T_1, T_2$ and $T_3$ is $O(\log n)$. ◀