# Greedily Computing Associative Aggregations on Sliding Windows[☆]

David Basin, Felix Klaedtke, Eugen Zălinescu

*Institute of Information Security, ETH Zurich, Switzerland*

**Abstract**

We present a greedy algorithm that combines with an associative operator elements of subsequences of a sequence of data items. The subsequences are given by a window that slides over the sequence of data items, where the window size may vary while it slides. The number of applications of the associative operator carried out by our algorithm is minimal.

*Keywords:* Greedy algorithm, complexity, optimality, sliding window

## 1. Introduction

*Problem Statement.* Let $\oplus : D \times D \to D$ be an associative operator. Consider a sequence $\bar{a} = (a_1, \ldots, a_n)$ of elements in $D$, with $n \geq 1$. A *window* $w$ in $\bar{a}$ is a pair $(\ell, r)$ with $1 \leq \ell \leq r \leq n$. We denote the *left margin* of a window $w$ by $\ell_w$ and the *right margin* by $r_w$, i.e., $\ell_w$ is $w$'s first component and $r_w$ its second component. Furthermore, we write $\oplus_w(\bar{a})$ for $a_{\ell_w} \oplus a_{\ell_w+1} \oplus \cdots \oplus a_{r_w}$.

We consider the following problem in which the number of applications of the $\oplus$ operator should be minimized.

**Input:** A nonempty sequence $\bar{a}$ of elements in $D$ and a sequence $\bar{w} = (w_1, \ldots, w_k)$ of windows in $\bar{a}$, with $\ell_{w_1} \leq \ell_{w_2} \leq \cdots \leq \ell_{w_k}$ and $r_{w_1} \leq r_{w_2} \leq \cdots \leq r_{w_k}$.

**Output:** The sequence $\big(\oplus_{w_1}(\bar{a}), \oplus_{w_2}(\bar{a}), \ldots, \oplus_{w_k}(\bar{a})\big)$.

The objective of minimizing the applications of $\oplus$ is motivated in settings where $\oplus$'s computation is expensive, e.g., when taking the union of large finite sets and when multiplying large matrices.

A straightforward but suboptimal algorithm to the problem computes $\oplus_{w_i}(\bar{a})$ for each window $w_i$ separately. It is easy to see that this algorithm applies the $\oplus$ operator $\sum_{i=1}^{k}(r_{w_i} - \ell_{w_i})$ times. One can do better by sharing intermediate results between overlapping windows. For illustration, consider the following example.

*Example.* Let $D$ be the domain $\mathbb{N}$ and $\oplus$ integer addition. For the sequence $\bar{a} = (2, 4, 5, 2)$ and the window sequence $\bar{w} = \big((1,3),(1,4),(2,4)\big)$, the output is the sequence $(11, 13, 11)$. The naive algorithm applies the $\oplus$ operator $2 + 3 + 2 = 7$ times. For this example, the minimal number of $\oplus$ applications is 3, since integer addition is associative and commutative and the windows $w_1$ and $w_3$ contain the same integers. However, the minimal number is 4 if we only exploit the associativity of $\oplus$.

Obviously, when computing $\oplus_{w_2}(\bar{a})$ we can reuse the result of the window $w_1$, since $\oplus_{w_2}(\bar{a}) = \oplus_{w_1}(\bar{a}) \oplus a_4$. If we compute the intermediate result $h := a_3 \oplus a_4$ when computing the result for the window $w_2$, we could reuse it for the window $w_3$, since $\oplus_{w_3}(\bar{a}) = a_2 \oplus h$. Note that we do not have $h$ as an intermediate result when computing the results of the previous windows $w_1$ and $w_2$ as $(a_1 \oplus a_2) \oplus a_3$ and $\big((a_1 \oplus a_2) \oplus a_3\big) \oplus a_4$, respectively. In case we compute the results of the windows $w_1$ and $w_2$ as $a_1 \oplus (a_2 \oplus a_3)$ and $a_1 \oplus \big(a_2 \oplus (a_3 \oplus a_4)\big)$, $h$ is available for the result of the window $w_3$. However, in this case, the computation of the result of the window $w_2$ does not use the result of the first window. So, it is important how we parenthesize $a_i \oplus a_{i+1} \oplus \cdots \oplus a_j$ when computing the result of a window. This choice has an impact on whether we can reuse intermediate results for other windows.

*Overview.* In Section 2, we present an algorithmic solution for the problem. Our algorithm processes the windows iteratively and reuses intermediate results from previously processed windows. It is greedy in the sense that it minimizes for each window the number of $\oplus$ applications. In Section 3, we prove its correctness. In Section 4, we show that it has linear running time in the length of the input sequence $\bar{a}$. In Section 5, we prove the algorithm's optimality with respect to the number of $\oplus$ applications. We conclude in Section 6 with discussing applications and related work.

## 2. Algorithm

We present our algorithm in a functional programming style, close to the Ocaml programming language [7].[1] To simplify the exposition, we fix the associative operator $\oplus : D \times D \to D$ and the input sequence $\bar{a} = (a_1, \ldots, a_n)$, i.e., we treat $\oplus$ and $\bar{a}$ as global variables. Our pseudo

[☆]This work was supported by Google Inc.

[1]Implementations in Ocaml and Haskell are provided as supplementary material (Appendix A and Appendix B).

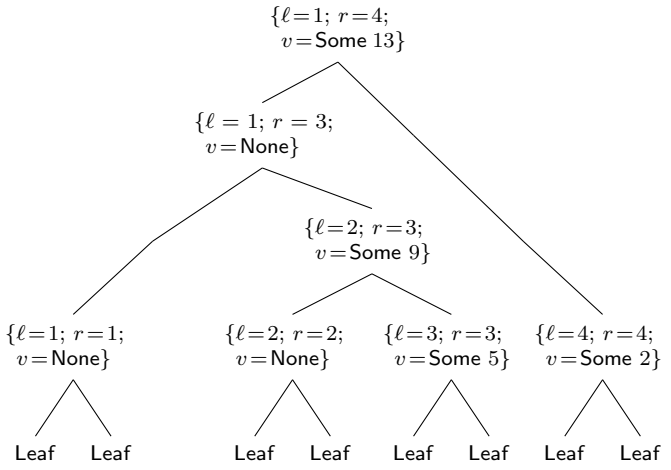


Figure 1: Instance of the tree data structure.

code can easily be modified so that $\oplus$ and $\bar{a}$ are algorithm parameters. Furthermore, we assume that we can access $\bar{a}$'s element at position $i$ in constant time.

We use binary ordered trees to store and reuse intermediate results, which are updated when iteratively processing the window sequence $\bar{w}$. The polymorphic datatype of these trees is

```
type 'a intermediate = 'a option node tree
```

where

```
type 'b node = {ℓ: ℕ; r: ℕ; v: 'b}
type 'c tree =
  | Leaf
  | Node of ('c * ('c tree) * ('c tree))
```

Figure 1 shows the tree built by our algorithm for the window $w_2 = (1, 4)$ for the input from the example in the introduction.

The content of an inner node, which represents some subtree $t$, is a record whose field values are denoted by $\ell_t$, $r_t$, and $v_t$, respectively. The field values $\ell_t, r_t$ are elements of $\mathbb{N}$, with $1 \leq \ell_t \leq r_t \leq n$. Intuitively, they describe $\bar{a}$'s elements that are covered by the tree $t$ and their combination $\oplus_{(\ell_t, r_t)}(\bar{a})$ is the field value $v_t$. If we know that the intermediate result $\oplus_{(\ell_t, r_t)}(\bar{a})$ is not reused in later iterations, we do not store it to reduce memory usage. In this case $v_t$ is actually None; otherwise, $v_t$ is Some $\oplus_{(\ell_t, r_t)}(\bar{a})$.

We lift the $\oplus$ operator in the canonical way to this extended domain. For $t =$ Leaf, we define $\ell_t := r_t := 0$ and $v_t :=$ None. Furthermore, we define the following function for extracting the children of a tree's root:

```
fun children t = match t with
                   | Leaf → error "No children at leaf."
                   | Node (_, t', t'') → (t', t'')
```

We first define two basic auxiliary functions for creating and combining trees. The function singleton $i$ builds the single-node tree $t$ with $\ell_t = r_t = i$ and $v_t =$ Some $a_i$.

```
fun singleton i = Node ({ℓ = i; r = i; v = Some a_i}, Leaf, Leaf)
```

The function combine $t'$ $t''$ builds the tree $t$ with the left child $t'$ and the right child $t''$ if neither $t'$ nor $t''$ is a leaf. The value at $t$'s root is $v_{t'} \oplus v_{t''}$. The field values $\ell_t$ and $r_t$ are obtained straightforwardly from the field values of its left and right children. If $t'$ is the tree Leaf then $t$ is $t''$. Analogously, if $t''$ is the tree Leaf then $t$ is $t'$.

```
fun discharge t = match t with
                    | Leaf → Leaf
                    | Node (n, t', t'') → Node ({ℓ = n.ℓ; r = n.r;
                                                 v = None}, t', t'')

fun combine t' t'' = match (t', t'') with
                       | (Leaf, _) → t''
                       | (_, Leaf) → t'
                       | (_, _) → Node ({ℓ = ℓ_{t'}; r = r_{t''};
                                         v = v_{t'} ⊕ v_{t''}},
                                        discharge t', t'')
```

Note that in the case where $t'$ and $t''$ are not the trees Leaf, the value of the left child of the tree $t$ is discharged by the homonymous function and becomes None.

The working horse of our algorithm is the function slide $t$ $w$. It updates the tree $t$ for the window $w$:

```
fun singletons i j =
  if i > j then [] else (singleton j) :: singletons i (j − 1)

fun reusables t w =
  if ℓ_w > r_t then []
  else if ℓ_w = ℓ_t then [t]
  else let (t', t'') = children t
       in if ℓ_w ≥ ℓ_{t''} then reusables t'' w else t'' :: reusables t' w

fun slide t w =
  let ts  = singletons (max ℓ_w (r_t + 1)) r_w
      ts' = reusables t w
      swap f x y = f y x
  in fold_left (swap combine) Leaf (ts @ ts')
```

The auxiliary function singletons $i$ $j$ returns the list of single-node trees for the elements $a_i, a_{i+1}, \ldots, a_j$. The auxiliary function reusables $t$ $w$ returns the list of maximal subtrees $t'$ of $t$ for which the value $v_{t'}$ can be used for computing the value $\oplus_w(\bar{a})$. The function slide $t$ $w$ combines both these lists of trees to a single tree for the window $w$ by folding the list $ts$ @ $ts'$, where @ denotes list concatenation. Note that we need to swap the order of the arguments of combine when building the tree, since singletons and reusables add trees to the front. Figure 2 shows the skeletons of the trees without their leaves that our algorithm builds for the input sequence $(2, 4, 5, 2)$ and the windows $(1, 3)$, $(1, 4)$, and $(2, 4)$. Nodes with value None are depicted as circles instead of black dots. The skeletons of the subtrees that are reused from the previous window are depicted with dashed lines.

Our algorithm iteratively calls the function slide $t_{i-1}$ $w_i$, for $i = 1, \ldots, k$, where $ws$ is the list consisting of the given windows $w_1, \ldots, w_k$, $t_0$ is the tree Leaf, and $t_i$ is the tree returned by slide $t_{i-1}$ $w_i$. For $i \in \{1, \ldots, k\}$, the value at the root of the tree $t_i$ is $\oplus_{w_i}(\bar{a})$, which we extract from the tree in the $i$th iteration and add to the returned list after updating the tree $t_{i-1}$.

```
fun extract t = match t with
```

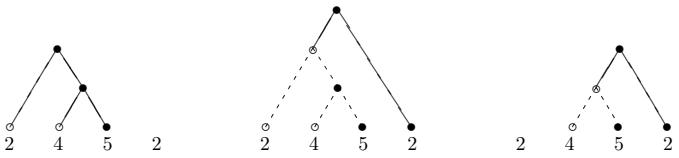


Figure 2: Skeletons of the trees built by the slide function for the window sequence $\big((1,3),(1,4),(2,4)\big)$.

```
        | Leaf → error "No value at leaf."
        | Node (n, _, _) →
          match n.v with
            | None → error "No value at node."
            | Some v → v

fun iterate t ws = match ws with
                     | [] → []
                     | w :: ws' → let t' = slide t w
                                  in (extract t') :: iterate t' ws'

fun sliding_window ws = iterate Leaf ws
```

## 3. Correctness

The algorithm presented in Section 2 obviously terminates. It successively processes the windows $w_1, \ldots, w_k$ in the list $ws$. In particular, in the function iterate, we process the window at the head of the window list $ws$ and proceed with the list's tail until it is empty.

In the following, we prove partial correctness. A tree $t$ is *correctly shaped* if the following conditions are satisfied, where $\hat{t}$ ranges over $t$'s subtrees distinct from Leaf and $\hat{t}'$ and $\hat{t}''$ are the left child and the right child of $\hat{t}$:

(S1) $\ell_{\hat{t}} \leq r_{\hat{t}}$.

(S2) If $\ell_{\hat{t}} = r_{\hat{t}}$ then $\hat{t}' = \hat{t}'' = \mathsf{Leaf}$.

(S3) If $\ell_{\hat{t}} < r_{\hat{t}}$ then $\hat{t}', \hat{t}'' \neq \mathsf{Leaf}$, $\ell_{\hat{t}} = \ell_{\hat{t}'}$, $r_{\hat{t}} = r_{\hat{t}''}$, and $r_{\hat{t}'} + 1 = \ell_{\hat{t}''}$.

A tree $t$ is *correctly valued* if the following conditions are satisfied, where $\hat{t}$ ranges over $t$'s subtrees distinct from Leaf:

(V1) If $v_{\hat{t}} \neq \mathsf{None}$ then $v_{\hat{t}} = \mathsf{Some} \oplus_{(\ell_{\hat{t}}, r_{\hat{t}})} (\bar{a})$.

(V2) If $\hat{t}$ is a right child then $v_{\hat{t}} \neq \mathsf{None}$.

(V3) If $t \neq \mathsf{Leaf}$ then $v_t \neq \mathsf{None}$.

A tree is *valid* if it is correctly shaped and correctly valued.

We prove the following lemma about the function slide $t$ $w$, where $w$ is the window for which we update the tree $t$.

**Lemma 1.** *Let $w$ be a window and $t$ a valid tree with $\ell_t \leq \ell_w$ and $r_t \leq r_w$. The tree $t'$ returned by* slide $t$ $w$ *is valid and $(\ell_{t'}, r_{t'}) = (\ell_w, r_w)$.*

*Proof.* We first introduce the following notion. A list $ts$ of trees is *adjacent* for $(\ell, r)$ with $\ell, r \in \mathbb{N}$ if the following conditions are satisfied:

(L1) All trees in $ts$ are distinct from Leaf.

(L2) If the trees $t_1$ and $t_2$ are next to each other in $ts$ with $t_1$ appearing before $t_2$ then $\ell_{t_1} - 1 = r_{t_2}$.

(L3) If $ts \neq []$ then $r_{t_1} = r$ and $\ell_{t_2} = \ell$, where $t_1$ is the first tree in $ts$ and $t_2$ is the last tree in $ts$.

Note that the empty list is adjacent for any $(\ell, r)$. If the singleton list consisting of the tree $t$ is adjacent for $(\ell, r)$ then $t = t_1 = t_2$, where $t_1$ and $t_2$ are the trees in the condition (L3).

The lemma follows straightforwardly from the following facts about the functions that are used by slide $t$ $w$ for building the tree $t'$:

(a) The list returned by reusables $t$ $w$ is adjacent for $(\ell_w, r_t)$ and its elements are valid trees.

(b) The list returned by singletons (max $\ell_w$ $(r_t + 1)$) $r_w$ is an adjacent list for $(\max\{\ell_w, r_t + 1\}, r_w)$ of valid trees.

(c) Let $ts$ be an adjacent nonempty list for $w$ of valid trees. The tree $t'$ returned by fold_left (swap combine) Leaf $ts$ is a valid tree with $(\ell_{t'}, r_{t'}) = w$.

We only prove (a) and (c); (b) is obvious.

We prove (a) by induction over the size of $t$. Note that all the elements of $ts$ are obviously correctly shaped since $ts$ only contains subtrees of $t$, which are correctly shaped. Similarly, properties (V1) and (V2) hold for the trees in $ts$, because they hold for $t$.

The base case $t =$ Leaf is trivial, since the returned list $ts$ is the empty list. For the step case, suppose that $t \neq$ Leaf. The cases where $\ell_w > r_t$ and $\ell_w = \ell_t$ are obvious, since $ts$ is either the empty list or the singleton list consisting of the tree $t$, respectively. For the other cases, we have that $\ell_t < \ell_w \leq r_t$. Let $t'$ be the left child of $t$ and let $t''$ be the right child of $t$. For $\ell_w \geq \ell_{t''}$, it follows from the induction hypothesis that the function reusables $t''$ $w$ returns an adjacent list for $(\ell_w, r_{t''})$. This concludes the case since $r_t = r_{t''}$. For $\ell_w < \ell_{t''}$, it follows from the induction hypothesis that reusables $t'$ $w$ returns an adjacent list $ts'$ for $(\ell_w, r_{t'})$. Putting $t''$ to the front of $ts'$ results in an adjacent list for $(\ell_w, r_t)$, because we have that $r'_t + 1 = \ell_{t''}$ from the fact that $t$ is correctly shaped. As $t''$ is a right child and $t$ is valid, we have from (V2) that $v_{t''} \neq$ None. It follows that (V3) holds for $t''$. Thus $t''$ is correctly valued and this concludes the case.

In the remainder of the proof, we show (c). We first remark that fold_left (swap combine) Leaf $ts$ is equivalent to fold_left (swap combine) $h$ $ts'$, where $h$ is the head of $ts$ and $ts'$ its tail. Note that $ts'$ is an adjacent list for $(\ell_w, \ell_h - 1)$. We define $r_{ts'}$ as $\ell_w - 1$ if $ts'$ is the empty list and as $r_{t_1}$ otherwise, where $t_1$ is the first tree in $ts'$.

It suffices to prove that for every valid tree $z$ distinct from Leaf with $(\ell_z, r_z) = (r_{ts'}+1, r_w)$, the tree $s$ returned by fold_left (swap combine) $z$ $ts'$ is valid and $(\ell_s, r_s) = (\ell_w, r_w)$. We use induction over the length of $ts'$. The base case is trivial since $s = z$. For the step case, suppose that $h$ is the head of $ts'$ and $ts''$ its tail. Composing $z$ with $h$ results

in a valid tree $z'$ with $(\ell_{z'}, r_{z'}) = (\ell_h, r_z)$. The list $ts''$ is adjacent for $(\ell_w, \ell_h - 1)$. As $r_{ts''} + 1 = \ell_h$, it follows that $(\ell_{z'}, r_{z'}) = (r_{ts''}+1, r_w)$. Using the induction hypothesis for fold_left (swap combine) $z'$ $ts''$ concludes the step case. □

The correctness of the algorithm follows easily from Lemma 1. Note that we assume that the given windows $w_1, \ldots, w_k$ always slide to the right over the sequence $\bar{a}$, i.e., $\ell_{w_i} \leq \ell_{w_{i+1}}$ and $r_{w_i} \leq r_{w_{i+1}}$, for all $i \in \{1, \ldots, k-1\}$. Hence, in each iteration of the algorithm, the assumptions of Lemma 1 are satisfied.

**Theorem 2.** *Let $\bar{a}$ be a nonempty sequence of elements in $D$ and $ws$ the list consisting of the windows $w_1, \ldots, w_k$. The function* slide_window *$ws$ returns a list consisting of the elements $\oplus_{w_1}(\bar{a}), \oplus_{w_2}(\bar{a}), \ldots, \oplus_{w_k}(\bar{a})$.*

## 4. Complexity

In this section, we determine the time and space used by our algorithm. We ignore the actual cost of applying the $\oplus$ operator, i.e., we assume that its application takes $\mathcal{O}(1)$ time and space. We use the following notation. The *size* of a window $w$ is $\|w\| := r_w - \ell_w + 1$ and $\|t\|$ denotes the *size* of the tree $t$, i.e., the number of its subtrees.

We first analyze the space and time complexity of a single iteration of our algorithm, i.e., the space and time for computing the tree $t'$ returned by slide $t$ $w$, where $w$ is a window and $t$ is a valid tree with $\ell_t \leq \ell_w$ and $r_t \leq r_w$. For building the tree $t'$, slide $t$ $w$ determines the list $ts$ of single-node trees for the new elements in the window $w$. This is done by singletons (max $\ell_w$ $(r_t + 1)$) $r_w$ and takes $\mathcal{O}(\|w\|)$ time. The length of $ts$ is at most $\|w\|$. Furthermore, slide $t$ $w$ determines the list $ts'$ of subtrees of $t$ that are reusable by executing reusables $t$ $w$. The length of the list $ts'$ is at most $\mathcal{O}(\|w\|)$, since each subtree in $ts'$ covers at least one and distinct elements in $w$. Since we visit each node in $t$ at most once, it takes $\mathcal{O}(\|t\|)$ time to compute the list $ts'$. Folding the concatenated list $ts$ @ $ts'$ takes $\mathcal{O}(\|w\|)$ time and space, resulting in the tree $t'$ with $\|t'\| \in \mathcal{O}(\|w\|)$. Overall, slide $t$ $w$ runs in $\mathcal{O}(\|w\| + \|t\|)$ time and space.

From this upper bound for a single iteration, we obtain for our algorithm the upper bounds $\mathcal{O}(km)$ for time and $\mathcal{O}(m)$ for space, when processing the windows $w_1, \ldots, w_k$, where $m := \max\{\|w\| \mid 1 \leq i \leq k\}$. However, the upper bound on the running time does not take into account that our algorithm reuses intermediate results. With an amortized complexity analysis [8], we establish next a linear-time upper bound when the windows are pairwise distinct.

We fix the algorithm's input: let $\bar{a}$ be a sequence of $n \geq 1$ elements in $D$ and $ws$ the list of windows $w_1, \ldots, w_k$, with $k \geq 1$ and $w_i \neq w_{i+1}$, for all $i \in \{1, \ldots, k\}$. Furthermore, let $t_0, t_1, \ldots, t_k$ be the trees that the algorithm successively builds for the windows $w_1, \ldots, w_k$, where $t_0 =$ Leaf. That is, $t_i$ is the output of slide $t_{i-1}$ $w_i$, for $i \in \{1, \ldots, k\}$.

**Lemma 3.** *The number $k$ of windows is in $\mathcal{O}(n)$.*

*Proof.* To see this, consider the relative movements of consecutive windows in the sequence, i.e., for $i \in \{1, \ldots, k\}$, let $\ell_i^\delta := \ell_{w_i} - \ell_{w_{i-1}}$ and $r_i^\delta := r_{w_i} - r_{w_{i-1}}$, where $\ell_{w_0} := r_{w_0} := 0$. Since the windows are pairwise distinct, we have that $\ell_i^\delta > 0$ or $r_i^\delta > 0$, for every $i \in \{1, \ldots, k\}$. If $k > 2n$ then $\sum_{i=1}^k \ell_i^\delta > n$ or $\sum_{i=1}^k r_i^\delta > n$, which contradicts $\ell_{w_k} \leq r_{w_k} \leq n$. Hence $k \leq 2n$. □

We say that a node built by combine is *good* if it is obtained from the new elements of a window, that is, from the list $ts$ in slide. A node build by combine is *bad* if it is obtained from the list $ts'$ of reusables trees in slide. We say that a tree is good or bad if its root is good or bad, respectively.

**Lemma 4.** *For any $i \in \{1, \ldots, k\}$, every bad node that is new in the tree $t_i$ (that is, not a node in the tree $t_{i-1}$), except at most one, has a good left child.*

*Proof.* We prove by induction on $i$ that the tree $t_i$ satisfies the following properties:

(a) A good node does not have bad children.

(b) At most one right bad node has at its left another right bad node.

The base case, when $i = 1$, is straightforward, as all internal nodes are new good nodes. For the inductive case, assume $t_{i-1}$ satisfies (a) and (b). Let $(s_1, \ldots, s_m)$ be the list of trees returned by reusables $t_{i-1}$ $w_i$. Note that $s_1$ is the right-most tree with respect to the sequence $\bar{a}$. There are $m$ new bad nodes in $t_i$, each of them having a reusable tree as the left child. We show first that at most two reusable trees are bad. Suppose that in $t_{i-1}$ there are three bad reusable trees $s_{j_1}$, $s_{j_2}$, and $s_{j_3}$ with $j_1 < j_2 < j_3$. Each reusable tree is a right child. Hence $s_{j_3}$ is a right bad child at the left of $s_{j_2}$ and $s_{j_2}$ is a right bad child at the left of $s_{j_1}$. This contradicts property (b).

Let $s'_1, \ldots, s'_m$ be the new bad trees in $t_i$ corresponding to the reusable nodes $s_1, \ldots, s_m$ in $t_{i-1}$, respectively. That is, $s'_i$ is the node that has $s_i$ as a left child and $s'_{i-1}$ as a right child when $i > 1$. Property (b) holds trivially if there are no bad reusables trees, and straightforwardly, by induction hypothesis when there is exactly one bad reusable tree.

Suppose there are two bad reusables trees, say $s_j$ and $s_{j'}$ with $j < j'$. Then $s'_1, \ldots, s'_{j-2}$ are new bad right nodes. However, at their left they only have good trees. The same holds for $s'_j, \ldots, s'_{j'-2}$ and $s'_{j'}, \ldots, s'_{m-1}$. Consider now the node $s'_{j-1}$. $s_j$ may have a right bad child but no left bad child (by induction hypothesis). However, the right child of $s_{j'}$ cannot be bad, because this would contradict property (b) on $t_{i-1}$ (be more precise).

Hence, there is at most one right bad node, namely $s'_{j-1}$ which has at its left another right bad node, namely the right child of $s_j$. □

**Lemma 5.** *The number of applications of the $\oplus$ operator is in $\mathcal{O}(n)$.*

*Proof.* The number of applications of the $\oplus$ operator is given by the number of calls to combine, which is equal to the number of good and bad trees. There are exactly $(r_1 - \ell_1) + \sum_{i=2}^{k}(r_i - r_{i-1}) = r_k - \ell_1$ good trees, that is, at most $n-1$. By Lemma 4, in every tree $t_i$, the left child of each bad tree, except at most one, is a good tree. Moreover, each tree can be the left child of a node at most once. This is because if a tree $t$ is the left child of a node build by combine, then the root value of $t$ is discharged, that is, it becomes None, and thus $t$ cannot be again an argument of combine. Hence the number of bad trees is at most the number of good trees plus $k$ (one for each iteration). That is, there are at most $2n-2+k$ good and bad trees. Thus, by Lemma 3, we get that the number of applications of the $\oplus$ operator is at most $4n-2$. □

We now state the main complexity result.

**Theorem 6.** *Let $\bar{a}$ be a nonempty sequence of $n$ elements in $D$ and ws the list consisting of the windows $w_1, \ldots, w_k$ with $w_i \neq w_{i+1}$, for all $i \in \{1, \ldots, k\}$. The function* slide_window *ws runs in $\mathcal{O}(n)$ time.*

*Proof.* The running time slide_window is linear in the total number of calls to combine and reusables. From Lemma 5, the number of calls to combine is in $\mathcal{O}(n)$.

It remains to prove that the number of calls to reusables is also in $\mathcal{O}(n)$. First note that reusables calls itself recursively at most once, and the argument is one of the children of the current argument. Hence the number of calls to reusables in slide $t_{i-1}$ $w_i$ equals the length of the path from the root of $t_{i-1}$ to the root of $s_m$, where $(s_1, \ldots, s_m)$ is the list of trees returned by reusables $t_{i-1}$ $w_i$. Note that $s_m$ is the tree returned by the last call to reusables in iteration $i$, and also the left-most tree with respect to the sequence $\bar{a}$. We show next that any two paths from different iterations share no edge. Consider first the paths $p$ and $q$ for two consecutive iterations $i$ and $i+1$, respectively. Path $p$ in $t_{i-1}$ does not have edges in the trees $s_1, \ldots, s_m$, while the edges of the path $q$ in $t_i$ are new edges (that is, edges not in $t_i$) and possibly edges in one of the reusables trees $s_1, \ldots, s_m$. The case of non-consecutive iterations is similar and simpler, as the paths are even more far apart.

Hence the total number of calls to reusables is bound by the number of edges created during the run of the algorithm. In each iteration, the number of new edges is twice the number of new nodes, that is, twice the number of calls to combine. By Lemma 5, this number is linear in $n$. □

## 5. Optimality

We prove the optimality of our algorithm by contradiction. Assume that it is not optimal for the window sequence $\bar{w} = (w_1, \ldots, w_k)$. Without loss of generality, we assume that $k$ is the minimal number for which our algorithm is not optimal, i.e., for all window sequences $(w'_1, \ldots, w'_{k'})$ with $k' < k$, our algorithm is optimal. Recall that $a_1, \ldots, a_n$ are the elements in the input sequence $\bar{a}$ and $w_i = (\ell_{w_i}, r_{w_i})$, for $i \in \{1, \ldots, k\}$.

In the following, let $t_0, t_1, \ldots, t_k$ be the trees that are iteratively built by our algorithm for the windows $w_1, \ldots, w_k$. Recall that $t_0$ is the tree Leaf. Furthermore, let $s_0, s_1, \ldots, s_k$ be trees that are optimal for the windows $w_1, \ldots, w_k$, where $s_0$ is the tree Leaf.

We define the following measure on trees $t, t'$. Let $\mathrm{cost}(t, t')$ be the number of subtrees $\hat{t}'$ of $t'$ with $r_{\hat{t}'} - \ell_{\hat{t}'} \geq 1$ and there is no subtree $\hat{t}$ of $t$ with $\ell_{\hat{t}} = \ell_{\hat{t}'}$ and $r_{\hat{t}} = r_{\hat{t}'}$. Note that $\mathrm{cost}(t, t')$ is the number of $\oplus$ operations that are necessary to build $t'$ by reusing intermediate results from $t$, where we assume that the value for each subtree in $t$ is available, even when the actual stored value is None. In particular, we have that $\sum_{i=1}^{k} \mathrm{cost}(t_{i-1}, t_i)$ is the number of $\oplus$ operations that are performed by our algorithm and $\sum_{i=1}^{k} \mathrm{cost}(s_{i-1}, s_i)$ is the number of $\oplus$ operations for the given optimal solution.

By the non-optimality assumption we have that

$$\sum_{i=1}^{k} \mathrm{cost}(t_{i-1}, t_i) < \sum_{i=1}^{k} \mathrm{cost}(s_{i-1}, s_i)\,.$$

Since $k$ is minimal, we also have that

$$\sum_{i=1}^{k-1} \mathrm{cost}(t_{i-1}, t_i) \geq \sum_{i=1}^{k-1} \mathrm{cost}(s_{i-1}, s_i)\,.$$

Intuitively speaking, the non-optimality of our algorithm is caused when building the tree $t_k$ for the last window $w_k$.

In the following, let $\hat{t}_1, \ldots, \hat{t}_p$, with $p \geq 0$, be the reusable subtrees of $t_{k-1}$. Furthermore, let $\hat{t}_{p+1}$ be the subtree in $t_k$ for the new elements in the window $w_k$, i.e., $\hat{t}_{p+1}$ stores the value $\oplus_{(\ell,r)}(\bar{a})$, with $\ell = \max\{\ell_k, r_{k-1}+1\}$ and $r = r_k$. Note that

$$\mathrm{cost}(t_{k-1}, t_k) = p + r_k - \ell\,.$$

It follows that

$$\mathrm{cost}(s_{k-1}, s_k) < p + r_k - \ell\,.$$

This inequality can only hold when there are $q$ subtrees of $s_{k-1}$, with $q < p$ that can be reused for building the tree $s_k$. For $p \in \{0, 1\}$, this is not possible.

In the remainder of the proof, we assume that $p \geq 2$. The following properties hold for the trees $\hat{t}_1, \ldots, \hat{t}_p$. See also Figure 3 for an illustration.

(i) Each subtree $\hat{t}_i$ combines the new elements of a window $w_{j_i}$, with $1 \leq j_i \leq k-1$. Note that the windows $w_{j_1}, \ldots, w_{j_p}$ are different, in particular, we have that $j_1 < j_2 < \cdots < j_p$. This is easy to see: if $j_i = j_{i+1}$, for some $i \in \{1, \ldots, p-1\}$, our algorithm would build a tree with $\hat{t}_i$ as a left child and $\hat{t}_{i+1}$ as a right child. This tree would be reusable when building the tree for $w_k$. Furthermore, we have that $\ell_{w_{j_1}} \leq \cdots \leq \ell_{w_{j_p}}$ and $r_{w_{j_1}} < \cdots < r_{w_{j_p}} = r_{w_{k-1}}$. Finally, note that $w_{j_p}$ can be the window $w_{k-1}$.

(ii) Each subtree $\hat{t}_i$ is the right child of a subtree $\check{t}_i$ of $t_{k-1}$, where $\ell_{\check{t}_i}$ is less than $\ell_k$. The existence of $\check{t}_i$ follows from $\ell_{w_{k-1}} < \ell_{w_k}$. Otherwise, if $\ell_{w_{k-1}} = \ell_{w_k}$, there is only a single tree ($p = 1$), which contradicts the assumption $p \geq 2$. Note that the trees are $\check{t}_{i-1}, \ldots, \check{t}_1$ are subtrees of $\check{t}_i$.

We further observe that for every $i \in \{1, \ldots, p\}$, the tree $\check{t}_i$ occurs also as a subtree in each of the trees $t_{j_i}, \ldots, t_{j_p}$. To see this, assume that $\check{t}_i$ does not occur in $t_{j_{i'}}$, for some $i' \in \{i, \ldots, p\}$. Let $i'$ be maximal. Obviously, $i' \neq p$ since $r_{w_{j_p}} = r_{w_{k-1}}$ and $\ell_{w_{j_i}} \leq \ell_{w_{k-1}}$. For $i' < p$, $\hat{t}_{i'}$ would be a reusable subtree when building the tree $t_{i'+1}$. Our algorithm would build a subtree of $t_{i'+1}$ with the left child $\hat{t}_{i'}$ and the right child $\hat{t}_{i'+1}$. This is contradicts the fact that $\hat{t}_{i'}$ is the right child of the subtree $\check{t}_{i'}$ of $t_{k-1}$.

Next, we show that we can assume that for each $i \in \{1, \ldots, p\}$, there is a tree $\hat{s}_i$ with $\ell_{\hat{s}_i} = \ell_{\hat{t}_i}$ and $r_{\hat{s}_i} = r_{\hat{t}_i}$ that appears as a subtree in the trees $s_{j_i}, \ldots, s_{j_p}$. Assume that such a tree $\hat{s}_i$ does not exist. When building the tree $s_{j_i}$, we need to combine the new elements in the window $w_{j_i}$. If we also combine it with old elements, no $\oplus$ operations are saved. Furthermore, for $i = 1$, we cannot reuse $\hat{s}_1$ to build $s_k$. Overall, it is not more expensive to build trees with the claimed property.

It follows that $\hat{s}_1, \ldots, \hat{s}_p$ are subtrees of $s_{j_p}$ and therefore also subtrees of $s_{k-1}$. If there is a subtree in $s_{k-1}$ that has as children two of these adjacent trees, say $\hat{s}_i$ and $\hat{s}_{i+1}$, then for its combination at least one additional application of the $\oplus$ operator is needed. Note that the tree $\hat{s}_i$ is a right child of a subtree of the tree $s_i$.

It follows that if we have $q$ reusable subtrees in $s_{k-1}$ to build $s_k$, then

$$\sum_{i=1}^{k-1} \mathrm{cost}(s_{i-1}, s_i) \geq (p - q) + \sum_{i=1}^{k-1} \mathrm{cost}(t_{i-1}, t_i)\,.$$

From this inequality, we obtain a contradiction to the non-optimality assumption of our algorithm:

$$\begin{aligned}
&\sum_{i=1}^{k} \mathrm{cost}(s_{i-1}, s_i) = \\
&\sum_{i=1}^{k-1} \mathrm{cost}(s_{i-1}, s_i) + (q + r_k - \ell) \geq \\
&(p - q) + \sum_{i=1}^{k-1} \mathrm{cost}(t_{i-1}, t_i) + (q + r_k - \ell) = \\
&\sum_{i=1}^{k-1} \mathrm{cost}(t_{i-1}, t_i) + (p + r_k - \ell) = \\
&\sum_{i=1}^{k} \mathrm{cost}(t_{i-1}, t_i)\,.
\end{aligned}$$

## 6. Applications and Related Work

The presented algorithm can be easily modified so that it solves the online version of the problem, where the input sequences $\bar{a}$ and $\bar{w}$ are iteratively given. In iteration $i$, the input is the window $w_i$ and the remaining elements of the sequence $\bar{a}$ up to $a_{r_{w_i}}$. The output of the $i$th iteration is $\oplus_{w_i}(\bar{a})$. In the online version of the problem, we do not restrict ourselves to finite sequences, i.e., $\bar{a}$ and $\bar{w}$ can be infinite. However, we require that the windows still have finite size, i.e., the right margin $r_w$ of a window $w$ cannot be $\infty$. Our algorithm, in particular the online version of it, has applications in areas like system monitoring (see, e.g., [4, 2]) and stream processing (see, e.g, [1]).

The *sliding-window-minimum problem* is a special instance of the problem considered in this article. It additionally assumes an ordering on the data element from $D$ and $\oplus$ returns the minimum of its arguments. An algorithmic solution to this problem where the window size is constant and the window always slides by one over the sequence of data items is described by Harter [5]. Harter's algorithm runs in $\mathcal{O}(n)$ time and uses $\mathcal{O}(m)$ space, where $n$ is the length of the input sequence $\bar{a}$ and $m$ is the window size. Lemire [6] presents a minimum-maximum filter, which is similar to Harter's algorithm. Lemire provides a detailed analysis of his algorithm. In particular, he shows that it performs at most three comparisons per element.

Approximation algorithms for computing statistics or evaluating aggregation queries over sliding windows in data-stream processing have received attention in the last decade. See [3], for a seminal paper in this area, in which the authors present and analyze an approximation algorithm for the *basic-counting problem*, i.e., for a given data stream consisting of 0s and 1s, maintain at every time instant the count of the number of 1s in the last $k$ elements. When restricting the memory usage, their algorithm estimates the answer at every instant within a certain bound. They prove that their algorithm is optimal in terms of memory usage and show how their algorithm extends to richer problems.

## References


[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[2] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monitoring data usage in distributed systems. *IEEE Trans. Software Eng.*, to appear.

[3] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[4] A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, 2010.

[5] R. Harter. The minimum on a sliding window algorithm. Blog entry `http://richardhartersworld.com/cri/2001/old_slidingmin.html`, 2001. Latest update of webpage May 13, 2009.

[6] D. Lemire. Streaming maximum-minimum filter using no more than three comparisons per element. *Nord. J. Comput.*, 13(4):328–339, 2006.

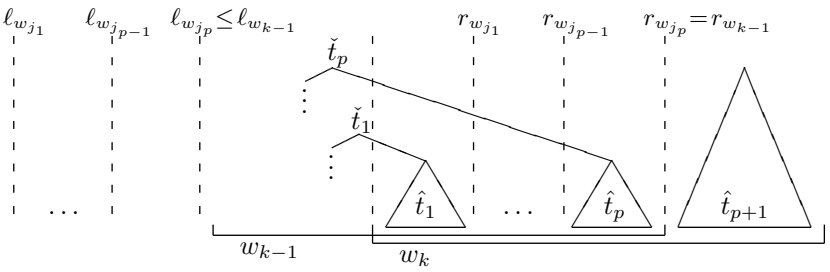


Figure 3: Building the tree $t_k$ from $t_{k-1}$.


[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system (release 3.12)*. Institut National de Recherche en Informatique et en Automatique (INRIA), July 2011. `http://caml.inria.fr`.

[8] R. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.

## Appendix A. Supplementary Material: Ocaml Source Code

```
(* Some general auxiliary functions *)

let swap f x y = f y x

let lift f x y = match x, y with
  | None, _          → None
  | _, None          → None
  | Some x', Some y' → Some (f x' y')

let rec drop n xs = match xs with
  | []       → []
  | hd :: tl → if n > 0 then drop (n-1) tl else xs

let rec take n xs = match xs with
  | []       → []
  | hd :: tl → if n > 0 then hd :: take (n-1) tl else []

let split_at n xs = (take n xs, drop n xs)

(* Datatypes for the sliding window algorithm *)

type 'a node = {left: int; right: int; v: 'a}
type 'a tree =
  | Leaf
  | Node of ('a * ('a tree) * ('a tree))
type 'a intermediate = 'a option node tree

(* Selection functions for datatype *)

let left_index = function
  | Leaf           → -1
  | Node (n, _, _) → n.left
let right_index = function
  | Leaf           → -1
  | Node (n, _, _) → n.right
let value = function
  | Leaf           → None
  | Node (n, _, _) → n.v

let extract t = match value t with
  | None   → invalid_arg "No value at node."
  | Some v → v
let children = function
  | Leaf              → invalid_arg "No children at leaf."
  | Node (_, t', t'') → (t', t'')

(* Auxiliary functions for the sliding window algorithm *)

let singleton (i,x) = Node ({left = i; right = i; v = Some x}, Leaf, Leaf)

let rec singletons xs i j = match xs with
  | []       → []
  | hd :: tl → if i > j then [] else (singleton (i, hd)) :: singletons tl (i+1) j

let discharge = function
  | Leaf              → Leaf
  | Node (n, t', t'') → Node ({left = n.left; right = n.right; v = None}, t', t'')

let c = ref 0
```

```
let combine op t' t'' = match t', t'' with
  | Leaf, _ → t''
  | _, Leaf → t'
  | _, _    → incr c;
    Node ({left = left_index t'; right = right_index t''; v = (lift op) (value t') (value t'')},
                    discharge t', t'')

let rec reusables t l =
  if l > right_index t then []
  else if l = left_index t then [t]
  else let (t', t'') = children t in
       if l ≥ left_index t'' then reusables t'' l
       else t'' :: reusables t' l

let slide op xs t (l,r) =
  let reuses       = reusables t l in
  let (news, rems) = split_at (1+r - (max l (1 + right_index t))) xs in
  let news'        = singletons news (max l (1 + right_index t)) r in
  (rems, List.fold_left (swap (combine op)) Leaf ((List.rev news') @ reuses))

(* Sliding window algorithm *)

let rec iterate op xs t = function
  | []          → []
  | (l,r) :: ws → let zs        = if right_index t < l then drop (l - 1-right_index t) xs else xs in
                  let (xs', t') = slide op zs t (l,r) in
                  (extract t') :: iterate op xs' t' ws

(* Arguments:
   (1) associative operator op : 'a → 'a → 'a
   (2) list [x_0; ...; x_n-1] of data elements xs : 'a list
   (3) list [(l_0,r_0); ...; (l_k-1,r_k-1)] of windows ws : (int * int) list
         with l_0≤l_2≤...≤l_k-1 and r_0≤r_1≤...≤r_k-1 and 0≤l_i≤r_i<n, for all i = 0, ..., k-1
   Return value: [y_0; ...; y_k-1] : 'a list
     with y_i = x_{l_i} op x_{l_i+1} op ... op x_{r_i}, for all i = 0, ..., k-1
*)

let sliding_window op xs ws = iterate op xs Leaf ws

let _ =
  let xsize = 100000 in
  let xs =
    let rec add i l =
      if i < xsize then
        1 :: add (i+1) l
      else
        l
    in
    add 0 []
  in
  let wsize = 100 in
  let ws =
    let rec add i (l,r) =
      let ldelta = Random.int 3 in
      let newsize = Random.int wsize in
      if l + ldelta + newsize < xsize then
        let new_window = (l + ldelta, max r (l + ldelta + newsize)) in
        new_window :: (add (i+1) new_window)
      else
        begin
          Printf.printf "There are %d windows.%!\n" i;
          []
```

```
      end
    in
    add 0 (1,wsize)
  in

  let _ = sliding_window (+) xs ws in

  Printf.printf "There were %d operations\n" !c
```

## Appendix B. Supplementary Material: Haskell Source Code

```
module Sliding (slidingWindow) where

-- Some general auxiliary functions

swap f x y = f y x

lift f (Just x) (Just y) = Just (f x y)
lift f _        _        = Nothing

-- Datatypes for the sliding window algorithm

data Tree a =  Leaf | Node a (Tree a) (Tree a)
data Label a = Label {left :: Int, right :: Int, val :: a}

type Intermediate a = Tree (Label (Maybe a))
type Window = (Int,Int)

-- Selection functions for datatypes

leftIndex Leaf         = -1
leftIndex (Node l _ _) = left l
rightIndex Leaf         = -1
rightIndex (Node l _ _) = right l
value Leaf         = Nothing
value (Node l _ _) = val l

extract t = case value t of
              Nothing → error "No value at tree's root."
              Just v  → v
children Leaf            = error "No children at leaf."
children (Node _ t' t'') = (t', t'')

-- Auxiliary functions for the sliding window algorithm

singleton (i,x) = Node l Leaf Leaf
  where l = Label {left = i, right = i, val = Just x}

discharge Leaf            = Leaf
discharge (Node l t' t'') = Node l' t' t''
  where l' = Label {left = left l, right = right l, val = Nothing}

combine _  t'    Leaf  = t'
combine _  Leaf  t''   = t''
combine op t'    t''   = Node l (discharge t') t''
  where l = Label {left = leftIndex t', right = rightIndex t'', val = op (value t') (value t'')}

reusables :: Tree (Label a)          -- tree t
             → Int                   -- left bound l
             → [Tree (Label a)]      -- list of t's maximal reusables subtrees
reusables t l
```

```
  | l >  rightIndex t  = [] -- case could be checked before calling reusables in slide
  | l == leftIndex t   = [t]
  | l ≥  leftIndex t'' = reusables t'' l
  | otherwise          = t'' : reusables t' l
    where (t', t'') = children t

slide :: (Maybe a → Maybe a → Maybe a)       -- lifted associative operator op
         → [Intermediate a]                  -- list of singleton trees ys
         → Intermediate a                    -- tree of previous window t
         → Window                            -- window (l,r)
         → ([Intermediate a], Intermediate a) -- remaining elements and updated tree
slide op ys t (l,r) = (rems, t')
    where (news, rems) = splitAt (1+r - (max l (1 + rightIndex t))) ys
          reuses       = reusables t l
          t'           = foldl (swap (combine op)) Leaf ((reverse news) ++ reuses)

-- Sliding window algorithm

slidingWindow :: (a → a → a)  -- associative operator op
                 → [a]        -- list of elements xs = [x_0, ..., x_n-1]
                 → [Window]   -- list of windows ws = [(l_0,r_0), ..., (l_k-1,r_k-1)]
                 → [a]
-- Conditions on windows: l_0≤l_2≤...≤l_k-1 and r_0≤r_1≤...≤r_k-1 and 0≤l_i≤r_i<n,
--                        for all i = 0, ..., k-1
-- Return value: [y_0, ..., y_k-1] :: [a]
--               with y_i = x_{l_i} (op) x_{l_i+1} (op) ... (op) x_{r_i}, for all i = 0, ..., k-1
-- Note: the lists xs and ws can also be infinite

slidingWindow op xs ws = iterate singletons Leaf ws
  where singletons          = map singleton (zip [0..] xs)
        iterate _  _ []     = []
        iterate ys t (w:ws) = let zs        = drop ((fst w) - 1-rightIndex t) ys
                                  (ys', t') = slide (lift op) zs t w
                              in
                                extract t' : iterate ys' t' ws
```