An algorithm for writing scheduled modifications to finite byte sequences

Jens Ebert

February 18, 2018

Abstract

This paper describes a simple, algorithm with quadratic complexity for modifying a finite sequence of bytes by a series of modifications, namely insertions, removes and replaces. The order of changes is not relevant except the case of multiple insertions at the same offset. The paper defines rules for valid modification series. For a valid modification series, the algorithm calculates the disjoint subsets of the byte sequence that need to be moved to another offset, as indicated by the modifications. The algorithm ensures that modifications are executed at the correct offsets and that existing bytes of the sequence not covered by a modification are never overwritten. The paper specifically covers an aspect relevant for practical implementation: Block-wise reading and writing of existing and new bytes.

1 Introduction

The most common use case of byte sequences are files on any platform. A *file* is basically a finite, ordered sequence of bytes with a clearly defined start and end. It is irrelevant for us that operating system implementations might fragment the bytes on hard disk, we always consider it as a fully consecutive chunk, as it transparently appears to all programming APIs. Another example of a byte sequence is a chunk of in-memory bytes, in some programming languages called *byte array*.

Both examples of byte sequences have one thing in common: They are inherently static. Inserting a new byte in the middle of the sequence means to first shift the bytes behind by the according distance towards higher offsets, then add the new insertion bytes starting at the insertion offset. It is quite similar with removes inside a byte sequence. All primitive file I/O APIs as well as byte arrays in most programming languages are working like that. To make this easier, an API on top is required.

Common use cases are modifications of files to write changed data, e.g. when changing the metadata of an audio or video file, which typically requires insertions, removes or overwrites. The problem with all this is of course: Performance. When inserting new bytes at the beginning of a large file, all bytes behind it must be read and rewritten at the correct higher offset. Of course, reading large byte sequences requires corresponding amounts of memory,

so a chunk-wise reading and writing might be necessary. Some of these issues have been adressed by different approaches:

- Some descriptive metadata formats such as ID3v1 and ID3v2 store their metadata at the end or near the end of files; thus reading bytes behind when modifying them is unnecessary
- Some data formats, e.g. metadata tag formats, use so-called *padding* bytes; these bytes have no meaning, but are just junk fillers that work like a crush zone: If new bytes are inserted before, the corresponding bytes from the padding must be removed, and if bytes before the padding are removed, the corresponding number of bytes must be added to the padding bytes. This way, the overall size of a tag remains stable and reading and writing of any bytes behind the tag is not necessary. For this to work, of course the number of padding bytes must be suitable for the corresponding insertions.

The contents of a byte sequence is in no way just consisting of unstructured individual bytes that line up linearly. They have a structure and consist of disjoint objects with a meaning, as shown in figure 1. In the figure we see a typical use case: A header with a so-called magic key starts at a given offset and has a fixed size. It is followed by a series of so-called frames of varying sizes, and finally a footer with a fixed size. Frames can be inserted or removed from the payload, or existing frames are changed, thereby shrinking or growing.

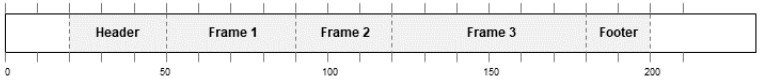

Figure 1: Most binary data formats are object-oriented and structure byte chunks correspondingly

The idea of the algorithm presented in this paper is to encapsulate all reads and writes necessary to implement byte sequence changes such that bytes are correctly written without data loss[1]. This is only one part of it. The second aspect is that it allows to schedule multiple (non-contradicting) modifications for a given byte sequence, executing these modifications in the correct order.

Figure 2 shows an example: Here we not only have an insertion of 10 bytes, but also a remove of 40 bytes at a later offset. What needs to be done to implement this?

1. First, the bytes between the insert offset and the remove offset are read and moved forward by 30 bytes - this essentially overwrites the first 10 bytes of the range to be removed, thus not leading to data loss
2. Second, the insertion bytes are written at the insertion offset
3. Third, the bytes behind the end of the removed range are read and moved backward by 10 bytes

These steps can also be done in another order: Step 3 can be done first, then step 1 and then step 2, or first step 1, then step 3, finally step 2. The only limitation is that step 2 must be done after step 1 to prevent overwriting the bytes between the insert and remove offset.

If there are more than just two modifications and adding the need to read and write chunk-wise, it gets more complicated to figure out a correct order of operations.

This paper presents an algorithm to find a correct execution plan for an arbitrary number of modifications to a byte sequence. This algorithm is good enough for implementing flushing of multiple modifications effectively, yet it leaves open some questions, probably to be answered by further follow-up papers: How many distinct correct execution plans exist? Among these, is there an optimal execution plan in sense of e.g. cache usage? If so, what is it and how to find it?

[1] Data loss might occur because in practical implementations, modifications are not written to a copy of the original byte sequence, but the original byte sequence is usually modified in-place.

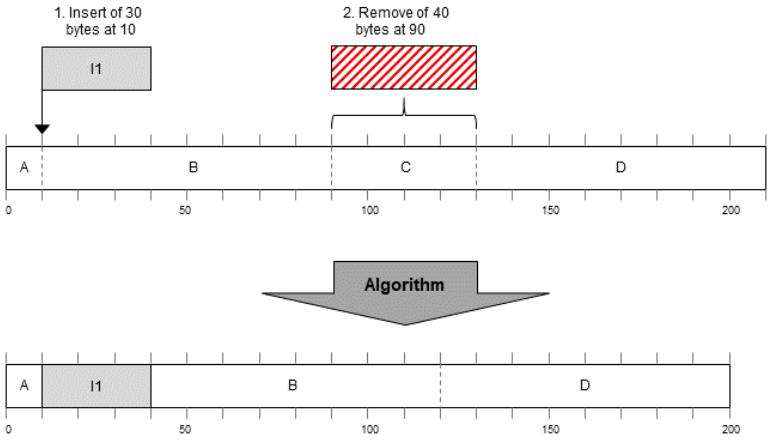

Figure 2: An example for two modifications on the same byte sequence

2 Definitions

2.1 Byte Sequences

In this paper, a **byte** is the smallest adressable unit to read and write. A byte has a **value**. It does not matter for this paper which values a byte can take. Two bytes are the same if they have the same value.

A **byte sequence** is a finite sequence of bytes of **length** $\text{len}(S) \in \mathbb{N}_0$, defining a linear chain of bytes. It assigns a zero-based index to each byte. We write byte sequences with upper-case letters:

$$S = (b_0, b_1, \ldots, b_{\text{len}(S)-1}).$$

The zero-based index of a byte in a sequence is called the **offset** of the byte. Offsets of value $\text{len}(S)$ are allowed and can be considered as pointing to the end of the sequence.

The **empty byte sequence** has a length of 0 and is written as $E := ()$.

Two byte sequences are said to be equal if they contain the same bytes in the same order.

A sequence T is called a **byte sub-sequence** of S, if $T \subseteq S$ and T is defined as:

$$T := (b_i, b_{i+1}, \ldots, b_k) \text{ for } i \leq k,$$

i.e. it contains a sequence of bytes from S, preserving their order and values. If the start and end offsets of a sub-sequence of S are important, the paper uses the following notation:

$$S[a, b) := (b_a, b_{a+1}, \ldots, b_{b-1}), a \leq b,$$

where $S[a, a) := E$.

The set of all possible finite byte sequences is written as $\mathcal{S}$ in this paper.

We also need the **concatenation operation** defined as:

$$+ : \mathcal{S} \times \mathcal{S} \to \mathcal{S},$$
$$(b_0, \ldots, b_{n-1}) + (b'_0, \ldots, b'_{m-1}) \mapsto (b_0, \ldots, b_{n-1}, b'_0, \ldots, b'_{m-1}),$$

i.e. it joins two byte-sequences in order.

2.2 Modification Series

A **modification** m_k is a potential change of a byte sequence, where the index $k \in \mathbb{N}_0$ is the order of applying the modification to the byte sequence, which is detailed in 3.2.

A modification has the following properties - all of them also indexed by k:

- The start and end offsets s_k and e_k of the modification in the byte sequence S, $0 \leq s_k \leq e_k \leq \text{len}(S)$.
- A byte sequence of insertion bytes $I_k^{l_k} := (i_0, \ldots, i_{l-1})$ of length l_k, for which also the empty sequence is allowed.

We write $m_k[s_k, e_k, I_k^{l_k}]$ if the properties of the modification m_k are relevant in a given context.

Based on the properties, a classification of modifications is possible:

- `INSERT` $\text{ins}_k[s_k, I_k^l] := m_k[s_k, s_k, I_k^{l_k}]$ with $l_k > 0$: Insert a non-empty byte sequence $I_k^{l_k}$ at offset s_k into the byte sequence S.
- `REMOVE` $\text{rem}_k[s_k, e_k] := m_k[s_k, e_k, ()]$ with $e_k > s_k$: Remove a non-empty byte sub-sequence $S[s_k, e_k)$ at offset s_k from the byte sequence S, without inserting any additional bytes.
- `REPLACE` $\text{rep}_k[s_k, e_k, I_k^{l_k}] := m_k[s_k, e_k, I_k^{l_k}]$ with $e_k > s_k$ and $l_k > 0$: Replace a non-empty byte sub-sequence $S[s_k, e_k)$ at offset s_k from the byte sequence S with the non-empty insertion bytes $I_k^{l_k}$.

The set of all possible modifications is called M.

A **modification series** $mod^n := (m_0, \ldots, m_{n-1}) \in M^n$ is a finite, ordered sequence of n modifications.

A modification series is called **valid** if it follows all rules defined in section 3.2.

2.3 Execution Plan and Transformation Function

The algorithm in this paper creates and executes an **execution plan** for performing modifications to byte sequences correctly. We briefly define what that means here.

Using the definitions in the previous sections, the algorithm can be said to define a **transformation function**

$$f : \mathcal{S} \times M^n \to \mathcal{S}, (S, mod^n) \mapsto f(S, mod^n) = T$$

that transforms an input byte sequence S combined with a modification series mod^n into a target byte sequence T. Of course it is only a function if an execution plan to do this exists. We show this by listing a corresponding algorithm in this paper. It is clear that f must follow some rules:

- The length of the target sequence T must be

$$\text{len}(T) = \text{len}(S) + \sum_{k=0}^{n-1} l_k - \sum_{k=0}^{n-1} (e_k - s_k),$$

  i.e. it is the length of the original byte sequence plus the number of inserted bytes minus the number of removed bytes.
- $S[e_k, s_{k+1}) \subseteq T$ for each $0 \leq k \leq n-1$, where $s_n := \text{len}(S)$, i.e. the bytes between the modifications are still present in T.
- $I_k^{l_k} \subseteq T$ for each $0 \leq k \leq n-1$, i.e. the inserted byte sequences are contained in T.
- We cannot explicitly require $S[s_k, e_k) \not\subseteq T$ for each $0 \leq k \leq n-1$. This criterion need not be met in any case. The reason is that S might contain another non-removed sub-sequence which is equal to a removed sub-sequence.

These criteria are descriptive, but not sufficient to define a correct algorithm, e.g. nothing is said about the order of the expected sub-sequences in T. Assuming that the modifications in mod^n are already correctly sorted (according to 3.2), we define f with concatenation as **correct** iff:

$$T := S[0, s_0) + \sum_{k=0}^{n-1} \left(I_k^{l_k} + S[e_k, s_{k+1}) \right), \text{ where } s_n := \text{len}(S),$$

which states that T contains all insertion bytes and the bytes between the modifications in the correct order, and it does not contain the removed sub-sequences $S[s_k, e_k)$.

3 Algorithm Description

3.1 Algorithm Overview

The input of the algorithm is as follows:

- A byte sequence $S \in \mathcal{S}$
- A modification series $mod^n \in M^n$
- A maximum read-write block size $r \in \mathbb{N}$ which indicates the chunking of reads and writes

The result of the algorithm is a correctly transformed byte sequence T with the properties defined in section 2.3.

Here, we give an overview of the basic phases of the algorithm that are further detailed in the follow-up sections:

1. **Scheduling Phase:** The modification series mod^n is validated and prepared for processing. If it is invalid, the algorithm terminates already in this phase without modifying S, so $T = S$ in this case.
2. **Plan Creation Phase:** An execution plan of read- and write operations in correct order is created.
3. **Execution Phase:** The created plan is executed on S to create T.

3.2 Scheduling Phase

The purpose of the scheduling phase is to validate the modification series $mod^n \in M^n$ for the byte sequence S, and to prepare it for the follow-up phases.

A modification series $mod^n \in M^n$ is **valid**, if the following conditions are met for all modifications $m_k, 0 \leq k \leq n-1$:

1. $0 \leq s_k, e_k \leq \text{len}(S)$
2. $s_k, e_k \notin (s_j, e_j)$ for $j \neq k$, i.e. the modification intervals $[s_j, e_j)$ of pairwise differing modifications do not overlap each other.

Note that validity rule (2) prevents overlapping `REMOVE`s or `REPLACE`s as well as `INSERT`s happening within a removed or replaced range. Allowing this would complicate algorithm and implementation, and would require a definition of the meaning of such overlapping operations, thus we exclude it here. Also note that using an open interval means that $s_k = s_j$, $e_k = e_j$ as well as $e_k = s_j$ is allowed for $j \neq k$, i.e. `INSERT`s can happen at the same offset as other `INSERT`s, `REMOVE`s or `REPLACE`s. Furthermore, a modification m_k can start where another modification m_j ends.

The algorithm terminates if the modification series mod^n is invalid.

Otherwise, the second part of the scheduling phase starts, whose purpose is to prepare the modification series for the follow-up phases. We assume here that changes are scheduled in no specific order (the scheduling order), which is reflected in mod^n. The task of part two of the scheduling phase is to sort mod^n in a clearly defined way. It also answers the question for the meaning of `INSERT`s at the same offset $s_k = s_j$. In this paper, we define that an `INSERT` scheduled before an `INSERT` with the same offset is executed first[2]. This essentially means the insertion bytes of the `INSERT` that has first been scheduled are inserted first into the resulting byte sequence T, followed by the insertion bytes of the second scheduled `INSERT` at this offset and so on.

The concrete sorting criteria are as follows: Given $m_k[s_k, e_k, I_k^{l_k}] \in mod^n, 0 \leq k < n$, the second part of the scheduling phase sorts these modifications as follows:

1. If $s_k > s_j$ for $j \neq k$, m_k is executed before m_j
2. If $s_k = s_j$ for $j \neq k$, m_k is executed before m_j if $k < j$

[2] Of course, with the same right, it could also have the meaning of prepending inserts instead. The meaning should be defined as appropriate for the use case.

This defines a total ordering on mod^n and an output modification series $\overline{mod^n}$, containing the same modifications in "offset-ascending order". Note that due to the validation part of this phase, sort criterion (2) can only apply to `INSERT`s at the same offset as another `INSERT`, `REMOVE` or `REPLACE`.

3.3 Plan Creation Phase

The purpose of the plan creation phase is to transform the ordered modification series $\overline{mod^n}$ coming from the scheduling phase into a sequence of ordered `READ` and `WRITE` operations, where each operation at maximum reads or writes $r \in \mathbb{N}$ bytes. $r > 0$ is a user-defined input parameter called the **maximum read-write block size**. Note that inserted sequences $I_k^{l_k}$ lead to `WRITE` operations only, which also adhere to the maximum size r.

The plan creation phase is the heart of the algorithm, as it:

- Ensures a correct output sequence T will be created in the execution phase, by computing a correct order of `READ`s and `WRITE`s
- Ensures that no `READ` or `WRITE` operation is bigger than r
- Ensures no unnecessary `READ` and `WRITE` operations happen

As starting point, a key idea of the algorithm already indicated in earlier sections is to partition the input sequence S into disjoint parts at the modification offsets, see figure 3.

The sub-sequence

$$S_k := S[e_k, s_{k+1})$$

is called the k-th **source region**. It represents the bytes between two modifications or between the last modification and the end of S at $s_n := \text{len}(S)$. Each source region might be subject to be moved to another location by `READ` and `WRITE` operations. First of all, it is clear that $S[0, s_0)$, i.e. the bytes before the first modification, never need to be moved. So the execution plan will not contain `READ` and

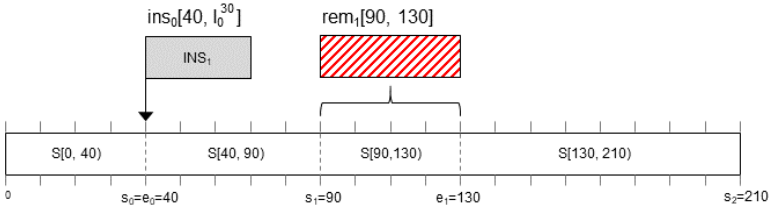

Figure 3: Partitioning of the input sequence S by modification offsets s_k and e_k

`WRITE` operations for this region. But which of the source regions need to be moved? To calculate this for the k-th source region, we introduce a variable δ_{k+1} associated with the k-th modification. We define $\delta_0 := 0$. δ_{k+1} is the number of bytes to move the k-th source region, summarizing all shifts of modifications $m_0, \ldots, m_k$. We can have the following cases:

1. $\delta_{k+1} > 0$: The k-th source region needs to be moved towards higher offsets by δ_{k+1}
2. $\delta_{k+1} < 0$: The k-th source region needs to be moved towards smaller offsets by $-\delta_{k+1}$
3. $\delta_{k+1} = 0$: The k-th source region does not need to be moved at all, so the algorithm must not create `READ` and `WRITE` operations for this source region.

The third case e.g. occurs if there is a `REMOVE` of 30 bytes at offset 50, and an `INSERT` of 30 bytes at offset 100. All bytes behind offset 100 do not need to be moved.

Having determined δ_{k+1}, we now need to define the order of moving the source regions. As we saw in an example in section 1, potentially multiple correct orders exist. The main requirement is that `WRITE` operations do not overwrite any other source region bytes before they have been moved. Thus, we introduce a new term, the

k-th **target region** is defined as

$$T_k := T[s_k + \delta_k, s_{k+1} + \delta_{k+1}),$$

where $T_k = I_k^{l_k} + S_k$.

The second key idea of the algorithm addresses the execution order issue. The idea is to sort modifications such that a modification m_k whose target region overlaps the source region of another modifcation m_j is executed after m_j. If m_k would be executed first, it would overwrite the bytes in the source region of m_j, thus leading to data loss. Put formatlly: m_j is executed before m_k for $k \neq j$ in the following cases:

1. If $S_j \cap T_k \neq ()$, m_j is considered smaller than m_k.
2. Otherwise if $S_j \cap T_k = ()$, m_j is considered smaller than m_k if $j < k$.

This defines a total ordering on the modification series $\overline{mod^n}$, and we call the resulting modification series $\underline{mod^n}$.

The last part of the plan creation phase is the creation of appropriately chunked `READ` and `WRITE` operations to execute the modifications. The chunk-wise reading and writing includes an additional aspect to avoid data loss: Chunk-wise reading and writing of the source region must be done in the correct order to prevent overwriting data that still needs to be preserved. We illustrate this in figure 4.

It would be wrong for a single `INSERT` to first read chunk c_0 and then write it at $s_0 + \delta_1 = 70$, because this would overwrite the first 30 bytes of chunk c_1, i.e. data loss. Instead, it requires backwards read-write: First read chunk c_3 to write it at 220, then chunk c_2 and so on. Similarly, it would be wrong for a single `REMOVE` to read backwards, so `REMOVE`s require forward reading starting with chunk c_0.

In more general terms, the read and write order of chunks of the source region depends on the value of δ_{k+1}:

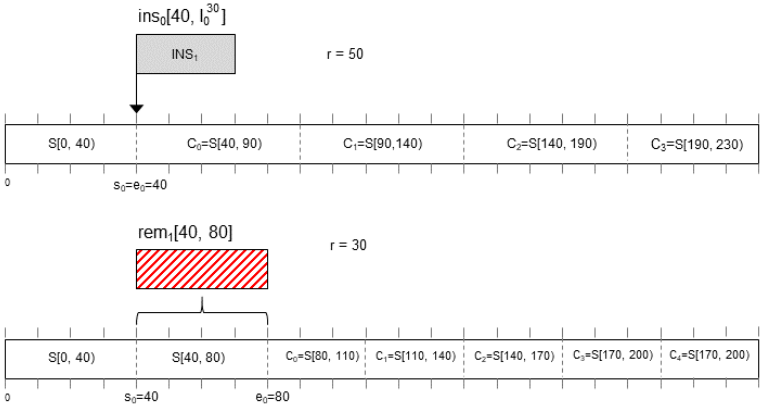

Figure 4: Examples for chunking after single removes and inserts

- If $\delta_{k+1} = 0$, no reading and writing of chunks is required
- If $\delta_{k+1} > 0$, reading must be done backwards, starting at the highest chunk index
- If $\delta_{k+1} < 0$, reading must be done forwards, starting at chunk index 0

The number of chunks to read and write for source region $S_k = [e_k, s_{k+1})$ of $\underline{mod^n}$ is

$$n_k := \left\lfloor \frac{e_k - s_{k+1}}{r} \right\rfloor + \begin{cases} 0 & \text{if } (e_k - s_{k+1}) \mod r = 0 \\ 1 & \text{if } (e_k - s_{k+1}) \mod r \neq 0 \end{cases}$$

Likewise, the number of chunks to write for the insertion bytes $I_k^{l_k}$ is:

$$w_k := \left\lfloor \frac{l_k}{r} \right\rfloor + \begin{cases} 0 & \text{if } l_k \mod r = 0 \\ 1 & \text{if } l_k \mod r \neq 0 \end{cases}$$

The result of the plan creation phase is a sequence $\mathcal{E}(r)$ of `READ` and `WRITE` operations with the following structure:

- It starts with w_0 `WRITE` operations at the corresponding chunk start offsets $s_k + \delta_k + x_i$, where x_i is the start offset of the i-th chunk, $0 \leq i \leq w_0 - 1$.
- Then n_k pairs of `READ` and `WRITE` operations follow at read offset $e_k + y_i$ and write offset $e_k + \delta_{k+1} + y_i$, where y_i is the start offset of the i-th chunk, $0 \leq i \leq n_0 - 1$.
- After this, the same for the second until the n-th modification in $\underline{mod^n}$ follows

$\mathcal{E}(r)$ is called the **execution plan** of $\underline{mod^n}$ with read-write block size r.

3.4 Execution Phase

The execution phase simply iterates the execution plan in order and executes the operations in an application dependent way. E.g., for files, corresponding operating system APIs offer primitive *read* and *write* calls. In addition, for files, a sindle *truncate* operation to shorten the file might be necessary.

3.5 Algorithm Summary

After detailed phase explanations, we want to give a concise summary of the algorithm here.

- **Input:**
  - Finite byte sequence $S \in \mathcal{S}$
  - Unordered modification series $mod^n \in M^n$:

$$mod^n := \left(m_0[s_0, e_0, I_0^{l_0}], \ldots, m_{n-1}[s_{n-1}, e_{n-1}, I_{n-1}^{l_{n-1}}]\right)$$

  - Read-write block-size $r \in \mathbb{N}$
- **Phase 1:** Scheduling Phase
  - **1a.** Validate the modification series mod^n by checking that the following conditions are met for all modifications $m_k, 0 \leq k \leq n-1$:
    - (i.) $0 \leq s_k, e_k \leq \text{len}(S)$
    - (ii.) $s_k, e_k \notin (s_j, e_j) \quad \text{for } j \neq k$
  - **1b.** If one of the conditions is not met for at least one m_k, set $T := S$ and terminate the algorithm.
  - **1c.** Otherwise sort mod^n, following the criteria:
    - (i.) If $s_k > s_j$ for $j \neq k$, m_k is executed before m_j
    - (ii.) If $s_k = s_j$ for $j \neq k$, m_k is executed before m_j if $k < j$
  - **1d.** The regular output of phase 1 is a validated modification series $\overline{mod^n}$ sorted by offset ascending.
- **Phase 2:** Plan Creation Phase
  - **2a.** Set $\delta_0 := 0$.
  - **2b.** For each modification $m_k[s_k, e_k, I_k^{l_k}] \in \overline{mod^n}$:
    - (i.) Set $\delta_{k+1} := \delta_k + l_k + (e_k - s_k)$
    - (ii.) Set $S_k := S[e_k, s_{k+1})$ with $s_n := \text{len}(S)$, S_k is the k-th source region.
    - (iii.) Set $T_k := T[s_k + \delta_k, s_{k+1} + \delta_{k+1})$ with $s_n := \text{len}(S)$, T_k is the k-th target region.
  - **2c.** Sort $\overline{mod^n}$, following the criteria:
    - (i.) If $S_j \cap T_k \neq ()$, m_j is considered smaller than m_k.
    - (ii.) If $S_j \cap T_k = ()$, m_j is considered smaller than m_k if $j < k$.
  - **2d.** The result is a validated modification series $\underline{mod^n}$ sorted by non-overlapping target and source regions.

2e. For each modification $m_k[s_k, e_k, I_k^{l_k}] \in \underline{mod^n}$:

(i.) Set

$$n_k := \left\lfloor \frac{e_k - s_{k+1}}{r} \right\rfloor + \begin{cases} 0 & \text{if } (e_k - s_{k+1}) \bmod r = 0 \\ 1 & \text{if } (e_k - s_{k+1}) \bmod r \neq 0 \end{cases}$$

with $s_n := \text{len}(S)$. n_k is the number of chunked read and write operations necessary to move the bytes between the current and next modification.

(ii.) If $\delta_{k+1} < 0$, forward reading and writing is necessary. Generate n_k pairs of `READ`s and `WRITE`s, start with read offset $ro_{k,0} := e_k$ and write offset $wo_{k,0} := e_k + \delta_{k+1}$. Increment both offsets by r to generate the next `READ` at $ro_{k,1}$ and $wo_{k,1}$, and so on, until all pairs are generated. For each chunk, exactly r bytes are read or written, except for the last chunk, if $t := (e_k - s_{k+1}) \bmod r$ is smaller than r, only t bytes are read and written.

(iii.) Otherwise if $\delta_{k+1} > 0$, backward reading and writing is necessary. Generate n_k pairs of `READ`s and `WRITE`s, start with read offset $ro_{k,0} := s_{k+1} - \max\{r, (e_k - s_{k+1}) \bmod r\}$ and write offset $wo_{k,0} := s_{k+1} - \max\{r, (e_k - s_{k+1}) \bmod r\} + \delta_{k+1}$. Decrement both offsets by r to generate the next `READ` at $ro_{k,1}$ and $wo_{k,1}$, and so on, until all pairs are generated. For each chunk, exactly r bytes are read or written, except for the first chunk, if $t := (e_k - s_{k+1}) \bmod r$ is smaller than r, only t bytes are read and written.

(iv.) Set

$$w_k := \left\lfloor \frac{l_k}{r} \right\rfloor + \begin{cases} 0 & \text{if } l_k \bmod r = 0 \\ 1 & \text{if } l_k \bmod r \neq 0 \end{cases},$$

w_k is the number of chunked write operations necessary to write the insert byte sequence $I_k^{l_k}$.

(v.) If $w_k > 0$, generate w_k `WRITE`s for the insertion bytes, starting at offset $wi_{k,0} := s_k + \delta_k$. Increment the offset by r to generate the next `WRITE` at $wi_{k,1}$, and so on, until all pairs are generated. For each chunk, exactly r bytes are written, except for the last chunk, if $t := l_k \bmod r$ is smaller than r, only t bytes are written.

2f. The regular output of phase 2 is a sequence of `READ` and `WRITE` operations $\mathcal{E}(r)$ in correct order, the execution plan.

- **Phase 3:** Execution Phase

  3a. Iterate all operations in $\mathcal{E}(r)$ in order and execute the corresponding operations (implementation-specific) to transform the byte sequence S into T.

- **Output:**

  – Finite byte sequence $T \in \mathcal{S}$ with

$$T := S[0, s_0) + \sum_{k=0}^{n-1} \left(I_k^{l_k} + S[e_k, s_{k+1}) \right), \text{ where } s_n := \text{len}(S).$$

3.6 Properties of the Algorithm

3.6.1 Termination

The algorithm terminates in any case, as all phases are limited by the fixed number n of modifications and the finite size of S.

3.6.2 Correctness

There are no undefined operations, as $r > 0$.

Phase 1 requires no proof, as it simply validates and sorts the modification series according to offset ascending. Likewise, phase 3 requires no proof.

For the algorithm to be fully correct, the following needs to be prooved for phase 2: Given a finite byte sequence $S \in \mathcal{S}$, a maximum read-write block-size $r \in \mathbb{N}$ and a modification series

$$\overline{mod^n} := \left(m_0[s_0, e_0, I_0^{l_0}], \ldots, m_{n-1}[s_{n-1}, e_{n-1}, I_{n-1}^{l_{n-1}}]\right),$$

the algorithm creates a target byte sequence T such that

$$T := S[0, s_0) + \sum_{k=0}^{n-1} \left(I_k^{l_k} + S[e_k, s_{k+1})\right), \text{ where } s_n := \text{len}(S).$$

Proof. If $S = ()$, only `INSERT` operations are permitted according to criterion (i.) in step 1a. Assuming n `INSERT`s of the sub-sequences $I_k^{l_k}$ at offset $s_k = 0$, we can see that T has the correct structure.

Let $S \neq ()$. To avoid confusion, in the following sections, we use the notation δ'_{k+1} to indicate the δ value associated with the k-th modification in the sorted modification series $\underline{mod^n}$, i.e. the modification series already sorted by step 2c. In general, this is a different value than the δ_{k+1} in step 2b, because the order of modifications might have changed due to step 2c.

T always starts with $S[0, s_0)$, because:

- If m_0 contains non-empty insertion bytes, the `WRITE` operation with smallest offset according to step 2f is done at offset s_0, thus not changing any bytes in $S[0, s_0)$.
- If m_0 does not contain insertion bytes, the `WRITE` operation with smallest offset according to step 2f is done at offset $e_0 + \delta'_1 = e_0 + s_0 - e_0 = s_0$, thus not changing any bytes in $S[0, s_0)$.

Step 2f guarantees to first write the insertion bytes $I_k^{l_k}$ (if non-empty) at offset $s_k + \delta'_k$ and then the follow-up bytes at $e_k + \delta'_{k+1}$, where $\delta'_{k+1} = \delta'_k + l_k - (e_k - s_k)$, because:

$$e_k + \delta'_{k+1} - s_k - \delta'_k = e_k + \delta'_k + l_k - e_k + s_k - s_k - \delta'_k = l_k.$$

Thus the algorithm is correct for $n = 1$, preserving the order of insertion bytes and follow-up bytes and ensuring that insertion bytes cannot overwrite follow-up bytes.

Let $n > 1$ be arbitrary. Step 1a ensures that source regions of different modifications cannot overlap each other. Likewise, target regions of different modifications created in step 2b (iii.) do not overlap each other, or more precisely:

$$T = S[0, s_0) + \sum_{k=0}^{n-1} T_k,$$

i.e. T_k is nothing else but $T_k := I_k^{l_k} + S[e_k, s_{k+1})$. Let $T_k := T[s_k + \delta_k, s_{k+1} + \delta_{k+1})$, then $T_k \cap T_{k+1} = ()$, as we can directly see that the end offset $s_{k+1} + \delta_{k+1}$ of T_k is the start offset of T_{k+1}.

It remains to be prooved that the bytes written by different modifications do not overwrite each other. For the first modification m_0, step 2c guarantees that its target region does not overlap the source region of any of the modifications $m_j, j > 0$. In general, for modification m_k it is guaranteed that its target region does not overlap any of the source regions of the follow-up modifications m_j with $j > k$. Thus there cannot be any overwrites.

This prooves that the algorithm is correct and yields the required result. □

3.6.3 Complexity

The validation of n modifications requires to check every modification against every other modification for validity. So in worst case, phase 1 can be assigned complexity of $O(n^2)$. The sorting in steps 1c and 2c is also done in worst case with $O(n^2)$. Assuming a maximum read-write block size of $r = 1$, n inserts as well as $\delta_{k+1} \neq 0$ for every $0 \leq k \leq n - 1$, the chunking generates

$\sum_{k=0}^{n-1} (l_k + (s_{k+1} - e_k))$ chunks. A rough upper bound is

$$\sum_{k=0}^{n-1} (l_k + (s_{k+1} - e_k)) \leq n \cdot \text{len}(S).$$

Thus, step 2e can also be said to have quadratic complexity. To summarize: In worst-case, we can assign an overall complexity of

$$\begin{cases} O(n \cdot \text{len}(S)) & \text{if } \text{len}(S) > n \\ O(n^2) & \text{otherwise} \end{cases}$$

to the algorithm.

4 Implementation Hints

The algorithm can be modified to also allow overlapping `INSERT`s, `REMOVE`s or `REPLACE`s, if appropriate for corresponding applications.

For implementation purposes, it should be considered to implement phase 1 to check validity of an added modification on the fly when adding it:

- Use a sorted data structure that corresponds to the sorting criteria defined in section 3.2.
- Check validity by checking region overlaps with earlier and later modifications in the sorted data structure

Further research is possible to find a better algorithm in terms of worst-case complexity. In a follow-up paper, the overall number of different correct execution plans can be determined, and it could be investigated if there is an optimum execution plan according to several possible criteria, e.g. cache usage or consecutiveness of disk positions.