Secure Merge in Linear Time and $O(\log \log N)$ Rounds

Mark Blunk, Paul Bunn, Samuel Dittmer, Steve Lu, Rafail Ostrovsky

Abstract

Secure merge considers the problem of combining two sorted lists (which are either held separately by two parties, or held by two parties in some privacy-preserving manner, e.g. via secret-sharing), and outputting a single merged (sorted) list in a privacy-preserving manner (typically the final list is encrypted or secret-shared amongst the original two parties). Just as algorithms for insecure merge are faster than comparison-based sorting ($\Theta(n)$ versus $\Theta(n \log n)$ for lists of size $n$), we explore protocols for performing a secure merge that are more performant than simply invoking a secure sort protocol. Namely, we construct a semi-honest protocol that requires $O(n)$ communication and computation and $O(\log \log n)$ rounds of communication. This matches the metrics of the insecure merge for communication and computation, although it does not match the $O(1)$ round-complexity of insecure merge. Our protocol relies only on black-box use of basic secure primitives, like secure comparison and shuffle.

Our protocol improves on previous work of [FNO22], which gave a $O(n)$ communication and $O(n)$ round complexity protocol, and other “naive” approaches, such as the shuffle-sort paradigm, which has $O(n \log n)$ communication and $O(\log n)$ round complexity. It is also more efficient for most practical applications than either a garbled circuit or fully-homomorphic encryption (FHE) approach, which each require $O(n \log n)$ communication or computation and have $O(1)$ round complexity.

There are several applications that stand to benefit from our result, including secure sort (in cases where two or more parties have access to their own list of data, secure sort reduces to secure merge since the parties can first sort their own data locally), which in-turn has implications for more efficient private set intersection (PSI) protocols; as well as secure mutable database storage and search, whereby secure merge can be used to insert new rows into an existing database.

In building our secure merge protocol, we develop several subprotocols that may be of independent interest. For example, we develop a protocol for secure asymmetric merge (where one list is much larger than the other), which matches theoretic lower-bounds for all three metrics (assuming the ratio of list sizes is small enough).

Keywords. Secure Merge, Secure Sort, Secure Database, Private Set Intersection.

---

This work was performed under the following financial assistance award number 70NANB21H064 from U.S. Department of Commerce, National Institute of Standards and Technology. The statements, findings, conclusions, and recommendations are those of the author(s) and do not necessarily reflect the views of the National Institute of Standards and Technology or the U.S. Department of Commerce.

Submitted to a conference in April 2022
1 Introduction

It is well-known that sorting a list of length $n$ using only comparisons requires $\Theta(n \log n)$ time, while merging two already sorted lists requires only $\Theta(n)$ time, see e.g., [MSS08, Ski98]. Namely, by taking advantage of the structure of the input (the ordering on the two lists), merge algorithms can out-perform their “input-agnostic” counterparts (sorting algorithms) by a factor of $\log n$ in run-time. This reflects and reinforces a general intuition that it is more efficient to maintain and update an existing structure than it is to tear down the structure and rebuild it from the ground up. In the area of database management, this general intuition (applied here to the specific example of merging versus sorting) applies directly to tables that need to be kept in a sorted order (e.g., to efficiently query order statistics). It is far more efficient to maintain a table in a sorted order by periodically merging in new data entries than it is to re-sort the entire table for each update cycle. Another frequently encountered scenario when a sorted list is desired occurs when data values are held by two or more parties who wish to produce a sorted, merged copy of their data. Each party can separately invoke an (insecure) sort algorithm locally on their own data, so that (the more costly) secure operations are required only for merging the lists. For example, many private set intersection (PSI) protocols (see e.g., [CHI+19, PSSZ15, HEK12]) are bottlenecked by an underlying invocation of secure sort, and consequently (in appropriate settings) could benefit by a factor of $\log n$ in performance by using secure merge instead.

Motivated by the above intuition, we investigate in this paper whether the theorized $\log n$ runtime improvement for merge versus sort can be realized in the secure setting as well. We answer this question in the affirmative, producing a protocol that instantiates the following:

**Theorem 1.1 (Informal).** There exists a two-party secure merge protocol with $\Theta(n)$ run-time (computation and communication) and $\Theta(\log \log n)$ rounds that relies only on black-box access to the secure functionalities defined in §3.7: comparison, shuffle, open, and conditional-addition.

1.1 Paper Structure

In Section 2 we summarize previous work targeting the secure sort/merge problem, and provide a comparison table of our main result with relevant earlier works in Table 1. In Section 3, we set notation and provide a high-level overview of the techniques and tools our protocols build on. Then in Section 4, we present our main secure merge protocol, as well as the sub-protocols that it builds upon. We give formal statements and proofs in Section 5.1, and describe our constant round secure merge in the asymmetric case where one list has size $n^\alpha$, for fixed $\alpha < 1$, in Section 6.

Supplementary material (instantiations of secure primitives, full proofs, and expanded discussion) can be found in Appendices A-D.

2 Previous Work

As the merging/sorting of lists is a fundamental problem in computer science, there has been enormous research in this area. Not surprisingly, many of the protocols for securely merging/sorting lists draw inspiration from these insecure counterparts (not to mention the generic approach of converting insecure protocols to secure protocols e.g., via garbled circuit or fully-homomorphic encryption). We discuss below\(^1\) previous work for both the insecure and secure variants, and for the sake of comparison summarize the most relevant results to our main result in Table 1 below.

---

\(^1\)As mentioned, the work in this area is extensive, so our summary of results is necessarily non-exhaustive. Please see cited works, and references therein, for a more complete overview and discussion.
2.1 Insecure Merge Algorithms

2.1.1 Sorting Networks

The relationship between secure merging and secure sorting can be traced back to [Bat68], which built a sorting network of size $O(n \log^2 n)$ from $\log n$ merging networks, each of size $O(n \log n)$. Although an asymptotically optimal sorting network of size $O(n \log n)$ was later achieved by Ajtai, Komlos, and Szemeredi [AKSS83], merging networks cannot match the combinatorially optimal $O(n)$ size. A merging network on lists of size $\Theta(n)$ require $\Omega(n \log n)$ comparators, as shown by Yao and Yao [YY76], and depth of $\Omega(\log n)$, as shown by Hong and Sedgewick [HS82].

2.1.2 RAM and PRAM

The classical merge algorithm requires $O(n)$ work on a RAM machine, see e.g., [MSS08]. As discussed below, the approach of [FNO22] uses an approach that is inspired by this classical merge algorithm, and achieves matching asymptotic work (albeit with $O(n)$ round-complexity).

A parallel RAM machine (PRAM) allows multiple processors to act on the same set of memory in parallel. We are in particular interested in Concurrent-Read-Exclusive-Write (CREW) PRAM, where all processors can read the same memory simultaneously, but processors cannot write to the same memory address at the same time, see e.g., [BH85] for more formal definitions and a discussion of various PRAM models.

For PRAM machines, Valiant showed in [Val75] that $O(n)$ processors could merge two lists of size $O(n)$ in time $O(\log \log n)$. This was improved to $O(n / \log \log n)$ processors by Borodin and Hopcroft [BH85], who also showed that the $O(\log \log n)$ time bound is optimal when limited to $O(n)$ processors. As a rough heuristic, we expect the number of processors and total work done by a PRAM algorithm to serve as a lower bound for the round complexity and communication complexity of a corresponding secure protocol, motivating the following:

**Conjecture.** Any linear-time protocol for two parties to securely merge their lists (each of size $\Theta(n)$) requires $\Theta(\log \log n)$ rounds of communication.

Notice that if the above conjecture is valid, then our secure merge protocol (Section 4) is asymptotically optimal in all three metrics: computation, communication, and round complexity.

More formally, if a 2-party protocol $\Pi$ securely realizes some functionality $F$ in $R$ rounds and $C$ communication, and each party can execute each round of the protocol in $O(1)$ time on a CREW PRAM with $C$ processors, then there exists an algorithm $A$ that realizes $F$ on a single CREW PRAM machine (playing the role of all parties) with $C$ processors, $O(RC)$ total work, and $O(R)$ time. This observation immediately implies that the conjecture is true for secure merge protocols with the property that each round of the protocol can be executed in $O(1)$ time on a CREW PRAM with $O(n)$ processors.

Both the Valiant and the Borodin-Hopcroft algorithms rely on the following basic construction: Split each list into blocks of size $\sqrt{n}$, with $\sqrt{n}$ pivots. Running all pairwise comparisons between pivots identifies which block of the opposite list each pivot is mapped to; then another round of pairwise comparisons identifies the exact position the pivot is mapped to within that block. This partitions the merge problem into $\sqrt{n}$ subproblems of size approximately $\sqrt{n}$, giving the recurrence $c(n) = \sqrt{n} \cdot c(\sqrt{n})$, if $c(n)$ is the cost of merging two lists of length $n$. With sufficiently many processors, this recurrence gives a run-time of $O(\log \log n)$. Our secure merge protocols (Section 4) are similar in spirit, although some adaptation is required to deal with blocks from one list that cover more than one block of the other list.

A follow-up work by Hayashi, Nakano, and Olariu [HNO98] demonstrates a protocol with $O(\log \log n + \log k)$ run-time for merging $k$ lists of size $O(n)$. Following the intuition that run-time in the PRAM model roughly corresponds to (a lower-bound on) round-complexity in the secure setting,
this suggests that a linear-time secure merge protocol for \( k \) parties could have round complexity \( O(\log k) \). For example, this would imply that an optimal secure merge protocol for \( k = O(\log n) \) parties (each with a list of size \( O(n) \)) could have linear run-time and \( O(\log \log n) \) round complexity. However, because our secure merge protocol (Section 4) relies on black-box use of a constant-round secure shuffle subprotocol with \( O(n) \) communication, and since currently known instantiations of secure shuffle with linear communication have linear or worse dependence on the number of parties (in terms of round-complexity) [FO21, LWZ11], direct extension of our secure merge protocol to \( k > 2 \) parties would have \( O(\log \log n + k) \) round complexity.

2.2 Secure Merge Algorithms

2.2.1 Security via Generic Transformation

We explore here two naïve solutions for transforming an insecure merge algorithm to a secure one (see Table 1 for a succinct comparison of our secure merge protocol to these naïve solutions):

**Garbled Circuits.** By choosing any (insecure) sorting algorithm that can be represented as a circuit, the parties can use garbled circuits to (securely) sort their list in \( O(1) \) rounds. As discussed above in §2.1.1, for comparison-based merging networks, \( \Omega(n \log n) \) comparisons and \( \Omega(\log n) \) depth are necessary, and achievable by the Batcher merging network [Bat68].

Additionally, we note that obtaining \( \kappa \) bits of computational security when merging lists whose elements have size \( W \) bits requires \( \kappa \cdot W \) bits, or \( \kappa \) words of communication for each comparison. Thus, obtaining secure merge via a garbled circuit approach (for a circuit representing a merging network) would result in a constant-round protocol with \( \Omega(\kappa \cdot n \log n) \) communication.

On the other hand, using GMW-style circuit evaluation [GMW87] instead of garbled circuits can reduce the overhead of each comparison (from \( \kappa \) down to constant), but incurs a hit in round complexity (proportional to the depth of the circuit instead of constant-round).

**Fully Homomorphic Encryption.** We can also construct a \( O(1) \) round protocol by having one party encrypt their inputs under fully homomorphic encryption (FHE) and send the result to the other party, who performs the desired calculations on ciphertexts. The other party then subtracts a vector of random values \( r \) from the (encrypted) merged list and sends the result back to the first party, who decrypts to obtain the merged list shifted by \( r \). Now the parties hold an additive sharing of the sorted list.

As with garbled circuits, however, the calculations the second party performs on the ciphertexts must be input-independent (in order to avoid information leakage), and so the calculation must be represented by a circuit. This means that communication is (asymptotically) lower than the garbled circuit approach, since the first party need only provide ciphertexts of his list (which correspond to inputs to the circuit), as opposed to providing information for each gate of the circuit. Therefore the communication required for the FHE approach is only \( O(\beta^* n) \), where \( \beta^* \) is the ciphertext expansion for the FHE scheme, which approaches 1 as the word size \( W \) grows. However, while communication in the FHE approach may be (asymptotically) reduced (compared to the garbled circuit approach), notice that the computation is still \( O(\gamma^* n \log n) \), where \( \gamma^* \) is the cost of a multiplication under FHE, as the circuit requires \( O(n \log n) \) comparison (multiplication) operations. Additionally, since the circuit has depth \( \Omega(\log n) \), the FHE scheme will require bootstrapping to avoid ciphertext blowup, which will be expensive in practice.

2.2.2 Shuffle-Sort Paradigm

One challenge facing any secure merge (or sort) protocol is that the results of each comparison must be kept secret from each party, or else security is lost. One approach to allowing the results of the comparisons to be known without information leakage is to first shuffle (in an oblivious manner) the
input lists. However, since this shuffling inherently destroys the property that the two input lists are already sorted, this approach reduces a secure merge problem to a secure sort problem, hence incurring the log \(n\) efficiency loss.

Comparison-based sorting requires \(O(n \log n)\) communication and computation as well as \(O(\log n)\) rounds, while secure shuffling requires \(O(n)\) communication and \(O(1)\) rounds, see e.g., [HKI\(^+\)12, FO21]. Therefore a secure sort using the shuffle-sort paradigm requires \(O(n \log n)\) communication and computation and \(O(\log n)\) rounds, which is worse than our secure merge approach on all three metrics; see Table 1.

2.2.3 Oblivious Sort

Because the AKS-constant is too large for practical applications, a number of other approaches to secure sorting have been explored. The shuffle-sort paradigm mentioned above is one example of a large family of oblivious sort protocols, which allow for a variable memory access pattern as long as it is independent of the underlying list values, or data oblivious. We mention here the radix sort of Hamada, Ikarashi, Chida, and Takahashi [HICT14] achieves \(O((W \log W + W)n + n \log n)\) communication (in memory words) and \(O(1)\) round complexity in the 3-party honest majority setting with constant bit lengths of elements, later improved by Chida, Hamada, Ikarashi, Kikuchi, Kiribuchi, and Pinkas [CHI\(^+\)19], to \(O(n \log n)\) memory words. However, the round complexity depends linearly on \(W\), so when \(W \approx \log n\), this matches the round complexity of the other protocols.

2.2.4 Secure Merge Protocols

There are several works that investigate secure merge directly. The first, due to Falk and Ostrovsky [FO21], achieves \(O(n \log \log n)\) communication complexity with \(O(\log n)\) round complexity. The second, due to Falk, Nema, Ostrovsky [FNO22], achieves the asymptotically optimal \(O(n)\) communication complexity (and with small constants), but requires \(O(n)\) rounds of communication. For many cryptographic applications, a high round complexity causes more of a bottleneck than a high communication complexity. Therefore, the secure merge protocol of [FNO22], while both simple and asymptotically optimal in terms of communication, may still not be practical for many applications.

A comparison of these results to our secure merge protocol can be found in Table 1 below.

2.3 Comparison of Results

The black box functionalities introduced in our main theorem can be efficiently implemented using additive secret sharing, GMW, and an additively homomorphic rerandomizable public key cryptosystem such as Paillier or lattice-based LWE, as discussed in §3.7.

To make the comparison with our protocols as concrete as possible, we introduce the following constants: \(\kappa\) is a computational security parameter for use in Yao’s garbled circuits. \(\beta\) is the ciphertext expansion in our chosen additively homomorphic cryptosystem, while \(\gamma\) is the cost of decryption in that cryptosystem.

Asymptotically \(\kappa \gg \log n\) is necessary so that \(\kappa\) cannot be guessed in the time it takes to traverse the list. In practice \(\kappa = 128\) is standard. Asymptotically, \(\beta\) approaches 1 as the word size \(W\) grows. In practice for LWE cryptosystems \(\beta\) can be quite reasonable, e.g. \(\beta = 17.7\) for shuffling 16384 29-bit plaintexts packed into a single ciphertext in the PALISADE FHE library [FO21].

Because running secure merge as a comparison-based circuit requires \(\Omega(\log n)\) depth, the ciphertext expansion and computational complexity of performing the sorting over FHE will take a hit compared to the depth one public key operations required for 2-party sort. Rather than attempting to estimate this gap, we use \((\beta^*, \gamma^*)\) to represent the same quantities over some FHE scheme.

We assume the objects to be sorted are contained in \(O(1)\) memory words of size \(W\) bits. Unless explicitly stated otherwise, our communication and computational complexity numbers are given in
terms of memory words and primitive operations on memory words.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Computation</th>
<th>Communication</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Garbled) Merging Network [Folklore]</td>
<td>$O(\kappa \cdot n \log n)$</td>
<td>$O(\kappa \cdot n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>(GMW) Merging Network [Folklore]</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>(FHE) Merging Network [Folklore]</td>
<td>$O(\gamma^* n \log n)$</td>
<td>$O(\beta^* n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shuffle-Sort [HKF+12]</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Oblivious Radix Sort [CHI+19]</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(W)$</td>
</tr>
<tr>
<td>Secure Merge of [FO21]</td>
<td>$O(n \log n + \gamma n)$</td>
<td>$O(n \log n + \beta n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Secure Merge of [FNO22]</td>
<td>$O(\gamma n)$</td>
<td>$O(\beta n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Our Secure (n$^\alpha$,n) Merge Protocol</td>
<td>$O(2^{1/(1-\alpha)^3} \gamma n)$</td>
<td>$O(2^{1/(1-\alpha)^3} \beta n)$</td>
<td>$O(1/(1-\alpha)^3)$</td>
</tr>
<tr>
<td>Our Secure Merge Protocol</td>
<td>$O(\gamma n)$</td>
<td>$O(\beta n)$</td>
<td>$O(\log \log n)$</td>
</tr>
</tbody>
</table>

Table 1: Comparison of our secure merge protocol with other existing solutions, where $n$ is the list size, $W$ is the size of a memory word, list elements fit into $O(1)$ memory words, $\kappa$ is a computational security parameter for Yao’s garbled circuits, and $\beta$ and $\gamma$ are the ciphertext expansion and computational cost of a decryption operation, respectively, over some additively homomorphic cryptosystem, while $(\beta^*, \gamma^*)$ represent the same costs over FHE. In our $(n^\alpha,n)$ merge protocol, $\alpha < 1$ is some fixed constant.

3 Overview of Techniques

3.1 Notation

For two lists $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$, we denote the zip of $A$ and $B$ as $A \boxtimes B = ((a_1, b_1), \ldots, (a_k, b_k))$. For any two sorted lists $L_1$ and $L_2$, let $\bigcup$ denote the “merge” of two lists (i.e. $\bigcup$ is functionally equivalent to (multi-)set union followed by sort): $L_1 \bigcup L_2 = \text{Sort}(L_1 \cup L_2)$. For any sorted list $L_j$ of size $n$, and for any $k|n$, let $M_{j,k}$ denote the $k$ “medians” of $L_j$. Namely, if list $L_j = \{u_1, \ldots, u_n\}$, then:

$$M_{j,k} := \{u_{j \cdot \frac{\beta}{k}}\}_{j=1}^{k}$$

Basic properties that follow from the definition of medians can be found in Appendix D.

3.2 Symmetric vs. Asymmetric Merge

Throughout the paper we distinguish between secure symmetric merges $\Pi_{\text{SSM}}$ and secure asymmetric merges $\Pi_{\text{SAM}}$. In a symmetric merge, the lists are of roughly the same size, or differ by at most a constant factor (since then one list can be padded with dummies to match the length of the other). In an asymmetric merge, the ratio of list sizes is larger than constant.

Both flavors of merge, symmetric and asymmetric can be produced using the other flavor of merge as a subprotocol, and we will see in our constructions. In particular, the use of asymmetric merge as a subprotocol is a valuable tool for separating the lists into blocks and aligning blocks.

3.3 Aligning blocks

As in the PRAM merge algorithm discussed in §2.1.2, our secure merge protocols work by partitioning one list based on the values of the $k$-medians of the other list, which gives a collection of $k$ subproblems, for some choice of $k$. 

5
However, following the PRAM merge algorithm exactly would leak the size of these subproblems and in turn would leak the number of elements of $L_2$ lying between successive $k$-medians of $L_1$. Instead, we need tools for obliviously aligning blocks. Our protocols rely on two primary strategies.

First, our lossy strategy works by using an asymmetric merge on medians to count the number of elements from one list that would be assigned to each block of the other list. When the number of elements is within a constant factor of the block size, these elements can be matched with the corresponding block, creating subproblems for which secure symmetric merge can be applied. For blocks which have too many elements matched to them, the elements are extracted into an overflow list, which is treated separately. Our approach guarantees that the overflow list is sufficiently small, and so we apply a less efficient merge protocol to the entirety of the overflow list.

Second, our lossless strategy begins by producing, from two lists of size $n$, two expanded lists of size $2n$, with $n$ dummy elements inserted into each list. These expanded lists are constructed so that they are “aligned”: we can partition both lists into blocks of size $n/k$, run secure merge separately on these blocks, and simply concatenate the outputs (and remove dummy elements) to obtain the final merged list. More concretely, the dummy elements inserted into each list represent duplicates of the $k$ medians from the opposite list, inserted into the appropriate position. We describe this insertion procedure further and show that it guarantees aligned blocks in Appendix D.

### 3.4 The Tag-Shuffle-Reveal Paradigm

We make repeated use of the tag-shuffle-reveal paradigm, which should be considered analogous to the shuffle-sort paradigm of prior sorting protocols, see e.g. [HKI+12], or an extension of the shuffle-reveal paradigm of [HICT14]. Our tag-shuffle-reveal paradigm is used implicitly in [FO21], but we expand its use here, and so describe it in more detail.

In the tag-shuffle-reveal paradigm, each element of a list is (obliviously) tagged with some label. This label can be (a secret sharing of) its current index, or it can be the result of some multiparty computation, for example a bit representing the output of a comparison against another value. Then, after shuffling the list, the tag or some part of the tag is opened, and the list entries are rearranged accordingly.

Because the shuffle step ensures that the tags are randomly ordered, The only requirement to ensure security is to ensure that the set of values the opened tags take on do not depend on the underlying data. The main application of this tag-shuffle-reveal paradigm is our extraction protocols, described below.

### 3.5 Extraction Protocols

Our construction relies on a collection of tools for extracting marked elements. The marked elements can be extracted and kept in their original order, or extracted and shuffled, or extracted into bins based on a tag they are marked with. The protocol for extracting marked elements and shuffling them is denoted $\Pi_{\text{Extract}}$ (§B.1); the protocol for extracting marked elements in order is denoted $\Pi_{\text{Extract–Ord}}$ (§B.2); and the protocol for extracting marked elements into bins is denoted $\Pi_{\text{Extract–Bin}}$ (§B.3).

Each of these protocols rely on the same fundamental approach. First, the protocol obtains a count of the number of marked elements, either by local computation or as an input. Then, using secure comparisons, they add additional dummy elements, some marked, and some unmarked, so that the total number of marked elements is input independent. Then the list is shuffled, the tags are revealed, and elements are placed in their correct location. Finally, if we wish to extract the elements in order, we also tag them with their original location, perform secure computations (before or after shuffling) to adjust this tag to their correct final locations, and then open this tag as well.

Both of our strategies for aligning blocks mentioned above require insertion and removal of dummy elements. In the lossy strategy, elements are sorted into bins based on which block from
the opposite list they are matched with, and elements that would overflow their bins are extracted into a separate list. This requires a call to $\Pi_{\text{Extract-Bin}}$ to obtain the elements for each bin, and a separate call to $\Pi_{\text{Extract-Ord}}$ to produce an ordered list of all overflow elements.

In the lossless strategy, dummy elements representing $n/k$ copies of the $k$-medians of each list are inserted into the opposite list, and then removed after merging on each subproblem. This requires a call to $\Pi_{\text{Extract-Ord}}$ to remove the dummies from the final list while keeping the ordering of the non-dummy elements unaltered.

### 3.6 Prepared vs. Executed Secure Merge

A number of our protocols require us to tag elements with their new locations after performing a merging subprotocol. In order to clarify the notation and avoid requiring excessive unnecessary shuffling, we treat separately the act of tagging each element with its eventual (merged) position, referred to as **prepared** merge ($\Pi_{\text{SM-prepare}}$), versus the act of actually moving elements as per their merge, referred to as **executed** merge ($\Pi_{\text{SM-secure}}$).

A prepared or executed merge protocol can be transformed into the other variant using a single secure shuffle. To go from executed merge to prepared merge, we tag the elements in the original lists with their original indices, and then after merging tag them with their corresponding indices in the destination list. Then the parties shuffle the destination list, open the original indices, and return the elements to their original positions. Meanwhile, to go from prepared merge to executed merge, the parties merely shuffle together both tagged lists, open the tags, and move the elements to the indices indicated by their tags. We give the details in Appendix A.

### 3.7 Primitive Functionalities

Our protocols are realized with black box calls to the following “primitive” functionalities: $\Pi_{\text{Open}}$, $\Pi_{\text{Comp}}$, $\Pi_{\text{Sel}}$, $\Pi_{\text{Shuffle}}$ that act on additively shared secret values. We explain these functionalities here and describe how they might be realized.

In the functionality $\Pi_{\text{Open}}$, the parties allow each other to learn the true value of any encrypted value. Under GMW-style MPC, each party sends their share to the other. Under homomorphic encryption, each party sends their share to the party who can decrypt, and then they broadcast the opened values.

In $\Pi_{\text{Comp}}$, the parties learn shares of a bit denoting the result of a comparison operator $[x > y]$, $[x \geq y]$ or $[x == y]$. These comparisons can be computed by converting to shares of bits and applying garbled circuits, which requires at least $O(W \log W)$ boolean gates on words of $W$ bits. A promising alternative is the GMW-based approach of Nishide and Ohta [NO07], which requires $O(W)$ bits of communication and $O(1)$ rounds of communication and is concretely efficient.

In $\Pi_{\text{Sel}}$, the parties perform multiplication of two values $[b]$ and $[x]$, where $b$ is known to be 0 or 1, and $x$ can be any value. Equivalently, the parties compute shares of the ternary operator $b ? x : 0$. The value $b$ can be XOR shared or additively shared over a larger field. $\Pi_{\text{Sel}}$ can be realized using any standard MPC multiplication, or by using string-OTs with some rerandomizable encryption.

In $\Pi_{\text{Shuffle}}$, the parties begin with shares of a given list, and end with shares of the same list, in some totally unknown order. To get the desired asymptotics, we require that the shuffle take $O(n)$ communication and $O(1)$ rounds of communication. Such protocols exist and can be realized via LWE with reasonably good ciphertext expansion [FO21]. At a high level, the shuffle works by allowing each party to hold an encryption of the list under the other party’s secret key, and then shuffling and re-randomizing; see [FO21] for the full protocol and a more thorough treatment of it.

We recall the secure merge protocol from [FNO22] discussed above, which requires $O(n)$ communication and $O(n)$ rounds, which we call $\Pi_{\text{SM-FNO}}$ in this work. Additionally, we use the trivial $O(n^2)$ communication, $O(1)$ round merge protocol, which works by making every possible pair of comparisons. We call this protocol $\Pi_{\text{SM-ALL}}$ and give it formally in §C.1 for completeness.
4 Overview of Merge Protocols

Our main protocol is built from the four sub-protocols described below in §4.1, §4.2, §4.3, and §4.4.

4.1 Abstract Secure Symmetric Merge

Our Abstract Secure Symmetric Merge $\Pi_{SSM}(n)$, which we give in Section 5.1, follows the lossless strategy for aligning blocks discussed in §3.3. In order to determine where the $k$ medians of each list map to in the other list, we require a Secure Asymmetric Merge protocol $\Pi_{SAM}(k, n)$, discussed below in §4.2 and given in full in Section 6. After determining where the $k$ medians map to, we make $n/k$ copies of each median, use a local protocol to determine each list element’s new destination index, and the tag-shuffle-reveal paradigm to route all elements to their destination in the new, expanded list. After this, we run $\Pi'_{SSM}(n/k)$ in parallel on each block.

We prove that this technique of expanding medians actually produces fully aligned blocks in Lemma D.1. We call this protocol Abstract Secure Symmetric Merge because it requires a choice of subprotocols to implement it. For our main result, we take $k = n/\log \log n$, and for $\Pi_{SSM}$ we use $\Pi_{SSM-FNO}$.

\begin{table}[!h]
\centering
\begin{tabular}{|c|c|}
\hline
Abstract Secure Symmetric Merge Protocol $\Pi_{SSM}(n)$ & \\
\hline
\hline
Input. Two parties $P_1, P_2$ (additively) secret-share two sorted lists $L_1$ and $L_2$, each of size $n$. \\
Also as input, parameter $k$ with $k/n$, and specifications of subprotocols $\Pi_{SSM}(n/k)$ and $\Pi_{SAM}(k, n)$. & \\
Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list \\
$L_1 \mathbin{\|} L_2$, which has size $2n$ and is (additively) secret-shared amongst the two parties. & \\
RCost. $O(1) + RCost(\Pi_{SSM}(n/k)) + RCost(\Pi_{SAM}(k, n))$. & \\
CCost. $2k \cdot CCost(\Pi_{SSM}(n/k)) + 2 \cdot CCost(\Pi_{SAM}(k, n)) + 8 \cdot CCost(\Pi_{Op})$. & \\
Protocol. & \\
\hline
1. Merge $M_{2,k}$ with $L_1$ by invoking $\Pi_{SSM}^{\text{prepare}}(k, n)$. Denote the outputs as $(\tilde{M}_{2,k}, \tilde{L}_1)$ (which are (additively) secret-shared amongst the two parties). & \\
2. Run the $\Pi_{SAM}^{\text{prepare}}(k, \tilde{M}_{2,k}, \tilde{L}_1)$ protocol (see Figure 13), which creates $k$ copies of $\tilde{M}_{2,k}$ and assigns all elements a destination index. Call $\Pi_{shuffle}'$ on the output, open the destination indices, and move the elements into a new list $L''_1$ as indicated by their destination indices. & \\
3. Repeat Steps 1-2 above, with the roles of $L_1$ and $L_2$ swapped. Let $L''_2$ denote the final output \\
(which has size $2n$ and is (additively) secret-shared amongst the two parties). & \\
4. Partition $L''_1$ and $L''_2$ into $2k$ blocks, each of size $n/k$, and run $\Pi_{SSM}(n/k)$ on each block. Concatenate the results from each block, and let $\hat{L}$ denote the output (which has size $4n$ and is (additively) secret-shared amongst the two parties). & \\
5. Run the $\Pi_{Extract-Ord}(\hat{L} \bowtie (\ell_i \neq 0)), 2n)$ protocol (see Figure 10) to remove the duplicates added in Steps 2 and 3 above. The output list has size $2n$, and is exactly (a secret-sharing of) $L_1 \mathbin{\|} L_2$. & \\
\hline
\end{tabular}
\caption{Abstract Secure Symmetric Merge Protocol. In our main protocol, we take $k = \log \log n$, $\Pi_{SSM}(n/k) = \Pi_{SSM-FNO}$, and $\Pi_{SAM} = \Pi_{SAM}(n, k, \Pi_{SSM-\log \log n})$.}
\end{table}

4.2 Abstract Secure Asymmetric Merge from Secure Symmetric Merge

We describe our Secure Asymmetric Merge protocol in Figure 2, and give proofs and analysis in §5.2. This protocol follows the lossy strategy for aligning blocks discussed in §3.3, and uses two auxiliary Secure Symmetric Merge protocols $\Pi_{SSM}, \Pi_{SSM'}$. The protocol begins by calling $\Pi_{SSM}^{\text{prepare}}(k)$ on the smaller list $L_1$ and the $k$ medians of $L_2$. The parties now collectively hold information about which blocks of $L_2$ each value of $L_1$ is assigned to, as well as how many elements of $L_1$ will be assigned to each block of $L_2$.

Whenever at most $n/k$ elements of the smaller list $L_1$ map to a block of $L_2$, we use $\Pi_{Extract-Bin}$ to extract these elements, giving $k$ subproblems of size $n/k$. We collect the remaining elements of
Secure Asymmetric Merge Protocol \(\Pi_{\text{SSM}}^{\text{prepare}}(k, n)\)

**Input.** Two parties \(P_1, P_2\) (additively) secret-share sorted list \(L_1\) of size \(k\) and \(L_2\) of size \(n\) (for \(k|n\)).

Also as input, specification of subprotocols \(\Pi_{\text{SSM}}^{\text{prepare}}(n/k)\) and \(\Pi_{\text{SSM}}^{\text{prepare}}(k)\).

**Output.** The two lists have been merged (i.e. combined so that the final list is sorted) into an output list \(L_1 \cup L_2\), which has size \(k + n\) and is (additively) secret-shared amongst the two parties.

**RCost.** \(O(1) + R\text{Cost(}\Pi_{\text{SSM}}^{\text{prepare}}(n/k)) + 2 \cdot R\text{Cost(}\Pi_{\text{SSM}}^{\text{prepare}}(k))\).

**CCost.** \((3n + k)\text{Cost(}\Pi_{\text{Sel}}) + (5n + 7k)\text{Cost(}\Pi_{\text{Comp}}) + (7n + 8k)\text{Cost(}\Pi_{\text{Open}}) + 4\text{Cost(}\Pi_{\text{Shuffle}}) + k \cdot \text{Cost(}\Pi_{\text{SSM}}^{\text{prepare}}(n/k)) + 2 \cdot \text{Cost(}\Pi_{\text{SSM}}^{\text{prepare}}(k))\).

**Protocol.**

1. Call in-place merge on \(L_1\) and the \(k\)-th medians of \(L_2\): \((\hat{L}_1, \hat{L}_2) := \Pi_{\text{SSM}}^{\text{prepare}}(L_1, M_{2,k})\).

2. Define \((\sigma_i)\) := \((i + \lfloor k/1 \rfloor \cdot n/k)\), so that \(\sigma_i\) is the correct index of \(\ell_{i,1}\) if \(\ell_{i,1}\) is less than all elements of the corresponding block of \(L_2\).

3. Choose random secrets \((r_i)\) and define \(T_1 := L_1 \bowtie ((\sigma_i)_i \bowtie ((r_i)_i \bowtie (e_i)_i))\).

4. Each party computes shares of \([\lambda_i] := [k/1] \cdot (k_\ell - k_{\ell-i,1}) \neq 0\). In other words, we tag the first \(n/k\) elements of \(L_1\) which belong between \(v_i-1\) and \(v_j\) in \(M_{2,k}\) with the label \(j\).

5. Write \((L_{1,m}^A) := \Pi_{\text{Extract-Bin}}(T_1 \bowtie (\lambda_i)_i, n/k, [k_\ell - 1, \ldots, n/k])\), for \(m = 1, \ldots, k\), that is, \(L_{1,m}^A\) holds all elements tagged with label \(m\), in order, followed by dummies, for \(1 \leq m \leq k\), and \(|L_{1,m}^A| = n/k\).

6. Construct a shared list \(T := ((k_{i,2} - k_{i,1}) > n/k) \bowtie (k_{i,2})\) of length \(k\), and extend it to length \(n\) by duplicating each share \(n/k\) times to give an expanded list \(T_n := (\tau_i)_i \bowtie (\tau_i)_i\).

7. Define \([e_i] := i\) for all \(i\) and \(T_2 := L_2 \bowtie (\{k_{i,i}\}) \bowtie (\{e_i\})\).

8. Call the extraction algorithm and compute \(L_{2,m}^B := \Pi_{\text{Extract-Ord}}(T_2 \bowtie (\{e_i\})_i, k)\), that is, \(L_{2,m}^B\) contains every block of \(L_2\) with at least \(n/k\) elements of \(L_1\) mapped to it, zipped with the flags \(k_{i,T}\) and \(e_i\).

9. Compute \(L_{2,m}^A := [T_2 \cdot (1 - \tau_i)]\), that is, \(L_{2,m}^A\) contains all blocks of \(L_2\) not extracted into \(L_{2,m}^B\), in their original positions: Partition \(L_{2,m}^A\) into blocks \((L_{2,m}^A)_i\) of each of size \(n/k\).

10. Compute \(L_{1,m}^A := L_{1,m}^A \cdot (1 - \tau_{mn/k})\) to replace all blocks \(L_{1,m}^A\) with dummies if the corresponding block \(L_{2,m}^A\) is dummy.

11. For \(m = 1, \ldots, k\), perform \((L_{1,m}^A, \hat{L}_{2,m}^B) := \Pi_{\text{SSM}}^{\text{prepare}}(L_{1,m}^A, L_{2,m}^A)\) on each of the \(k\) blocks of size \(n/k\), with \(L_{1,m}^A := L_{1,m}^A \bowtie (\mu_{i,j,m})_i\).

12. Let \(L_{2,m}^B := L_1\) and perform \((L_{1,m}^A, \hat{L}_{2,m}^B) := \Pi_{\text{SSM}}^{\text{prepare}}(L_{1,m}^B, L_{2,m}^B)\), with \(\hat{L}_{2,m}^B := L_{2,m}^B \bowtie (\mu_{i,j,m})_i\).

13. Define:

\[
L_1^1 = \bigcup_m (L_{1,m}^A \bowtie [(\mu_{1,m,1} - i) \bowtie (\sigma_i)]_1 \bowtie (1)_i) \cup L_{2,m}^B \bowtie [(\mu_{i,1,1} - i) \mod n/k] + [\sigma_i] \bowtie (2)_i.
\]

14. Take \(L_1^C := \Pi_{\text{Shuffle}}(L_1^C)\). Open the secrets \([r_i]\) and the tags 1 or 2 to match elements marked 1 from a set \(L_{1,m}\) with elements marked 2 from the set \(L_{1,m}^C\). Construct a new list \(L_{1,m}^C\) by using \(\Pi_{\text{Sel}}\) to copy the element marked 1 if it is non-dummy and the paired element marked 2 otherwise.

15. Define:

\[
\hat{L} = L_{1,m}^C \cup (L_{2,m}^B \bowtie (\mu_{i,2,m} - i + [e_i])_i) \cup (L_{2,m}^A \bowtie (\mu_{i,1,m} - i + [\sigma_i]_i + [e_i])_i)
\]

Compute \(T := L \bowtie (\{e_i\}) := \Pi_{\text{Extract}}(\hat{L} \bowtie (\{\ell_i \neq 0\}), n + k)\).

16. Open the values \([e_i]\), and re-arrange the entries of \(L\) to match the opened values \([e_i]\).

**Figure 2:** Secure Asymmetric Merge Protocol. In our main protocol, we take \(k = \log \log n\), \(\Pi_{\text{SSM}}^{\text{pre}} = \Pi_{\text{SSM-}}\text{FNO}\), and \(\Pi_{\text{SSM}}^{\text{pre}} := \Pi_{\text{SSM-}}\log \log n\).
4.3 Building Block: Secure Merge with $O(n \log \log n)$ Communication

As an ingredient for secure asymmetric merge, we need a secure symmetric merge with $O(n \log \log n)$ communication and $O(\log \log n)$ rounds, which we call $\Pi_{SSM-\log \log n}$ (Figure 3). This protocol in turn builds off of the $(n^{1/3}, n)$ asymmetric merge below, and follows the lossy strategy for aligning blocks described in §3.3.

As in $\Pi_{SAM}(n, k)$, taking the $k$-medians of $L_1$ (in this case, with $k = n^{1/3}$) and using the $\Pi_{SAM-n^{1/3}}$ protocol to merge with $L_2$ allows us to obtain destination indices for each element of the merged list, and counts of the number of elements from $L_2$ mapped to each block of $L_1$. We can similarly do the asymmetric merge of the $k$-medians of $L_2$ with $L_1$ to count the number of elements from $L_1$ mapped to each block of $L_2$.

For each block of $L_1$ that maps entirely inside a block of $L_2$ in the merged list, we extract that block together with all (less than $n/k$) corresponding elements from $L_2$, to get a subproblem of size $n/k = n^{2/3}$. After this process is complete, each block of $L_2$ has at most $2n/k$ elements mapped to it. Extracting these elements gives subproblems of size $2n/k$, which can be cut in half to give subproblems of size $n/k$.

When we count subproblems in this way, we have too many for the recursion to work correctly. However, as we show in Lemma 5.1, most of the subproblems created using a naive approach would contain all dummy elements. Shuffling and removing fully-dummy subproblems at each step gives us the recursion $CCost(\Pi_{SSM-\log \log n})(n) = O(n) + n^{1/3}CCost(\Pi_{SSM-\log \log n})(n^{2/3})$, from which the desired bounds follow immediately.

4.4 Building Block: Secure Asymmetric Merge on $(n^{1/3}, n)$

For the special case where $|L_1| = O(n^{1/3})$ and $|L_2| = O(n)$, we present in Figure 4 an asymmetric merge protocol with communication complexity $O(n)$ and round complexity $O(1)$. By calling $\Pi_{Compare-ALL}$ on $L_1$ and the $n^{2/3}$ medians of $L_2$, we can identify every block of $L_2$ which contains an element of $L_1$ after merging the two lists. Because there are $O(n^{1/3})$ elements of $L_1$, there are $O(n^{1/3})$ such blocks of $L_2$. After extracting these blocks, we run $\Pi_{Compare-ALL}$ again on $L_1$ and the $O(n^{2/3})$ elements of the extracted blocks of $L_2$. After some careful accounting of the index shifts during these steps, we get the destination indices for all the elements, which gives the desired merge protocol.

In Section 6, we give an extension of this protocol, which uses recursion to construct an asymmetric merge protocol with communication complexity $O(n)$ and round complexity $O(1)$ for a list $L_1$ of size $O(n^\alpha)$ and any fixed $\alpha < 1$, where the implied constants depend on $\alpha$. 
Secure Symmetric Merge $\Pi_{SSM}$-log-log(n)

Input. Two parties $P_1, P_2$ (additively) secret-share two sorted lists $L_1$ and $L_2$ of size $n$.

Output. The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $L_1 \cup L_2$, which has size $2n$ and is (additively) secret-shared amongst the two parties.

RCost: $O(\log \log n)$ rounds.

CCost: $O(n \log \log n) \cdot \text{CCost}(\Pi_{\text{Open}} + \Pi_{\text{Comp}} + \Pi_{\text{Sol}}) + O(\log \log n) \text{CCost}(\Pi_{\text{Shuffle}}(n))$.

Protocol.

1. Compute $d = \frac{\log \log n}{\log(3/2)} - O(1)$ such that $8 \leq n^{(2/3)^d} < 16\sqrt{2}$.
2. Define $A_0 := \{L_1, L_2, 0\}$, to be the top-level array of sub-problems of size $n$. Each subproblem consists of two lists $L_1, L_2$ and a final offset $\omega$. Then, for $k = 0, \ldots, d$, do Steps 3 through 18.
3. For each pair $(L_1, L_2, \omega) \in A_{k-1}$, do Steps 4 through 16.
4. Call $(L_1 \bowtie \iota_1, M_{2, n/3} \bowtie \theta_1, 2) := \Pi_{\text{Prepare}}_{\text{SAM}, n/3}(L_1, M_{2, n/3})$ and $(L_2 \bowtie \iota_2, M_{1, n/3} \bowtie \theta_1, 3) := \Pi_{\text{Prepare}}_{\text{SAM}, n/3}(L_2, M_{1, n/3}).$
5. Compute interactively $[s_1] := ([\theta_{(k+1)n^{2/3} + \iota} \cdot \theta_{n^{2/3} + 1} = 0])$ for $i = 1, \ldots, n^{1/3}$. The indicator $s_i$ denotes blocks of $L_1$ which map entirely inside of a block of $L_2$ during secure merge.
6. For $1 \leq i \leq n^{1/3}$ and $1 \leq m \leq n^{3/3}$, compute interactively $[\tau_{(i-1)n^{2/3} + m}] := \lfloor \theta_{(i-1)n^{2/3} + m} \rfloor = \lfloor \theta_{n^{2/3} - i} \rfloor$ and $[t_i] := [\tau_i > 0]$. Then re-define $[\tau_i] := [\tau_i \cdot [t_i]]$. The indicator $t_i$ denotes blocks of $L_2$ which envelop entire blocks of $L_1$ during secure merge, while $\tau_i$ counts the number of such blocks enveloped by the $i$th block of $L_2$.
7. Generate $T_2 := L_2 \bowtie (u_i)$ by generating (locally) uniformly random tags $u_i$ for each element of $L_2$.
8. Call $T_2 := \Pi_{\text{Extract}}_{\text{Ord}}(T_2 \bowtie ([t_i], \bowtie ([t_i, 2] - i)).$
9. Call $(B_j)_j := \Pi_{\text{Extract}}_{\text{Bin}}(T_2), 2n^{2/3}, [\theta_{i+1} - i], [\theta_{i+1} - \theta_{i+1} - n^{2/3}], (\theta_{i+1} - \theta_{i+1} - n^{2/3}))$, where the output is a series of lists $B_j$, for $1 \leq j \leq n^{1/3}$.
10. Delete the last $n^{2/3}$ entries of each $B_j$.
11. Partition $L_1$ into blocks based on $M_{1, n/3}$, and call the resulting blocks $(C_j)_j$, for $1 \leq j \leq n^{1/3}$. Append to $A_{k+1}$ every pair $(B_j \cdot [s_1], C_j \cdot [s_1], \text{ind}(B_j) + \text{ind}(C_j))$, where $\text{ind}(X)$ is the tag $\omega$ of subproblem $X$.
12. Call $\Pi_{\text{Shuffle}}$ on each of $T_2 \bowtie (i), \cup_j (B_j \bowtie (s_j))$. Open all values $u_i$ from each list, and match corresponding elements, then transfer the values $(s_j)_j$ to the shuffled list of $T_2 \bowtie (i)$. In other words, set $T_2$ equal to some permutation of $L_2$ of $(i), \bowtie (s_j))$.
13. Perform $\Pi_{\text{Shuffle}}$ again, open $(i)_i$, and rearrange to give $L''_2 := L_2 \cdot [1 - (\bar{s}_i)]_i).$ In other words, $L''_2$ holds every element of $L_2$ not already entered into $A_{k+1}$.
14. Partition $L''_2$ into $n^{1/3}$ blocks $(D_j)_j$ along the $n^{1/3}$-medians, as in Step Z.
15. Call $(C_j)_j := \Pi_{\text{Extract}}_{\text{Bin}}(L_1 \bowtie ([1 - s_1] \cdot ([i, 1] - i)), 2n^{2/3}, \cdot (i + 1) - i - n^{2/3} \cdot [\theta_{i+1, 2} - \theta_{i, 2} - n^{2/3}(1 + \tau_{i+1})))$. Append every pair $(D_j, E_j, \text{ind}(D_j) + \text{ind}(E_j))$ to $A_{k+1}$.
17. Compute $S_k := \sum_{i=1}^{k} 4^{1+k-i} n^{1-i}, \frac{1}{\tau_i}.$
18. Set $A_{k+1} = \Pi_{\text{Extract}}(A_{k+1} \bowtie ([\cup_i (a_i \neq 0) \neq 0], S_k))$.
19. Perform $\Pi_{\text{SM}}(L_1, L_2, \Pi_{\text{SM}} - \text{ALL})$ on every pair $(L_1, L_2, [\omega]) \in A_d$, and add $[\omega]$ to all resulting shares $(\epsilon_{i,j})$.

Figure 3: $O(n \log \log n)$ Secure Symmetric Merge Protocol
Cubic Secure Asymmetric Merge \((n^{1/3}, n)\) Protocol \(\Pi_{\text{SAM-}n^{1/3}}^{\text{prepare}}(n^{1/3}, n)\)

**Input.** Two parties \(P_1, P_2\) (additively) secret-share sorted list \(L_1\) of size \(n^{1/3}\) and \(L_2\) of size \(n\).

**Output.** The two lists have been merged (i.e. combined so that the final list is sorted) into an output list \(L_1 \bigcup L_2\), which has size \(n + n^{1/3}\) and is (additively) secret-shared amongst the two parties.

**RCost.** \(O(1)\).

**CCost.** \((3n + 2n^{2/3} + 2n^{1/3})\)\(\text{CCost}(\Pi_{\text{Comp}}) + (4n + 2n^{2/3})\)\(\text{CCost}(\Pi_{\text{Open}}) + 3\)\(\text{CCost}(\Pi_{\text{Shuffle}})(n + n^{2/3})\).

**Protocol.**

1. Run compare all on the list \(L_1\) and the \(n^{2/3}\)-medians of \(L_2\), that is, 
   \[
   (\hat{L}_1, \hat{L}_2) := \Pi_{\text{SM-ALL}}(L_1, M_{2,n^{2/3}}).
   \]
   Write elements of \(\hat{L}_j\) as \((a_{i,j}, \tau_i, \nu_i)\) so that \(\hat{L}_j = L_j \bowtie (\nu_i)\).

2. Construct a shared list \(T := ([t_{1,2} - i] \bowtie ([t_{1,2} - \tau_{i-1,2}] > 1))\) of length \(n^{2/3}\), and a second list \(T_n\) formed by copying each element of \(T\) a total of \(n^{1/3}\) times, so that \(T_n\) has total length \(n\). Define \(\tau_i\) to be \(i\) plus the first coordinate of the \(i\)th element of \(T_n\), that is, write \(T_n := ([\tau_i] - i) \bowtie (\nu_i)\).
   Note that \([\tau_i]\) denotes the correct index of \(a_i, 2\) after merging if no elements of \(L_1\) lie in the block to which the \(i\)th element of \(L_2\) belongs, and \(\nu_i\) is an indicator for whether \(a_i, 2\) lies in a block that contains an element of \(L_1\) after merging.

3. Define \(e_i := i\) for all \(i\), and construct \(T_2 := L_2 \bowtie ([\tau_i]) \bowtie ([e_i]) \bowtie ([\nu_i])\), a sequence of ordered \(4\)-tuples.

4. Construct \(T_2'\) similarly, but reverse the condition tested by \(T\), that is, replace each element \(([\tau_i], [\nu_i]) \in T_n\) with \(([\tau_i], 1 - [\nu_i])\).

5. Define \(L^A_1 := \Pi_{\text{Extract}}(T_2, n^{2/3})\) and \(L^B_1 := \Pi_{\text{Extract}}(T_2', n)\), so that the union of the non-dummy elements of \(L^A_1\) and \(L^B_1\) is \(L_2\).

6. Run compare all on the lists \(L_1\) and \(L^A_1\), that is, 
   \[
   (\hat{L}_1, \hat{L}^A_1) := \Pi_{\text{SM-ALL}}(L_1, L^A_1).
   \]
   Write elements of \(\hat{L}_1\) as \((a_{i,1}, [\tau_i, 1])\) and elements of \(\hat{L}^A_1\) as \((a^A_{i,1}, [\tau_i, 2], [e_i, 2], [\nu_i, 2])\).

7. Define \(T^A_2 := ([a^A_{i,1}], [e_i, 2] + [\nu_i, 2] - i, [\nu_i, 2])\).

8. Construct the list \(L^B_2 := \Pi_{\text{Shuffle}}(T^A_2 \cup L^B_1) := ([\ell_i] \bowtie (\phi_i, 2) \bowtie (e_i, 2))\).

9. Open the values \(e_i, 2\). Discard the dummy elements with \(e_i, 2 = 0\) and arrange the remaining values \((\ell_i, \phi_i, 2)\) in order based on these values. Call the resulting list \(L^B_2\).

10. Return \((L_1, i + n^{1/3} : ([t_{1,1} - i] + [\nu_i, 2] \text{ mod } n^{1/3}))\) and \(L^B_2\).

Figure 4: Cubic Secure Asymmetric Merge \((n^{1/3}, n)\) Protocol
5 Analyses of Protocols in Section 4

5.1 Analysis of Abstract Secure Symmetric Merge Protocol $\Pi_{SSM}(n)$ of §4.1

5.1.1 Security
The security of the $\Pi_{SSM}(n)$ protocol follows immediately from the security of the underlying $\Pi_{Dup}(k, m, n)$, $\Pi_{Shuffle}$, $\Pi_{Extract-Ord}$, $\Pi_{SSM^\prime}(n/k)$, and $\Pi_{SAM}(k, n)$ subprotocols.

5.1.2 Correctness
Assuming correctness of the $\Pi_{Dup}(k, m, n)$, $\Pi_{Shuffle}$, $\Pi_{Extract-Ord}$, $\Pi_{SSM^\prime}(n/k)$, and $\Pi_{SAM}(k, n)$ subprotocols, we need only demonstrate that the concatenation done in Step 4 above is correct, that is, that the blocks “align” as per the partitioning. Namely, this follows from Lemma D.1, which demonstrates that lists $L_1^\prime$ and $L_2^\prime$ have the same list of $2k$ medians (both equal $M_{1,k} \bigcup M_{2,k}$).

5.1.3 Cost

- Step (1) has $RCost(\Pi_{SAM}(k, n))$ and $CCost(\Pi_{SAM}(k, n))$.
- Step (2) has $RCost(\Pi_{Shuffle}(n) + O(1))$ and $CCost(\Pi_{Shuffle}(2n) + 2n \cdot \Pi_{Open})$, since $\Pi_{Dup}$ can be performed locally (see Figures 13).
- Step (3) repeats the costs of (1) and (2).
- Step (4) has $RCost(\Pi_{SSM^\prime}(n/k))$ and $2k \cdot CCost(\Pi_{SSM^\prime}(n/k))$.
- Step (5) has $RCost(\Pi_{Extract-Ord}(4n, 2n))$ and $CCost(\Pi_{Extract-Ord}(4n, 2n))$.

Using the cost of $\Pi_{Extract-Ord}$, the total cost is:

$$RCost(\Pi_{SSM}(n)) = RCost(\Pi_{SAM}(k, n)) + RCost(\Pi_{Shuffle}(2n)) + RCost(\Pi_{SSM^\prime}(n/k)) + RCost(\Pi_{Extract-Ord}(4n, 2n))$$

$$CCost(\Pi_{SSM}(n)) = 2 \cdot CCost(\Pi_{SAM}(k, n)) + CCost(\Pi_{Shuffle}(2n)) + 2n \cdot CCost(\Pi_{Open}) + 2k \cdot CCost(\Pi_{SSM^\prime}(n/k)) + CCost(\Pi_{Extract-Ord}(4n, 2n))$$

Plugging in $\Pi_{SAM}$ for $\Pi_{SSM^\prime}(n/k)$ in $\Pi_{SSM}$ and $\Pi_{SAM}$, and plugging in $\Pi_{SM-\log \log n}$ for $\Pi_{SSM^\prime}(k)$ in $\Pi_{SAM}$, (3) becomes:

$$RCost(\Pi_{SSM}(n)) = RCost(\Pi_{SSM^\prime}(k)) + 2 \cdot RCost(\Pi_{SSM^\prime}(k))$$

$$CCost(\Pi_{SSM}(n)) = O(1) + O(\log \log n) + O(\log \log n) = O(\log \log n).$$

5.2 Analysis of Abstract Secure Asymmetric Merge $\Pi_{SAM}(k, n)$ Protocol of §4.2

Security.
A simulator for the protocol for either party calls the simulator for each subprotocol, and then in Step 14 generates a random permutation of $k$ elements tagged 1 and $k$ elements tagged 2, and in step 16 generates random permutations of $\{1, \ldots, k\}$ and $\{1, \ldots, n\}$. By the correctness of the protocol, shown below, and the correctness of $\Pi_{Shuffle}$ and $\Pi_{Extract}$, Steps 14 and 16 do output uniformly random permutations of these elements during an honest run of the protocol. Also by the
Correctness.

To verify correctness, we proceed through the protocol, checking correctness of each claim about the intermediate tags used, the required bounds on number of dummy elements for the extraction protocols, and finally, verifying that the tags \( e_i \) opened at the end give the correct destination indices.

In Step 2, \( \kappa_{i,1} \) counts the number of medians from \( M_{2,k} \) less than \( \ell_{i,1} \), so \( \kappa_{i,1} \cdot n/k \) counts the number of elements in the blocks of \( L_2 \) which are placed entirely to the left of \( \ell_{i,1} \), and \( \sigma_i \) holds the index of \( \ell_{i,1} \) in the merged list, assuming that \( \ell_{i,1} \) is less than all elements in the block of \( L_2 \) to which it is matched, as claimed.

In Step 4, \( \kappa_{i,1} - \kappa_{i-n/k,1} = 0 \) if and only if \( \ell_{i,1} \) and \( \ell_{i-n/k,1} \) are mapped into the sample block of \( L_2 \), so that there are more than \( n/k \) elements to the left of \( \ell_{i,1} \) mapped into the same block. Thus \( \lambda_i \neq 0 \) precisely for the first \( n/k \) elements of \( L_1 \) mapped to each bucket of \( L_2 \), as claimed.

In Step 5, the above construction ensures that there are at most \( n/k \) elements tagged with each label, and they are consecutively placed within \( L_1 \). Because \( \kappa_{i-1,2} \) counts the number of elements of \( L_1 \) less than the \( i-1 \)st median of \( M_{2,k} \), \( \kappa_{i-1,2} + 1 \) gives the first index of \( L_1 \) greater than the \( i-1 \)st median of \( M_{2,k} \), that is, the index of the first element which maps into the \( i \)th block of \( L_2 \). Finally, \( \kappa_{i,2} - \kappa_{i-1,2} \) counts the total number of elements which map into the \( i \)th block of \( L_2 \), which is equal to the number of elements of \( L_1 \) tagged with \( i \) unless there are more than \( n/k \) of them, in which case exactly \( n/k \) elements are tagged with \( i \). Thus the conditions to call \( \Pi_{\text{Extract-Bin}} \) are satisfied.

In Step 6, the expression \( \kappa_{i,2} - \kappa_{i-1,2} > n/k \) is an indicator that checks if more than \( n/k \) elements of \( L_1 \) map to the \( i \)th block of \( L_2 \). After duplicating \( n/k \) times to give the list \( T_n \), the indicators \( \tau_i \) correspond to elements which belong to blocks with this property.

In Step 8, let \( B \) be the number of blocks of \( L_2 \) that have more than \( n/k \) elements mapped to them. Then \( n/kB < \lfloor n/1 \rfloor = k \), so \( B < k^2/n \), and the number of elements with \( \tau_i = 1 \) is less than \( k^2/n \cdot n/k = k \), and the conditions to call \( \Pi_{\text{Extract-Ord}} \) are satisfied.

In Step 9, all elements not marked by \( \tau \) are copied in place via calls to \( \Pi_{\text{Sel}} \). In Step 10, the corresponding blocks \( L^{A,m}_1 \) are copied in place. Since \( mn/k \) is of course an index from the \( m \)th block of \( L_2 \), \( 1 - \tau_{mn/k} \) is the desired indicator.

Now, since the tags \( [e_i] \) opened in Step 16 allows all elements to be returned to their original positions, we must now only verify that the tags zipped in on Steps 13, 14, and 15 give the desired destination indices.

For non-dummy elements \( \ell \in L^A_1 \), the expression \( \mu_{i,1,m}^A - i \) counts the number of elements from the \( m \)th block of \( L_2 \) less \( \ell \), so the tag \( \mu_{i,1,m}^A - i + [\sigma_i] \) gives the correct destination index of \( \ell \). Any element \( \ell \) of \( L_1 \) which is dummy in \( L^A_1 \) maps to one of the blocks that makes up \( L^B_2 \). Thus \( \mu_{i,1,m}^B - i \) counts the total number of elements of \( L^B_2 \) less than \( \ell \), which is equal to \( n/k \) times the number of whole blocks of \( L^B_2 \) less than \( \ell \) plus the number of elements from the block to which \( \ell \) is mapped. Thus reducing \( \mu_{i,1,m}^B - i \) modulo \( n/k \) gives the desired count of elements from \( L_2 \) in the matched block, and so adding \( \sigma_i \) gives the desired destination index. The procedure in Step 14 ensures that the computed destination index from \( L^A_1 \) is taken unless the element is dummy, in which case the computed destination index from \( L^B_1 \) is taken, as desired.

Elements of \( L_2 \) are rerouted into exactly one of \( L'^{A}_{2,m}, L'^{B}_{2} \). For elements \( \ell \in L'^{A}_{2,m} \), the value \( \kappa_{i,T} \) counts the number of elements of \( L_1 \) which sit entirely to the left of the \( m \)th block of \( L_2 \), while \( \mu_{i,2,m}^A - i \) counts the number of elements from \( L_1 \) which map into the \( m \)th block of \( L_2 \), but land to the left of \( \ell \). Adding back in \( e_i \) gives the correct output index. Elements \( \ell \in L'^{B}_{2} \) are compared to the entire list \( L_1 \) in Step 12, so \( \mu_{i,2,m}^B - i \) counts the number of elements from \( L_1 \) that land to the left, and adding back \( e_i \) gives the desired destination index.
Cost.
The cost of Steps 1, 11, and 12 are two calls to $\Pi^{\text{prepare}}_{\text{SSM}}(k)$ and $k$ calls to $\Pi^{\text{prepare}}_{\text{SSM}}(n/k)$. Steps 2 and 3 can be performed locally. The cost of Step 4 is $n$ calls to $\Pi_{\text{Sel}}$ and $n$ calls to $\Pi_{\text{Comp}}$, which can be executed in parallel. The costs of Steps 5, 8, and 15 are $O(1)$ rounds and the communication is $(4n + 3k)\text{CCost}(\Pi_{\text{Comp}}) + (6n + 5k)\text{CCost}(\Pi_{\text{Open}}) + 3\text{Shuffle}(n + k)$. The cost of Step 6 is $k$ calls to $\Pi_{\text{Comp}}$ which can be executed in parallel. Steps 7 and 8 can be computed locally. Steps 9 and 10 require $2n$ calls to $\Pi_{\text{Sel}}$. Step 14 requires a call to $\Pi_{\text{Shuffle}}(2k)$ and $k$ parallel calls to $\Pi_{\text{Sel}}$. Steps 14 and 16 together require $n + 3k$ calls to $\Pi_{\text{Open}}$.

Additionally, the call to $\min$ in Step 5 requires $k$ calls to $\Pi_{\text{Comp}}$, and the call to $\text{mod}$ in Step 13 requires $2k$ calls to $\Pi_{\text{Comp}}$ (both $\min$ and $\text{mod}$ can be computed by conditional subtraction of a public value, which requires no communication once shares of the conditional indicator have been generated). In total, this gives a cost of

$$O(1) + R\text{Cost}(\Pi^{\text{SSM}}_{\text{Sel}}(n/k)) + 2 \cdot R\text{Cost}(\Pi^{\text{SSM}}_{\text{Sel}}(k))$$

rounds and

$$\text{CCost}(\Pi^{\text{prepare}}_{\text{SAM}}(k, n)) = (3n + k) \cdot \text{CCost}(\Pi_{\text{Sel}}) + (5n + 7k) \cdot \text{CCost}(\Pi_{\text{Comp}}) + (7n + 8k) \cdot \text{CCost}(\Pi_{\text{Open}}) + 4 \cdot \text{CCost}(\Pi_{\text{Shuffle}})(n + k) + k \cdot \text{CCost}(\Pi^{\text{SSM}}_{\text{Sel}}(n/k)) + 2 \cdot \text{CCost}(\Pi^{\text{SSM}}_{\text{Sel}}(k)).$$

communication, with $O(n)$ computation.

5.3 Analysis of the $\Pi^{\text{SSM}}_{\log \log (n)}$ Protocol of §4.3

Security.
To simulate either player’s view during an execution of the protocol, a simulator can call the simulator of the sub-protocols on every step except for Step 12 and Step 13, since these are the two steps where values are opened to the parties. On Step 12, both parties only see a list of random values, and a second list that is some random permutation of these views can be simulated by randomly sampling from these distributions.

Correctness.

The values from $M_{1,n^{1/3}}$ and $M_{2,n^{1/3}}$ each partition the range of possible values into $n^{1/3} + 1$ blocks. There would be at most $n^{2/3}$ elements from either list between any two adjacent values. In an insecure protocol, we could recursively call $\Pi_{\text{SM}}$ on each of these sub-problems.

To make this protocol secure, we do something similar, hiding the size of the sub-problem while ensuring that values can be efficiently transferred into sub-problems without any sub-problem growing too large.

First, we take every block from $M_{1,n^{1/3}}$ that sits entirely inside of a block from $M_{2,n^{1/3}}$. Such blocks certainly correspond to a sub-problem of size $(n^{2/3}, n^{2/3})$ or smaller, and we extract all such sub-problems in Steps 5-11.

We then remove all of these extracted elements from our original list in Steps 12-14. Now each block from $M_{2,n^{1/3}}$ only touches remaining blocks from $M_{1,n^{1/3}}$ if the block from $M_{1,n^{1/3}}$ covers one or more of the endpoints of the block from $M_{2,n^{1/3}}$. (The only other option is that the block from $L_1$ lies entirely within the block from $L_2$, and we have already removed all of these cases). Therefore there are at most 2 blocks from $L_1$ that touch any block of $L_2$, which gives the $2n^{2/3}$ bound in Step 15.

The nonzero tags $[t_i] \cdot [t_{i,2} - i]$ in Step 8 are arranged in increasing order ordered. The largest possible tag is equal to $|M_{1,n^{1/3}}| = n^{2/3}$. The value $\theta_{i,1} - i$ denotes the index of the first entry of
$L_2$ that is greater than the $i$th element of $M_{1,n^{1/3}}$, while $\theta_{i+1,1} - \theta_{i,1} - n^{2/3}$ counts the number of elements from $L_2$ that map to this block of $L_1$. Therefore, after removing the zero tags, the list $T_2$ meets the requirements of $\text{Extract-Bin}$.

The correctness of the index lists in Step 15 are similar, with the values $\tau_i$ being used to shift the starting indices by the removed blocks and to deduct the removed blocks from the total counts.

To show that there are at most $S_k$ sub-problems generated in steps 11 and 16, we prove the following lemma:

**Lemma 5.1.** At a depth of $d$, there are at most

$$S_d := \sum_{i=1}^{d} 4^{1+d-i} n^{1-\frac{2i}{3^d}}$$

sub-problems without all elements dummy. For $d = \frac{\log \log n}{\log (3/2)} - O(1)$, we have

$$S_d \leq 8 n^{1 - \frac{2^d}{3^d}}.$$

**Proof.** We begin with the case $d = 1$. In Step 11, for every $j$ with $s_j = 1$, $B_j \cdot [s_j]$ contains no dummies, and otherwise $(B_j \cdot [s_j], C_j \cdot [s_j])$ are all dummies. In Step 16, $D_j$ contains no dummies unless some of the entries of the $j$th block of $L_2^n$ are already assigned to a pair $(B_j' \cdot [s_j'], C_j' \cdot [s_j'])$ from Step 11. Thus at least one in four non-empty sets contains $n^{2/3}$ elements, so there are at most $8n/n^{2/3} = 8n^{1/3}$ non-dummy sets, so that $|A_1| \leq 4n^{1/3}$, that is, there are at most $4n^{1/3}$ non-dummy sub-problems of size $n^{2/3}$.

Similarly, at any depth $d$, there can be at most $\frac{2n}{2^{2d}}$ sets containing $n^{2d}$ elements, all non-dummy. Again, sets can be divided into groups such that at least $1/4$ of sets are entirely non-dummy, except when the sets $B_j, C_j$ or $D_j, E_j$ both contain dummy elements. Because the dummy elements are all contiguous, at most two blocks from each of the $L_1$ and $L_2$ sub-lists contain a mix of dummy and non-dummy elements, and so at most four additional sub-problems arise in this way containing elements both dummy and non-dummy from each prior sub-problem. This gives the recurrence

$$S_d = \frac{4n}{2^{2d}} + 4S_{d-1},$$

which gives the first equation stated in the lemma. For the second equation, note that there exists some unique value of $d$, with $d < \frac{\log \log n}{\log (3/2)}$, such that $8 \leq n^{1/2d} < 16\sqrt{2}$. We thus have, for $i < d$,

$$\frac{4^{1+d-i} n^{1-\frac{2i}{3^d}}}{4^{1+d-(i+1)} n^{1-\frac{2i+1}{3^{d+1}}}} = 4n^{-\frac{2i+1}{3^{d+1}}} \leq \frac{1}{2},$$

and so

$$S_d \leq 4n^{1-\frac{1}{2^d}} \sum_{i=1}^{d} \left(\frac{1}{2}\right)^{d-i} < 8n^{1-\frac{2^d}{3^d}},$$

as desired. \qed

**Cost.**

At a depth $k$, Steps 4-16 require $O(n^{(2/3)k})$ communication and computation and $O(1)$ rounds for each element of $A_{k-1}$. By the lemma, at a depth of $k$, there are at most $O(n^{1-(2/3)k})$ subproblems, so the total work for Steps 3 through 18 is $O(n)$ communication and computation and $O(1)$ rounds. Since this loop is called $\log \log n - O(1)$ times, the total communication is $O(n \log \log n)$ and the round complexity is $O(\log \log n)$, as desired.
5.4 Analysis of Secure Asymmetric Merge $Π_{SM−n^{1/3}}$ Protocol of §4.4

Security.

Security follows from the security of the underlying protocols, after verifying that the conditions for the calls to $Π_{Extract}$ are satisfied and that nothing is revealed in the $Π_{Open}$ call in Step 9. We verify both of these conditions in our proof of correctness, below. Correctness.

The indicator $v_1$ introduced in Step 2 tests whether any elements of $L_1$ map into the $i$th block of $L_2$. Because there are $n^{1/3}$ total elements of $L_1$, there are at most $n^{1/3}$ such blocks, and so at most $n^{2/3}$ elements in the blocks, since the blocks have size $n^{1/3}$. Thus the condition for calling $Π_{Extract}$ to generate $L_2^3$ in Step 5 is satisfied. Since there are $n$ elements total in $L_2$, the condition for $L_2^B$ is also satisfied.

For elements of $L_1$, $v_{i,1} = i$ counts the number of blocks of $L_2$ situated entirely to the left, and $κ_{i,1}$ counts the number of elements of $L_2$ situated to the left, plus some additional blocks that are double counted. Thus, after modding out by $n^{1/3}$, The indices for $L_1$ computed in Step 10 are correct.

For elements $ℓ \in L_2^A$, the value $κ_{i,2} = i$ counts the total number of elements from all of $L_1$ that end left of $ℓ$, so adding back in $e_i$ gives the desired destination index. For element $ℓ \in L_2^B$, the value $τ_i$ computed in Step 2 is already the correct destination index, since no elements from $L_1$ maybe into the block of $ℓ$. Thus in Step 9, all elements of $L_2$ are given the correct destination index.

Cost.

Step 1 requires $n^{1/3} \cdot n^{2/3} = n$ calls to $Π_{Comp}$. Step 2 requires $n^{2/3}$ calls to $Π_{Comp}$. Step 3 and 4 can both be computed locally. Step 5 requires $n + n^{2/3}$ comparisons, $3n + n^{2/3}$ calls to $Π_{Open}$, and two calls to $Π_{Shuffle(n)}$. Step 6 again requires $n$ comparisons. Step 7 is free. Step 8 requires another call to $Π_{Shuffle}$. Step 9 requires $n + n^{2/3}$ calls to $Π_{Open}$. Finally, Step 10 requires $n^{1/3}$ modular reductions, which in turn requires $2n^{1/3}$ calls to $Π_{Comp}$.

This gives a total cost of $O(1)$ rounds and communication:

$$(3n + 2n^{2/3} + 2n^{1/3}) CCost(Π_{Comp}) + (4n + 2n^{2/3}) CCost(Π_{Open}) + 3 CCost(Π_{Shuffle})(n + n^{2/3})$$

6 Secure $n^α$ Asymmetric Merge Protocols

In this section, we generalize the Secure Asymmetric Merge protocol from §4.4 to work on a pair of lists $L_1$ of size $n^α$ (for any constant $α < 1$), and $L_2$ of size $n$, that runs in time $O(n)$ with $O(1)$, for any fixed $α < 1$, where the implied constants depend on $α$. As we show below, the implied constants are small for $α = 1/3$, and bounded above by $2^{2/(1−α)^3}$ in general for communication and $2/(1−α)^3$ for round complexity.

Our protocol works by bootstrapping up from an initial protocol for merging $n^{1.3}, n$, and going in general from

$$β \to γ = \frac{1}{β^2 − 3β + 3}$$

We write $Π_{SM−n^{α}}$ for a particular secure merge protocol for the constant $β < 1$, and $Π_{SAM−γ−β}$ for the protocol that bootstraps from $Π_{SM−n^{α}}$ to $Π_{SM−n^γ}$.

Note that the construction of $Π_{SM−n^{1/3}}$, which is the only subprotocol needed for our main protocol, was presented above in §4.4, and its analysis is given in §5.4.
6.1 Bootstrapping Protocol $\Pi_{\text{SM}-\beta-\gamma}$ Protocol

We give in Figure 5 a protocol that bootstraps from $\Pi_{\text{SM}-n^\beta}$ to $\Pi_{\text{SM}-n^\gamma}$, for $\gamma$ close to $\beta$, in $O(1)$ rounds and roughly twice the communication of the previous level of bootstrapping.

---

**Secure Asymmetric Merge Recursion Step Protocol $\Pi_{\text{prepare}}_{\text{SM}-\beta-\gamma}(n^\gamma, n, \Pi_{\text{prepare}}_{\text{SM}-\beta})$**

**Input.** Two parties $P_1, P_2$ (additively) secret-share sorted list $L_1$ of size $n^\gamma$ and $L_2$ of size $n$. Additionally, an algorithm $\Pi_{\text{prepare}}_{\text{SM}-\beta}$ is given, with

$$\gamma = \frac{1}{\beta^2 - 3\beta + 3}.$$  

**Output.** The two lists have been merged (i.e. combined so that the final list is sorted) into an output list $L_1 \cup L_2$, which has size $n + n^\gamma$ and is (additively) secret-shared amongst the two parties.

**RCost.** $O(1) + \text{RCost}(\Pi_{\text{SM}-n^\beta})(n)$.

**CCost.** $O(n) + 2\text{CCost}(\Pi_{\text{SM}-n^\beta})(n)$.

**Protocol.**

1. Run compare all on the list $L_1$ and the $n^{1-\gamma}$-medians of $L_2$, that is, 

   $$(\widehat{L}_1, \widehat{L}_2) := \Pi_{\text{prepare}}_{\text{SM}-\text{ALL}}(L_1, M_{2,n^{1-\gamma}}).$$

   Write elements of $\widehat{L}_j$ as $(a_{i,j}, t_{i,j})$ so that $\widehat{L}_j = L_j \bowtie (a_{i,j})_i$.

2. Generate a random secretly shared sequence $([r_i])_i$ of length $n^{1-\gamma}$ and construct a shared list $T := ([r_i, 2] - i \bowtie ([r_i, 2] - i_{i-1,2} > 1 + n^\beta]) \bowtie ([r_i])_i$. Generate a second list $T_0$ formed by copying each element of $T$ a total of $n^\gamma$ times, so that $T_0$ has total length $n$. As in Figure 4, define $T_i := ([r_i] - i, [r_i], [s_i])$, so that $T_i$ is the correct index of $a_{i,2}$ if no elements from $L_1$ map into the corresponding block of $L_2$ to which $a_{i,2}$ belongs, $[v_i]$ is an indicator for blocks matched to more than $n^\beta$ elements from $L_1$, and $[s_i]$ is the random value $[r_i]$ from the block to which $a_{i,2}$ belongs.

3. Define $e_i := i$ for all $i$, and construct $\mathcal{T}_2 := L_2 \bowtie ([r_i])_i \bowtie ([s_i])_i \bowtie ([v_i])_i$.

4. Define $([t_i])_i := [\min\{t_{i,2} - t_{i-1,2} - 1, n^\beta\}]$ and $([c_i])_i := [t_{i-1,2}] - (i - 1) + 1$.

5. Construct shares of the value $[\gamma_i] := [t_{i,1} - i] \cdot [1 + t_{i,1} - t_{i-1,2}, 1 > n^\beta]$ and $[\rho_i] := i + n^\gamma \cdot [t_{i,1} - i]$.

6. Define $(L_4)_{k,2}$ as the partition of $L_2$ into blocks of size $n^\gamma$ by the medians $M_{2,n^{1-\gamma}}$ and $L_2^B := \Pi_{\text{Extract-Ord}}(\mathcal{T}_2, n^{\gamma(2-\beta)})$.

7. Define $(L_4)_{k,2} := \Pi_{\text{Extract-Bin}}(L_1 \bowtie ([c_i])_i \bowtie ([\rho_i])_i \bowtie ([\sigma_i])_i, n^\beta, (c_i), (t_i), (\sigma_i))$ and $L_2^B := \Pi_{\text{Extract-Ord}}(\mathcal{T}_1 \bowtie ([c_i])_i \bowtie ([\rho_i])_i \bowtie ([\sigma_i])_i = 0, n^\gamma)$.

8. Call $(\widehat{L}_4, \widehat{L}_2) := \Pi_{\text{prepare}}_{\text{SM}-\beta}(L_4^B)_{k,2}$ in parallel on each pair of blocks, for $k = 1, \ldots, n^{1-\gamma}$, and write $\widehat{L}_4 := L_4^B \bowtie (\kappa_{i,2,k})_i$ as above.

9. Divide $L_2^B$ into blocks $B^B_{k,2}$ of size $n^\gamma(3-\beta) - 1$, and perform $(\widehat{B}^B_{k,2}, \widehat{B}^B_{k,2}) := \Pi_{\text{prepare}}_{\text{SM}-\beta}(B^B_{k,2}, L_2^B)$ for each $k$, in parallel, and define $\kappa_{i,j,k}$ as above.

10. Define $L_2^B := L_2^B \bowtie ([\gamma_i] + \sum_{k=1}^{n^{1-\gamma}(2-\beta) - 1} (\kappa_{i,2,k} - i))_i := L_2^B \bowtie (\kappa_{i,2,0})_i$.

11. Partition $L_2^B$ into blocks of size $n^\gamma$. Call $\Pi_{\text{Shuffle}}((\widehat{L}_2)_{k,2} \cup (T_2)_{k,2})$ on blocks.

12. Open the values $[\kappa_i]$ to match blocks, and for every pair of matching blocks, replace $[\kappa_{i,2,k}]$ with $[\kappa_{i,2,k} + \kappa_{i,2,0} - \epsilon_i]$. Call the result $L_2^B$.

13. Call $\Pi_{\text{Shuffle}}(L_2^B)$ again on blocks, and open the values $[c_i]$ and re-order the elements accordingly. Call the result $L_2^B$.

14. Write $L_2^B$ for the union of all lists $L_4^B_{i,2}$ and $B^B_{k,2}$, and define $L_2^C := L_2^B \bowtie ([\kappa_{i,j,k} - i + [\rho_{i,j,k}])], where $i$ is the index of the element within the list $L_4^B_{i,2}$ or $B^B_{k,2}$, and $\rho_{i,j,k}$ is the tag (appended in Step 5) of the corresponding element.

15. Call $\Pi_{\text{Shuffle}}(L_2^C)$, Open the values $[c_i]$, discard dummies, and re-order the remaining elements. Call the result $L_2^D$.

16. Return $L_2^D$ and $L_2^B$.

---

Figure 5: Secure Asymmetric Merge ($n^\gamma, n$) Bootstrapping Protocol
6.2 Secure Asymmetric Merge

In this section, we give the protocol for performing secure asymmetric merge in time $O(n)$ and $O(1)$ rounds for fixed $\alpha < 1$. In terms of $\alpha$, we give the upper bound of $O(n2^{1/(1-\alpha)^3})$ communication and $O(1/(1-\alpha)^3)$ rounds as $\alpha \to 1$.

The correctness and security of the protocol follows immediately from correctness and security of $\Pi_{\text{SAM}^{-n^{\alpha}}}$ and $\Pi_{\text{SAM}^{-\gamma \cdot \beta}}$, as long as we assume that $\gamma_i$ is constructed in Step 1 of this protocol actually terminates. We show that in fact it terminates in at most $2/(1-\alpha)^3$ steps, from which the above bounds on communication and round complexity follow from the communication and round complexity of the previous two protocols.

But indeed, for any $\beta < \alpha$, we have

$$\frac{1}{\beta^2 - 3\beta + 3} - \beta = \frac{(1 - \beta)^3}{(1 - \beta)(2 - \beta) + 1} \geq \frac{(1 - \beta)^3}{3} \geq \frac{(1 - \alpha)^3}{3}.$$

Since we begin with $\beta = 1/3$, it requires at most

$$\frac{\alpha - 1/3}{(1-\alpha)^3/3} \leq \frac{2}{(1-\alpha)^3}$$

steps of the recurrence before the sequence of $(\gamma_i)$’s becomes greater than $\alpha$. 

Security.

Security follows from the security of the underlying protocols, after verifying that the conditions for the calls to $\Pi_{\text{Extract}}$ are satisfied and that nothing is revealed in the $\Pi_{\text{Open}}$ call in Steps 12, 13, and 15. We verify both of these conditions in our proof of correctness, below. Correctness.

The lists $(L_{1,k}^A)$ contain the first $n^{3\gamma}$ elements mapped to each block of size $n^\gamma$ of $L_2$, and so we can call $\Pi_{\text{SM}^{-n^\beta}}$ on these pairs of blocks. We extract into $L_1^B$ all elements after the first $n^{3\gamma}$. There are at most $n^{\gamma}$ such elements. We extract into $L_2^B$ all blocks which have more than $n^{3\gamma}$ elements mapped to them. There are at most $n^{\gamma(1-\beta)}$ such blocks, and so at most $n^{\gamma(2-\beta)}$ elements.

After dividing $L_1^B$ into blocks in Step 9, there are $n^{1-\gamma(2-\beta)}$ subproblems each of size $(n^{\gamma(2-\beta)}, n^{(\gamma(3-\beta))-1})$. By the assumption that $\gamma = 1/(\beta^2 - 3\beta + 3)$, we have $\beta \cdot ((\gamma(3 - \beta)) - 1) = \gamma(2 - \beta)$, so we can apply the $\Pi_{\text{SM}^{-n^\beta}}$ protocol here as well, in Step 9.

Correctness follows from similar index chasing as in the previous protocols. Cost.

Step 1 requires $n$ comparisons. Step 2 requires $o(n)$ comparisons. Step 3 is free. Step 4 requires $n$ calls to min which gives $n$ calls to $\Pi_{\text{Comp}}$. Step 5 requires $o(n)$ calls to $\Pi_{\text{Comp}}$ and $\Pi_{\text{Sel}}$. Because $n^{\gamma(2-\beta)} = o(n)$, Step 6 requires $n + o(n)$ calls to $\Pi_{\text{Open}}$ and $o(n)$ calls to $\Pi_{\text{Comp}}$. Similarly Step 7 requires $2n \cdot \Pi_{\text{Comp}} + 3n \cdot \Pi_{\text{Open}} + n \cdot \Pi_{\text{Sel}}$ plus $o(n)$ communication. Step 8 requires $n^{1-\gamma}$ calls to $\Pi_{\text{SM}^{-n^\beta}}(n^{\beta\gamma}, n^{\gamma})$, and similarly Step 9 requires $n^{1-\gamma}$ calls to $\Pi_{\text{SM}^{-n^\beta}}(n^{\beta\gamma}, n^{\gamma})$. Finally, Steps 10 through 15 require $3n + o(n)$ calls to $\Pi_{\text{Open}}$ and 3 shuffles.

This gives a total cost of $O(1) + R\text{Cost}(\Pi_{\text{SM}^{-n^\beta}})$ rounds and

$$(4n + o(n))\text{Cost}(\Pi_{\text{Comp}}) + (7n + o(n))\text{Cost}(\Pi_{\text{Open}}) + n\text{Cost}(\Pi_{\text{Sel}}) + 6\text{Cost}(\Pi_{\text{Shuffle}})(n) + n^{1-\gamma}\text{Cost}(\Pi_{\text{SM}^{-n^\beta}}(n^{\gamma})) + n^{1-\gamma(2-\beta)}\text{Cost}(\Pi_{\text{SM}^{-n^\beta}}(n^{\gamma(2-\beta)}))$$

communication, and the last two terms can be bounded above by $2\text{Cost}(\Pi_{\text{SM}^{-n^\beta}}(n))$. 

6.2 Secure Asymmetric Merge ($n^\alpha, n$) Protocol

In this section, we give the protocol for performing secure asymmetric merge in time $O(n)$ and $O(1)$ rounds for fixed $\alpha < 1$. In terms of $\alpha$, we give the upper bound of $O(n2^{1/(1-\alpha)^3})$ communication and $O(1/(1-\alpha)^3)$ rounds as $\alpha \to 1$.

The correctness and security of the protocol follows immediately from correctness and security of $\Pi_{\text{SAM}^{-n^{1/3}}}$ and $\Pi_{\text{SAM}^{-\gamma \cdot \beta}}$, as long as the sequence $(\gamma_i)$ constructed in Step 1 of this protocol actually terminates. We show that in fact it terminates in at most $2/(1-\alpha)^3$ steps, from which the above bounds on communication and round complexity follow from the communication and round complexity of the previous two protocols.
Secure Asymmetric Merge \((n^\alpha, n)\) Protocol \(\Pi_{\text{SAM}}\)\((n^\alpha, n)\)

<table>
<thead>
<tr>
<th>Input</th>
<th>Two parties (\mathcal{P}_1, \mathcal{P}_2) (additively) secret-share sorted list (L_1) of size (n^\alpha) and (L_2) of size (n) (for (\alpha \leq 1/2)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>The two lists have been merged (i.e. combined so that the final list is sorted) into an output list (L_1 \cup L_2), which has size (n + n^\alpha) and is (additively) secret-shared amongst the two parties.</td>
</tr>
<tr>
<td>(R\text{Cost})</td>
<td>(O(1/(1 - \alpha)^3)).</td>
</tr>
<tr>
<td>(C\text{Cost})</td>
<td>(O(n \cdot 2^{2/(1 - \alpha)^3})).</td>
</tr>
</tbody>
</table>

**Protocol.**

1. Construct a finite sequence \((\gamma_i)\) as follows:
   - Set \(\gamma_1 := 1/3\), and \(i = 1\).
   - While \(\gamma_i < \alpha\), set \(\gamma_{i+1} \leftarrow \frac{1}{\gamma_{i} + 3}\gamma_{i} + 3\) and \(i \leftarrow i + 1\).

2. Let \(\ell\) equal the length of \((\gamma_i)\). Then for \(i = 1, \ldots, \ell - 1\), call \(\Pi_{\text{SAM}} - \gamma_{i+1} - \gamma_i(n^{\gamma_{i+1}}, n, \Pi_{\text{SAM}} - \gamma_i)\) to construct the protocol \(\Pi_{\text{SAM}} - \gamma_{i+1}\). This gives a protocol \(\Pi_{\text{SAM}} - \gamma_i\), for \(\gamma_i > \alpha\).

3. Pad the list \(L_1\) with \(n^\gamma - n^\alpha\) dummy elements, and call the result \(\mathcal{T}_1\).

4. Call \(\Pi_{\text{Extract}}(\Pi_{\text{SAM}} - \gamma_i(\mathcal{T}_1, L_2))\) to obtain the desired result, extracting all non-dummy elements.

Figure 6: Secure Asymmetric Merge \((n^\alpha, n)\) Protocol

### A Converting between Prepared Merge and Executed Merge

We give here the formal definitions and conversion protocols between secure *prepared merge* and secure *executed merge*. Secure executed merge, or \(\Pi_{\text{SM}}\text{execute}\), is the usual secure merge functionality, where parties input their additive shares of secret sorted lists \(L_1, L_2\) and receive back additive shares of the merged list \(L_1 \cup L_2\).

Secure prepared merge is a functionality that takes the same inputs, but instead outputs to each party shares of the lists \(\hat{L}_j := L_j \bowtie (t_{i,j})_i\), where \(t_{i,j}\) is the index of the \(i\)th element of \(L_j\) in \(L_1 \cup L_2\). By convention, we require that, when elements from \(L_1\) and \(L_2\) have equal values, the elements from \(L_1\) are to the left of the elements from \(L_2\) in the final merged list.

Below, we give a protocol for constructing a prepared merge protocol from an existing executed merge, and a protocol for constructing an executed merge protocol from an existing prepared merge.

#### A.1 Prepared Merge from Executed Merge

**A.1.1 Analysis**

**Security.**
Steps 1, 3, 6, and 7 are performed locally, and so are automatically secure. The security of Steps 2 and 4 follow from the security of \(\Pi_{\text{Shuffle}}\) and the underlying secure merge protocol \(\Pi_{\text{SM}}\). For Step 5, note that for \(1 \leq k \leq n_j\), there is a unique entry \(i\) with \(f_i = k\) and \(g_i = j\).

Therefore both players learn only a random shuffling of pairs \((i, j)\) for \(j \in \{1, 2\}\) and \(1 \leq i \leq n_j\), and either party could simulate the interaction with the other by generating such a random shuffling in Step 5 randomly, and computing the other party’s message from this information.

**Correctness.**
In step 3, the value \(h_i\) holds the destination index of each entry of the list, and steps 6 and 7 ensure that the values \((a_{i,j}, h_i) = (a_{i,j}, t_i)\) are placed back in their original positions in \(L_j\).

**Cost.**
The only interactive cost beyond the costs of \(\Pi_{\text{SM}}\) and \(\Pi_{\text{Shuffle}}\) is the cost of Step 5, which requires \(2(n_1 + n_2)\text{CCost}(\Pi_{\text{OP}})\) communication and \(O(1)\) rounds. This gives the numbers stated in the protocol.
Prepared Merge from Executed Merge $\Pi_{\text{execute} \rightarrow \text{prepare}}(L_1, L_2, \Pi_{\text{prepare}})$

Input. Two parties $P_1, P_2$ (additively) secret-share sorted lists $L_1, L_2$, with elements $\ell_{i,1} \in L_1$, $\ell_{i,2} \in L_2$. Additionally, we have a secure executed merge protocol $\Pi_{\text{execute}}$.

Output. A pair of shared lists $\hat{L}_1, \hat{L}_2$, such that $L_j = \hat{L}_j \bowtie (t_{i,j})$, and the index of $t_{i,j}$ in the merged list $L_1 \bigcup L_2$ is $t_{i,j}$.

$\text{RCost} = O(1) + \text{RCost}(\Pi_{\text{shuffle}}(n)) + \text{RCost}(\Pi_{\text{SM}}(n))$.

$\text{CCost} = \text{CCost}(\Pi_{\text{shuffle}}(n, 4)) + (2n_1 + 2n_2) \cdot \text{CCost}(\Pi_{\text{SM}}) + \text{CCost}(\Pi_{\text{SM}}(n, 3))$.

Protocol.

1. Generate the list $\overline{L}_1$ made up of elements $(\ell_{i,1}, t_{i,1}, 1)$, and likewise let $\overline{L}_2$ be the list of elements $(\ell_{i,2}, t_{i,2}, 2)$, where here $t_{i,j} := i$ is the index of the $i$th element of $L_j$.
2. Compute $L = \Pi_{\text{SM}}(\overline{L}_1, \overline{L}_2)$.
3. Append to each element $([e_i], [f_i], [g_i])$ of $L$ the element $[h_i] = i$.
4. Compute $\hat{L} = \Pi_{\text{shuffle}}(L)$.
5. Open the values $\hat{f}_i, \hat{g}_i$.
6. Set $\hat{L}_1$ equal to $([\hat{e}_i], [\hat{h}_i])$, for all $i$ with $\hat{g}_i = 1$, ordering the elements of $\hat{L}_1$ by the corresponding values of $\hat{f}_i$.
7. Set $\hat{L}_2$ equal to $([\hat{e}_i], [\hat{h}_i])$, for all $i$ with $\hat{g}_i = 2$, ordering the elements of $\hat{L}_2$ by the corresponding values of $\hat{f}_i$.

Figure 7: Secure Prepared Merge from Secure Executed Merge

A.2 Executed Merge from Prepared Merge

Exported Merge from Prepared Merge $\Pi_{\text{prepare} \rightarrow \text{execute}}(L_1, L_2, \Pi_{\text{prepare}})$

Input. Two parties $P_1, P_2$ (additively) secret-share sorted lists $L_1, L_2$, with elements $\ell_{i,1} \in L_1$, $\ell_{i,2} \in L_2$. Additionally, we have a secure prepared merge protocol $\Pi_{\text{prepare}}$.

Output. Each party holds additive shares of the merged list $L = L_1 \bigcup L_2$.

$\text{RCost} = O(1) + \text{RCost}(\Pi_{\text{shuffle}}(n)) + \text{RCost}(\Pi_{\text{prepare}}(n))$.

$\text{CCost} = \text{CCost}(\Pi_{\text{shuffle}}(n, (1, 1, 0))) + (n_1 + n_2) \cdot \text{CCost}(\Pi_{\text{SM}}) + \text{CCost}(\Pi_{\text{prepare}}(n, (1, 1, 0)))$.

Protocol.

1. Each party locally generates shares of the sequence $(e_i)_i$, with $e_i := i$.
2. Generate the lists $\overline{L}_j := L_j \bowtie ([e_i])$, that is, replace $[a_{i,j}]$ with $([a_{i,j}],[e_i])$.
3. Each party computes their share of $(\hat{L}_1, \hat{L}_2)$ using $\Pi_{\text{prepare}}(\overline{L}_1, \overline{L}_2)$. Write $\hat{L}_j = L_j \bowtie ([t_{i,j}])$.
4. Set $\hat{L} = \Pi_{\text{shuffle}}(\hat{L}_1 \cup \hat{L}_2)$. Call the resulting elements $([a_i], [t_i])$.
5. Open the values $t_i$.
6. In an empty list $L$ of length $n_1 + n_2$ place at index $j$ the (unique) $[a_i]$ with $t_i = j$, and return $L$.

Figure 8: Secure Executed Merge from Secure Prepared Merge

A.2.1 Analysis

Security.

Steps 1 and 2 are performed locally, and so are secure. The security of steps 3 and 4 follow from the security of the protocols $\Pi_{\text{prepare}}$ and $\Pi_{\text{shuffle}}$ (in other words, a simulator for $\Pi_{\text{inPlace} \rightarrow \text{SM}}$ can perform steps 1 and 2 as in the actual protocol, and use the simulator for the subprotocols in steps
3 and 4).

In Step 5, each party sees a uniformly random permutation of \( \{1, \ldots, n_1 + n_2\} \), which can be generated at random by a simulator. Note that the correctness of \( \Pi_{\text{SM}}^{\text{prepare}} \) guarantees that each value in this range occurs exactly once as an entry of \( \iota_i \), and the security of \( \Pi_{\text{Shuffle}} \) guarantees that these values are distributed randomly.

**Correctness.**

By the correctness of \( \Pi_{\text{SM}}^{\text{prepare}} \), step 6 places each value \( \ell_{i,j} \) into their correct final position in the output list \( L \).

**Cost.**

The only steps that require interaction are Step 3 and Step 4, where subprotocols are called, and Step 5, where \( n_1 + n_2 \) values are opened, requiring \( (n_1 + n_2) \Pi_{\text{Open}} \) communication and \( O(1) \) rounds. It follows immediately that round complexity is \( O(1) + \text{RCost}(\Pi_{\text{Shuffle}}(n)) + \text{RCost}(\Pi_{\text{SM}}^{\text{prepare}}(n)) \) and communication is \( \text{CCost}(\Pi_{\text{Shuffle}}(n, (1, 1, 0))) + (n_1 + n_2) \cdot \text{CCost}(\Pi_{\text{Open}}) + \text{CCost}(\Pi_{\text{SM}}^{\text{prepare}}(n, (1, 1, 0))) \).

## B Extraction Protocols

### B.1 Extracting Unordered Marked Elements

**Extraction Protocol** \( \Pi_{\text{Extract}}(A, t) \)

**Input.** Two parties \( P_1, P_2 \) (additively) secret-share a list \( A \) of size \( n \) of elements of the form \( (a_i, \iota_i) \), of which \( t' \leq t \) elements are marked by \( \iota_i = 1 \), and the other elements have \( \iota_i = 0 \).

**Output.** Each party holds additive shares of a list \( B \) of length \( t \) containing all elements \( a_i \) with \( \iota_i = 1 \), shuffled randomly.

**RCost.** \( O(1) + \text{RCost}(\Pi_{\text{Shuffle}}(n)) \).

**CCost.** \( \text{CCost}(\Pi_{\text{Shuffle}}(n, (1, 1, 0))) + (n_1 + n_2) \cdot \text{CCost}(\Pi_{\text{Op}}) \).

**Protocol.**

1. Generate shares of the total number of marked elements \( [t'] := [\sum_{i=1}^{n} \iota_i] \). This can be done locally.
2. For \( i = n + 1, \ldots, n + t \), append to \( A \) the element:
   \[
   (d_i, \iota_i) := (0, \delta_i).
   \]
   where \( \delta_i \) is an indicator on \( (i - n) > t' \). Each party sets their share of \( d_i \) to be 0, and computes their share of \( \iota_i \) using parallel invocations of a comparison protocol.
3. Invoke the shuffling protocol and set \( \hat{A} := \Pi_{\text{Shuffle}}(A) \).
4. Open all values \( \hat{\iota}_i \).
5. Initialize \( B \) to an empty list.
6. For \( i = 1, \ldots, n + t \), if \( \hat{\iota}_i = 1 \), each party copies their share of the value \( \hat{a}_i \) to \( B \).

**Figure 9:** Extract and Shuffle Marked Elements Protocol

### B.1.1 Analysis

**Security.**

Security of steps 1, 2, and 3 follow from security of the underlying comparison and shuffling protocols. By correctness (shown below), the values \( \hat{\iota}_i \) will contain \( t \) 1’s and \( n \) 0’s, and after calling the \( \Pi_{\text{Shuffle}} \) protocol, these values will be randomly arranged in \( \hat{A} \), and so both party’s view in Step 4 can be generated uniformly at random by a simulator.

**Correctness.**

There are exactly \( t - t' \) values in \( \{n + 1, \ldots, n + t\} \) with \( i - n > t' \), and so Step 2 adds \( t - t' \) values.
with \( \ell_i = 1 \) to the existing \( t' \) values with \( \ell_i = 1 \) in \( A \). These are the values placed into \( B \) in Step 6, and they are randomly distributed by the correctness of \( \Pi_{\text{shuffle}} \).

**Cost.**

Step 2 requires \( t \) comparison operations \( \Pi_{\text{Comp}} \) and Step 4 requires \( n + t \) opening operations. Combining with Step 3 gives \( O(1) + R\text{Cost}(\Pi_{\text{shuffle}}(n)) \) rounds and \( C\text{Cost}(\Pi_{\text{shuffle}}(n, (1, 0, 1)) + tR\text{Cost}(\Pi_{\text{comp}}) + (n + t)R\text{Cost}(\Pi_{\text{open}}) \).

### B.2 Extracting Ordered Marked Elements

<table>
<thead>
<tr>
<th>Extraction Protocol ( \Pi_{\text{Extract-Ord}}(A, t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input.</strong> Two parties ( P_1, P_2 ) (additively) secret-share a sorted list ( A ) of size ( n ) of elements of the form ( (a_i, \ell_i) ), of which ( t' \leq t ) elements are marked by ( \ell_i = 1 ), and the other elements have ( \ell_i = 0 ).</td>
</tr>
<tr>
<td><strong>Output.</strong> Each party holds additive shares of a list ( B ) of length ( t ) containing all elements ( a_i ) with ( \ell_i = 1 ), in order, followed by dummy elements.</td>
</tr>
<tr>
<td><strong>RCost.</strong> ( O(1) + R\text{Cost}(\Pi_{\text{shuffle}}(n)) ).</td>
</tr>
<tr>
<td><strong>CCost.</strong> ( C\text{Cost}(\Pi_{\text{shuffle}}(n, (1, 1, 1)) + t \cdot R\text{Cost}(\Pi_{\text{comp}}) + (n + 2t) \cdot R\text{Cost}(\Pi_{\text{open}}) ).</td>
</tr>
</tbody>
</table>

**Protocol.**

1. Each party computes locally shares of \( [e_i] := (\sum_{j \leq i} [\ell_i]) \), and define \( [t'] := [e_n] \) to be a share of the total number of marked elements.
2. Generate locally shares of the list: \( \overline{A} := (a_i) \bowtie ([\ell_i]) \bowtie ([e_i]) \). |
3. For \( i = n + 1, \ldots, n + t \), compute \( [\delta_i] \) to be an indicator representing the comparison \( (i - n) > t' \). |
4. For \( i = n + 1, \ldots, n + t \), append to \( \overline{A} \) the element: \( ([a_i], [\ell_i], [e_i]) := (0, [\delta_i], [\delta_i] \cdot (i - n)) \). |
5. Invoke the shuffling protocol and set \( \overline{A} := \Pi_{\text{shuffle}}(\overline{A}) \). |
6. Open all values \( \widehat{e}_i \). |
7. Open all values \( \widehat{e}_i \) for values \( \widehat{e}_i = 1 \). |
8. Initialize \( B \) to an empty list. |
9. For \( i = 1, \ldots, t \), set the \( i \)-th element of \( B \) to the (unique) element \( [\widehat{a}_j] \) with \( \widehat{e}_j = i \).

**Figure 10:** Stably Extract Marked Elements Protocol

### B.2.1 Analysis

**Security.**

Security of steps 1, 3, 4, and 5 follow from security of the underlying selection, comparison, and shuffling protocols. By correctness (shown below), the values \( \widehat{e}_i \) correspond to the integers from 1 to \( t \), and after calling the \( \Pi_{\text{shuffle}} \) protocol, these values will be randomly arranged in \( \overline{A} \), and so both party’s view in Step 6 can be generated uniformly at random by a simulator. All remaining steps are performed locally, and can be imitated by a simulator.

**Correctness.**

As in \( \Pi_{\text{Extract}} \), there are exactly \( t \) values in \( \{1, \ldots, n + t\} \) with \( e_i \neq 0 \), the values which will be extracted into \( B \). For the \( t' \) non-dummy values from \( A \) with \( \ell_i = 1 \), the index \( [e_i] \) counts the number of non-dummy values to the left of \( a_i \) in \( A \), and so \( [e_i] \) holds the destination index of \( a_i \), and collectively, the indices \( [e_i] \) take on the values \( \{1, \ldots, t\} \).

The remaining \( t - t' \) values in \( \{n + 1, \ldots, n + t\} \) with \( i - n > t' \) are in fact \( n + t' + 1, \ldots, n + t \), and so the corresponding values \( [e_i] \) are \( t' + 1, \ldots, t \). Thus the nonzero values of \( e_i \), for \( 1 \leq i \leq n + t \), collectively cover the values \( \{1, \ldots, t\} \), with the first \( t' \) values being the non-dummy elements of \( A \), in ordered, as desired.

**Cost.**
Note that Step 1 requires linear computation when implemented via the recursion \( e_i = [i] + [e_{i-1}] \), and zero communication, since it can be performed locally.

Step 2 requires \( t \) comparison operations \( \Pi_{\text{Comp}} \) and Step 4 requires \( n + t \) opening operations. Combining with Step 3 gives \( O(1) + R\text{Cost}(\Pi_{\text{Shuffle}}(n)) \) and zero communication, since it can be performed locally.

B.3 Stably Extracting into Bins

<table>
<thead>
<tr>
<th>Stable Bin Extraction Algorithm ( \Pi_{\text{Extract-Bin}}(A, t, C, T') )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input.</strong> Two parties ( P_1, P_2 ) (additively) secret-share a sorted list ( A ) of size ( n ) of elements of the form ((a_i, \iota_i)), with ( \iota_i \in {0, \ldots, k} ), and ( t' \leq t ) elements have ( \iota_i = j ), for ( j = 1, \ldots, k ), with ( tk \leq \ell n ), for some given constant ( \ell ). Additionally, we are guaranteed that all elements with ( \iota_i = j ), for ( j \neq 0 ), are in a contiguous block. Finally, all parties hold shares of another pair of lists ((C, T')), each of length ( k ), with the property that ([c_j]) is the index of the first element of ( A ) with ( \iota_i = j ), and ([t'_j]) is the number of elements of ( A ) with ( \iota_i = j ) (if no such element ( \iota_i ) exists, ([t'_j] := [c_j] := 0)).</td>
</tr>
<tr>
<td><strong>Output.</strong> Each party holds additive shares of a sequence of lists ((B_j)), for ( j = 1, \ldots, k ), where each list ( B_j ) has length ( t ) and contains all elements ( a_i ) with ( \iota_i = j ), in the same order that they occur in ( A ), followed by dummy elements.</td>
</tr>
<tr>
<td><strong>RCost.</strong> ( O(1) + R\text{Cost}(\Pi_{\text{Shuffle}}(n)) ).</td>
</tr>
<tr>
<td><strong>CCost.</strong> ( C\text{Cost}(\Pi_{\text{Shuffle}}(1 + \ell n, (1, 2, 0))) + (1 + 2\ell) \cdot C\text{Cost}(\Pi_{\text{Comp}}) + (2 + 2\ell)n \cdot C\text{Cost}(\Pi_{\text{Open}}) + (1 + \ell)n \cdot C\text{Cost}(\Pi_{\text{Sel}}) ).</td>
</tr>
<tr>
<td><strong>Protocol.</strong></td>
</tr>
<tr>
<td>1. Append ( tk ) elements to ( A ), where for ( n + t(j - 1) &lt; i \leq n + tj ), we define ((a_i, \iota_i) := (0, [i - (n + t(j - 1))] &gt; t'_j))).</td>
</tr>
<tr>
<td>2. Generate the shared list ( A' = A \bowtie ([\kappa_i])_i ), where ( \kappa_i ) is equal to ( i ) reduced modulo ( t ). This can be done locally by ( P_1 ) setting their share equal to ( \kappa_i ) and all other parties setting their share equal to zero.</td>
</tr>
<tr>
<td>3. Invoke the shuffling algorithm and set ( \hat{A} := \Pi_{\text{Shuffle}}(A') ).</td>
</tr>
<tr>
<td>4. Open all values ( \hat{\kappa}_i ).</td>
</tr>
<tr>
<td>5. For ( i = 1, \ldots, n + tk ), let ( j = \hat{\iota}_i ), and set ([\hat{\kappa}_i] \leftarrow ([\hat{\kappa}_i] - [c_j - 1] \cdot [\hat{\kappa}_i \neq 0]) ) (mod ( t )).</td>
</tr>
<tr>
<td>6. Open all values ( \hat{\kappa}_i ).</td>
</tr>
<tr>
<td>7. Initialize ((B_j)) as a sequence of ( k ) lists of length ( t ).</td>
</tr>
<tr>
<td>8. Set the ( \ell )-th element of ( B_j ) equal to the (unique) element of ( \hat{A} ) with ( \hat{\iota}_i = j ) and ( \hat{\kappa}_i = \ell ).</td>
</tr>
</tbody>
</table>

Figure 11: Stably Extract Marked Elements By Bin Protocol

B.3.1 Analysis

**Security.**

A simulator can simulate the view of either party during Steps 1, 3, and 5 by invoking a simulator of the underlying protocols, and can imitate Steps 2, 7, and 8 exactly, since these steps are performed locally. Security for these steps follows from the security of the underlying protocol.

We show in our discussion of correctness that, in Step 6, the parties learn a random permutation of the \( tk \) ordered pairs \((i, j)\) with \( 1 \leq i \leq t \) and \( 1 \leq j \leq k \), while in Step 4, the parties learn the first coordinates of all terms in that permutation. During Step 4 of a simulated execution of the protocol, the simulator generates and stores the permutation for Step 6 uniformly at random, and then computes the resulting values for Step 4.

**Correctness.**
As in $\Pi_{\text{Extract}}$ and $\Pi_{\text{Extract-Ord}}$, the comparison operation in Step 1 adds $t - t'_{j}$ elements with $\iota_{i} = j$. The sequence of values $\kappa_{i}$ for these dummy values is $\{t'_{j} + 1, \ldots, t - 1, 0\}$, since the term with $\kappa_{i} = t$ is reduced modulo $t$.

Before Step 5, the remaining non-dummy values with $\iota_{i} = j$ have values $\kappa_{i}$ equal to the sequence $\{c_{j}, c_{j} + 1, \ldots, c_{j} + t'_{j} - 1\}$, taken modulo $t$, where entry $c_{j} + (h - 1)$ corresponds to the $h$th element of $A$ with $\iota_{i} = j$. Therefore, after Step 5, the values $\kappa_{i}$ take on the values $\{1, 2, \ldots, t'_{j}\}$, where entry $k$ corresponds to the $h$th element of $\iota_{i} = j$.

Therefore, for $1 \leq h \leq t$ and $1 \leq j \leq k$, every pair $(h, j)$ occurs exactly once in Step 6, and, for a fixed $j$, the first $t'_{j}$ entries of $B_{j}$ are the non-dummy elements of $A$ with $\iota_{i} = j$, in order, as desired.

**Cost.**

Step 1 requires $tk \leq \ell n$ comparisons. Steps 4 and 6 require $(1 + \ell)n$ opening of secret values. Step 5 requires $k$ subtractions, which of course can be performed locally, $(1 + \ell)n$ comparisons (to zero) and $(1 + \ell)n$ selection operations. Steps 2, 7, and 8 are performed locally. Each of Steps 1, 4, 5, and 6 therefore require $O(1)$ rounds.

This gives a total round cost of $O(1) + R\text{Cost}(\Pi_{\text{Shuffle}}(n))$ and a total communication cost of $CC\text{Cost}(\Pi_{\text{Shuffle}}((1 + \ell)n, (1, 2, 0))) + (1 + 2\ell) \cdot C\text{Cost}(\Pi_{\text{Comp}}) + (2 + 2\ell)n \cdot CC\text{Cost}(\Pi_{\text{Open}}) + (1 + \ell)n \cdot CC\text{Cost}(\Pi_{\text{Sel}})$, with linear total computation by each party.

### C Other Sub-Protocols

In this section, we present other sub-protocols invoked by any of the Secure Merge protocols above.

#### C.1 Secure Symmetric Merge via Compare All

The following protocol (Figure 12) is a naïve protocol that simply performs all $n^2$ possible comparisons (securely) and is useful for terminating iterative/recursive processes when the reduced list size $n$ is sufficiently small.

<table>
<thead>
<tr>
<th>Secure Merge - Compare All $\Pi_{\text{SM-ALL}}(n_1, n_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input.</strong> Two parties $P_1, P_2$ (additively) secret-share two sorted lists $L_1$ and $L_2$, of size $n_1, n_2$, with elements $\ell_{i,j}$, for $j \in {1, 2}$. When $n_1 = n_2 = n$, we write $\Pi_{\text{SM-ALL}}(n)$.</td>
</tr>
<tr>
<td><strong>Output.</strong> Lists $\tilde{L}_1, \tilde{L}<em>2$, with $\tilde{L}<em>j = L_j \bowtie (i</em>{i,j})<em>1$, where $i</em>{i,j}$ denotes the index of $\ell</em>{i,j}$ in $L_1 \cup L_2$.</td>
</tr>
<tr>
<td><strong>R\text{Cost.}</strong> $O(1)$ rounds.</td>
</tr>
<tr>
<td><strong>C\text{Cost.}</strong> $n_1 n_2 \cdot C\text{Cost}(\Pi_{\text{Op}})$.</td>
</tr>
<tr>
<td><strong>Protocol.</strong></td>
</tr>
<tr>
<td>1. Compute shares $[e_{i,j}] = [(x_{i1} &gt; x_{j2})]$ for every pair $(i, j) \in {1, \ldots, n_1} \times {1, \ldots, n_2}$.</td>
</tr>
<tr>
<td>2. Locally compute $[e_{i,1}] = \sum_{j=1}^{n_2} [e_{i,j}]$ and $[e_{i,2}] = n_1 - \sum_{j=1}^{n_2} [e_{j,i}]$.</td>
</tr>
<tr>
<td>3. Return $(\tilde{L}<em>1, \tilde{L}<em>2) := (L_1 \bowtie (i + i</em>{1,1}), L_2 \bowtie (i + i</em>{1,2}))$.</td>
</tr>
</tbody>
</table>

**Figure 12:** Secure Merge - Compare All Protocol

#### C.1.1 Analysis of the $\Pi_{\text{SM-ALL}}(n_1, n_2)$ Protocol

**Security.**

Security follows from the security of the underlying comparison protocol used in Step 1, since Steps 2 and 3 are performed locally.

**Correctness.**

By convention, we require elements from $L_1$ to be to the left of elements from $L_2$ if their values are equal.
Because this protocol is executed locally, it requires zero rounds and no communication. We need to prepare \( \Pi_{\text{Comp}} \).

**Cost.**

- Step (1) incurs \( O(1) \) round cost (the cost of the underlying comparison protocol) and \( 2n_1n_2 \) \( \text{Cost}(\Pi_{\text{Comp}}) \), since the shares \([e_{i,j}]\) and \([f_{i,j}]\) can all be computed in parallel, and there are \( 2n_1n_2 \) such comparisons to compute.

- Steps (2) and (3) can be performed locally.

### C.2 Duplicate Values In Place \( \Pi_{\text{Dup}}^{\text{prepare}}(k, L_1 \bowtie (i_{i,1}), L_2 \bowtie (i_{i,2})) \)

**Input.** Two parties \( \mathcal{P}_1, \mathcal{P}_2 \) (additively) secret-share two sorted lists \( \tilde{L}_1 \) and \( \tilde{L}_2 \), of sizes \( n_1, n_2 \) respectively, with \( \tilde{L}_j := L_j \bowtie (i_{i,j}) \), with the property that \( i_{i,j} \) is the index of \( \ell_{i,j} \) in the merged list \( L_1 \cup L_2 \).

**Output.** Two (sorted, additively secret-shared) output lists \( \tilde{L}_1', \tilde{L}_2' \) of size \( kn_1 \) and \( n_2 \), respectively, with the property that \( \tilde{L}_j' := L_j' \bowtie (i_{i,j}') \). \( L_1' = L_2 \), and \( L_1' \) consists of \( k \) copies of each entry of \( L_1 \), sorted in place. The values \( i_{i,j}' \) give the index of the entries \( \ell_{i,j}' \) in the merged list \( L_1' \cup L_2' \) of size \( n_2 + k \cdot n_1 \).

**RCost.** None.

**CCost.** None. (i.e. this protocol requires only \( O(n_2 + kn_1) \) local computations).

**Protocol.**

1. Each party constructs their share of the list \( L_1' \) by replacing each share \([\ell_{i,1}]\) with \( k \) copies, now in positions \([\ell_{k \cdot (i-1)+m}]\), for \( 1 \leq m \leq k \).
2. Each party generates their shares of the list \( (i_{i,1}') \) by setting \([i_{k \cdot (i-1)+m}'] := (|i_{i}|-i)+k \cdot (i-1)+m \), for \( 1 \leq m \leq k \).
3. Each party generates their shares of the list \( (i_{i,2}') \) by setting \([i_{k \cdot (i-1)+m}'] := k \cdot (|i_{i}|-i) + i \).
4. Return \((L_1' \bowtie (i_{i,1}'), L_2' \bowtie (i_{i,2}'))\).

**Figure 13:** Duplicate Values In Place Protocol

### C.2.1 Analysis

**Security.**

This protocol is executed locally, and is therefore secure.

**Correctness.**

Each entry \( \ell_{i,1} \) in \( L_1 \) has \( i_{i,1} - i \) elements from \( L_2 \) to its left in \( L_1 \cup L_2 \), since it has \( i_{i,1} - 1 \) elements total to its left, of which \( i-1 \) come from \( L_1 \). The corresponding element \( \ell_{k \cdot (i-1)+m} \), for \( 1 \leq m \leq k \), will likewise have \( i_{i,1} - i \) elements from \( L_2 \) to its left in \( L_1 \cup L_2' \), and so a total of \(|i_{i}|-i)+k \cdot (i-1)+m-1 \) elements to its left.

Similarly, each entry \( \ell_{i,2} \) in \( L_2 \) has \( i_{i,2} - i \) elements from \( L_1 \) to its left in \( L_1 \cup L_2 \), and \( k \cdot (i_{i,2} - i) \) elements to its left in \( L_1' \cup L_2' \), and correctness follows.

**Cost.**

Because this protocol is executed locally, it requires zero rounds and no communication. We need to check only that it requires \( O(n_2 + kn_1) \) computation. Steps 1 and 2 each require \( O(kn_1) \) computation, and Step 3 requires \( O(n_2) \) computation.
D Results on Medians

Notice that if we write $M_{j,k} = \{v_1, v_2, \ldots, v_k\}$, then:

(a) $|M_{j,k}| = k$
(b) If $k = 1$, then: $M_{j,1} = \{u_n\}$ (the last element of $L_j$)
(c) If $k = n = |L_j|$, then: $M_{j,n} = L_j$
(d) $\forall v_i \in M_{j,k}$: There are at least $i \cdot \left(\frac{n}{k}\right)$ items in $L_j$ that are $\leq v_i$
(e) $\forall v_i \in M_{j,k}$ and $v_i > v \in L_j$: There are fewer than $i \cdot \left(\frac{n}{k}\right)$ items in $L_j$ that are $\leq v$
(f) $\forall v_i \in M_{j,k}$: There are at least $1 + (k - i) \cdot \left(\frac{n}{k}\right)$ items in $L_j$ that are $\geq v_i$
(g) $\forall v_i \in M_{j,k}$ and $v_i < v \in L_j$: There are fewer than $1 + (k - i) \cdot \left(\frac{n}{k}\right)$ items in $L_j$ that are $\geq v$
(h) $v_k = u_n$ (i.e. the last element of $M_{j,k}$ is the last element of $L_j$)  

(5)

For any value $v \in S$, where $S$ is a totally ordered set, let:

$$
LT L_j(v) = \text{Number of elements in } L_j \text{ that are less than } v \\
LT E L_j(v) = \text{Number of elements in } L_j \text{ that are less than or equal to } v \\
GT L_j(v) = \text{Number of elements in } L_j \text{ that are greater than } v \\
GT E L_j(v) = \text{Number of elements in } L_j \text{ that are greater than or equal to } v
$$

Note that each of the above four functions are monotone step functions with respect to $v$ (increasing for $LT$ and $LTE$ and decreasing for $GT$ and $GTE$), with jumps at each $v \in L_j$. Then for any $v \in L_j$ and any $v_i \in M_{j,k}$:

$$
n = LTE L_j(v) + GTE L_j(v) = LT L_j(v) + GTE L_j(v) \quad \text{(6a)}
$$

$$
LTE L_j(v_i) \geq i \cdot \left(\frac{n}{k}\right) \quad \text{(6b)}
$$

$$
LTE L_j(v) < i \cdot \left(\frac{n}{k}\right) \iff v_i > v \quad \text{(6c)}
$$

$$
LTE L_j(v) \geq i \cdot \left(\frac{n}{k}\right) \iff v_i \leq v \quad \text{(6d)}
$$

$$
LTE L_j(v) = i \cdot \left(\frac{n}{k}\right) \implies v_i = v \quad \text{(6e)}
$$

$$
GTE L_j(v_i) \geq 1 + (k - i) \cdot \left(\frac{n}{k}\right) \quad \text{(6f)}
$$

$$
GTE L_j(v) < 1 + (k - i) \cdot \left(\frac{n}{k}\right) \iff v_i < v \quad \text{(6g)}
$$

$$
GTE L_j(v) \geq 1 + (k - i) \cdot \left(\frac{n}{k}\right) \iff v_i \geq v \quad \text{(6h)}
$$

$$
GTE L_j(v) = 1 + (k - i) \cdot \left(\frac{n}{k}\right) \implies v_i \leq v \quad \text{(6i)}
$$

where (6a) is by definition; (6b) and (6f) are simply (5.d) and (5.f), respectively; the left implication of (6c) and (6g) are simply (5.e) and (5.g), respectively, and the right implication is from (6b) and (6f) together with the fact that $LTE$ and $GTE$ are monotone (increasing/decreasing, respectively) functions in $v$; (6d) and (6h) are the contrapositive of (6c) and (6g), respectively; and (6e) is because $v_i \leq v$ (by (6c)) and also $v_i \geq v$ by (6h) together with (6a) (and similarly (6i) follows from (6a), (6g), and (6d)).

The correctness of the above protocols will rely on the “alignment” property claimed in the Secure Symmetric Merge rubric above. Formally:
Lemma D.1. Let $L_1$ and $L_2$ denote two (sorted) lists of size $n$, and let $M_{1,k}$ and $M_{2,k}$ denote their $k$ medians (as per (1)). Let $L_1' = L_1 \cup M_{2,k}$ denote the list (of size $2n$) resulting from merging $M_{2,k}$ with $L_1$, with each element in $M_{2,k}$ duplicated $n/k$ times in $L_1'$. Let $M_{2k}$ denote the $2k$ medians of $L_1'$. Then the $2k$ medians of $L_1'$ are exactly the (merged) $k$ medians of $L_1$ and $L_2$:

$$M_{2k} = M_{1,k} \bigcup M_{2,k}$$

Proof. Denote the lists of $k$ medians as:

$$M_{1,k} = \{u_1, u_2, \ldots, u_k\}$$

$$M_{2,k} = \{v_1, v_2, \ldots, v_k\}$$

Let $M_{2k} = M_{1,k} \bigcup M_{2,k}$, and denote $M_{2k}$ and $M_{2k}'$ as:

$$M_{2k} = \{w_1, w_2, \ldots, w_{2k}\}$$

$$M_{2k}' = \{z_1, z_2, \ldots, z_{2k}\}$$

Then the claim is that $M_{2k} = M_{2k}'$, which we argue by demonstrating that $z_i = w_i$ for all $1 \leq i \leq 2k$. Notice that $|L_1'| = 2n$ and $|M_{2k}| = 2k$, and by definition $M_{2k}'$ evenly partitions $L_1'$ into $2k$ blocks, with each block of size $n/k$. Furthermore, notice that $L_1'$ is the merge of two lists:

- $n$ elements are $L_1$
- $n$ elements are the $k$ elements of $M_{2,k}$, each duplicated $n/k$ times.

In particular, if we write $L_2' := \bigcup \frac{n}{k} M_{2,k}$ (i.e. $L_2'$ is $M_{2,k}$ duplicated $n/k$ times), then observe that the $k$ medians of $L_2'$ are exactly the same as the $k$ medians of $L_2$, namely $M_{2,k}$. Therefore, if we write (as multi-sets): $L_1' = L_1 \cup L_2'$, then (as per (5)):

For any $u_i \in M_{1,k}$: There are at least $i \cdot \left(\frac{n}{k}\right)$ items in $L_1'$, from $L_1$, that are $\leq u_i$.

There are at least $1 + (k - i) \cdot \left(\frac{n}{k}\right)$ items in $L_1'$, from $L_1$, that are $\geq u_i$.

(7)

For any $v_i \in M_{2,k}$: There are at least $i \cdot \left(\frac{n}{k}\right)$ items in $L_1'$, from $L_2'$, that are $\leq v_i$.

There are at least $(1 + k - i) \cdot \left(\frac{n}{k}\right)$ items in $L_1'$, from $L_2'$, that are $\geq v_i$.

Notice that the last statement in (7) (unlike the second statement) does not come directly from (5.e). In particular, it relies on the fact that the medians of $L_2'$ are exactly described by $M_{2,k}$ (the medians of $L_2$), and each such median appears a total of $n/k$ times in $L_2'$. This is why the ‘1’ appears inside the parentheses (as a multiplicative factor of the $n/k$) as opposed to outside (as an additive constant). We conclude the proof by showing that $z_i = w_i$ for all $1 \leq i \leq 2k$ by showing both inequalities:

$z_i \geq w_i$. By (6d), this will follow if we can show that there are at least $i \cdot \frac{2n}{k}$ values in $L_1'$ that are less than or equal to $w_i$. Let $x = x_i$ denote the maximal index such that $u_x \in M_{1,k}$ appears among the first $i$ coordinates of $M_{2k}$. Similarly, let $y = y_i$ denote the the maximal index such that $v_y \in M_{2,k}$ appears among the first $i$ coordinates of $M_{2k}$. By definition of $M_{2k}$ (as the merge of $M_{1,k}$ and $M_{2,k}$), we have that $x_i + y_i = i$. And then by (7) above, we have that there are at least $x_i \cdot \frac{n}{k}$ elements in $L_1$ that are less than or equal to $u_x$, and since $u_x$ is in the first $i$ coordinates of $M_{2k}$, we have that $u_x \leq w_i$, and hence (by monotonicity of LTE) there are at least $x_i \cdot \frac{n}{k}$ elements in $L_1$ that are less than or equal to $w_i$. Similarly, there are at least $y_i \cdot \frac{n}{k}$ elements in $L_2'$ that are less than or equal to $w_i$. Consequently, there are at least $(x_i + y_i) \cdot \frac{n}{k} = i \cdot \frac{n}{k}$ values in $L_1'$ that are less than or equal to $w_i$, as required.

$z_i \leq w_i$. By (6h), this will follow if we can show that there are at least $1 + (2k - i) \cdot \frac{2n}{k}$ values in $L_1'$ that are greater than or equal to $w_i$. Let $x = x_i$ denote the minimal index such that $u_x \in M_{1,k}$ appears at or after coordinate $i$ of $M_{2k}$ (or define $x_i = k + 1$ if none of the values
in $M_{1,k}$ appear at or after coordinate $i$ of $M_{2k}$). Similarly, let $y = y_i$ denote the minimal index such that $v_y \in M_{2,k}$ appears at or after coordinate $i$ of $M_{2k}$ (or define $y_i = k + 1$ if none of the values in $M_{2,k}$ appear at or after coordinate $i$ of $M_{2k}$). By definition of $M_{2k}$ (as the merge of $M_{1,k}$ and $M_{2,k}$), we have that $x_i + y_i = i + 1$. By (7) above, we have that there are at least $\max(0, 1 + (k - x_i) \cdot \frac{n}{k})$ elements in $L_1$ that are greater than or equal to $u_x$, and since $u_x$ has index at least $i$ in $M_{2k}$, we have that $u_x \geq w_i$, and hence (by monotonicity of $\text{GET}$) there are at least $\max(0, 1 + (k - x_i) \cdot \frac{n}{k})$ elements in $L_1$ that are greater than or equal to $w_i$. Similarly, there are at least $(1 + k - y_i) \cdot \frac{n}{k}$ elements in $L'_2$ that are greater than or equal to $w_i$. Consequently, there are at least $1 + (1 + 2k - (x_i + y_i)) \cdot \frac{n}{k} = 1 + (2k - i) \cdot \frac{n}{k}$ values in $L'_1$ that are greater than or equal to $w_i$, as required.

References


