Can We Overcome the $n \log n$ Barrier for Oblivious Sorting? [*]

Wei-Kai Lin [†] Elaine Shi [‡] Tiancheng Xie [§]

Abstract

It is well-known that non-comparison-based techniques can allow us to sort n elements in $o(n \log n)$ time on a Random-Access Machine (RAM). On the other hand, it is a long-standing open question whether (non-comparison-based) circuits can sort n elements from the domain $[1..2^k]$ with $o(kn \log n)$ boolean gates. We consider weakened forms of this question: first, we consider a restricted class of sorting where the number of distinct keys is much smaller than the input length; and second, we explore Oblivious RAMs and probabilistic circuit families, i.e., computational models that are somewhat more powerful than circuits but much weaker than RAM. We show that Oblivious RAMs and probabilistic circuit families can sort $o(\log n)$-bit keys in $o(n \log n)$ time or $o(kn \log n)$ circuit complexity. Our algorithms work in the *indivisible model*, i.e., not only can they sort an array of numerical keys — if each key additionally carries an opaque ball, our algorithms can also move the balls into the correct order. We further show that in such an indivisible model, it is impossible to sort $\Omega(\log n)$-bit keys in $o(n \log n)$ time, and thus the $o(\log n)$-bit-key assumption is necessary for overcoming the $n \log n$ barrier.

Finally, after optimizing the IO efficiency, we show that even the 1-bit special case can solve open questions: our oblivious algorithms solve tight compaction and selection with optimal IO efficiency for the first time.

[*]This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a DARPA Safeware grant (subcontractor under IBM), a Sloan Fellowship, Google Faculty Research Awards, a Baidu Research Award, and a VMWare Research Award.

[†]Cornell University.

[‡]Cornell University.

[§]Shanghai Jiao Tong University, work done while visiting Cornell.

Contents

1 Introduction **1**
1.1 Main Results . . . 2
1.2 IO Efficiency on Oblivious RAMs and Applications to Open Problems . . . 4
1.3 Related Works . . . 5

2 Technical Roadmap **7**
2.1 Partition 1-Bit Keys . . . 7
2.2 Sorting Longer Keys: A Simple Algorithm . . . 9
2.3 A Better Recurrence . . . 10
2.4 Extension: Large Key Space but Few Distinct Keys . . . 11
2.5 IO Efficiency in the Cache-Agnostic Model . . . 13
2.6 Lower Bounds for Oblivious Sorting . . . 13
2.7 Comparison with Known ORAM Lower Bounds . . . 14

3 Definitions and Preliminaries **15**
3.1 Oblivious Algorithms on a RAM . . . 15
3.2 External-Memory and Cache-Agnostic Algorithms . . . 16
3.3 Building Blocks . . . 17
3.3.1 Cache-Agnostic Algorithms . . . 17
3.3.2 Oblivious Sorting . . . 17

4 Partitioning 1-Bit Keys **18**
4.1 Intuition . . . 18
4.2 New Building Blocks . . . 19
4.3 Algorithm for Partitioning 1-Bit Keys . . . 19
4.4 Analysis . . . 19

5 Sorting Short Keys **21**
5.1 Intuition . . . 21
5.2 Warmup Scheme . . . 22
5.3 Improved Recurrence . . . 22

6 Open Questions **24**

A Additional Preliminaries **29**
A.1 Probabilistic Circuits for Sorting in the Indivisible Model . . . 29
A.2 Standard Assumptions of External-Memory Model . . . 30

B Oblivious Sorting Lower Bounds in the Indivisible Model **30**
B.1 Preliminaries on Routing Graph Complexity . . . 31
B.2 Limits of Oblivious Sorting in the Indivisible Model . . . 31
B.3 Limits of Stability . . . 33
B.4 Implications for Probabilistic Sorting Circuits in the Indivisible Model. . . . 34

C New Building Blocks **34**
C.1 Deterministic Partitioning 34
C.1.1 Intuition 34
C.1.2 Algorithm 35
C.1.3 Analysis 37
C.2 Move Stripe to Working Buffer 37

D Sorting Arbitrary-Length but Few Distinct Keys **38**
D.1 Counting Few Number of Distinct Keys 38
D.2 Counting Distinct Keys 38
D.2.1 Preliminaries 39
D.2.2 Algorithm for Estimating Number of Distinct Keys 40
D.2.3 Analysis 40
D.3 Sorting Arbitrary-Length but Few Distinct Keys 42

E Applications to Other Open Problems **45**
E.1 Tight Compaction 45
E.2 Selection 45
E.3 Additional Applications 47

F Deferred Proofs **47**
F.1 Proof of Theorem 5.1 47
F.2 Proof of Theorem 5.2 50

1 Introduction

Sorting has always been one of the most central abstractions in computing. Traditionally, sorting was intensively studied in both the circuit and Random Access Machine (RAM) models of computation. It is well-understood that there exist comparison-based sorting circuits of $O(n \log n)$ size and $O(\log n)$ depth where n denotes the number of input elements [2, 36]. Comparison-based sorting, in either the circuit or the RAM model, has a well-known $\Omega(n \log n)$ lower bound in circuit size or runtime [44]. This lower bound can be circumvented on a RAM using non-comparison-based techniques, and (almost) linear-time sorting is possible [5, 40, 41, 43, 68].

In this paper, we consider a specific version of sorting referred to as sorting in the *indivisible* model[1]. Imagine that initially we have n balls each tagged with a key from some domain $[1..K]$ of size $K = 2^k$. Our goal is to sort the balls based on the relative order of their keys. Informally speaking, if an algorithm (or a circuit) calls only comparator operations on the keys, we say that the algorithm (or circuit) is *comparison-based*; else if the algorithm (or circuit) performs arbitrary boolean operations on the keys, it is said to be *non-comparison-based* (see Section 1.3 for further clarifications). Regardless of which case, we always assume that the balls are *opaque and indivisible* — they can be moved around atomically, but cannot be computed upon[2].

A long-standing open problem is whether there exist (non-comparison-based) circuits of $o(kn \log n)$-size capable of sorting n inputs chosen from the domain $[1..2^k]$ where k is the word-length [14]. In this paper, we explore weakened forms of this question: first, we consider restricted classes of sorting where the number of distinct keys is much smaller than the input length; and second, instead of circuits, we consider Oblivious RAMs and probabilistic circuit families as the computational model both of which are somewhat more powerful than circuits but much weaker than Random Access Machine (RAM). We now elaborate on these two computation models.

- **Oblivious RAMs.** An algorithm in the Oblivious RAM model [32, 33, 65] (also referred to as an oblivious algorithm[3]) is one that executes on a standard Random-Access Machine (RAM), but we additionally require that for any two inputs of the same length, the CPU's memory access patterns be *statistically* close. In other words, by observing the memory access patterns, an adversary learns almost no information about the secret input. We allow the CPU to have access to private random bits that are unobservable by the adversary. Throughout this paper we consider RAMs with $O(1)$ CPU registers.
- **Probabilistic circuit family.** A probabilistic circuit family $\mathcal{C}$ is said to sort n balls whose keys are from the domain $[K]$ iff given any input key assignment from $[K]^n$, a randomly chosen circuit from $\mathcal{C}$ will correctly sort this input with overwhelming probability. This means that the input key assignment may be chosen adversarially but the adversary cannot examine the circuit that is randomly sampled prior to choosing the input. In this paper, we assume that each circuit in the family has two types of gates: 1) *boolean gates* with constant fan-in and fan-out; and 2) *selector gates* that takes a single bit and two balls, and selects one of the two balls.

Despite the rich results on (almost) linear-time sorting on (non-oblivious) RAMs [5, 40, 41, 43, 68], in the aforementioned Oblivious RAM or probabilistic circuit family models, the most asymptotically efficient sorting algorithm remains comparator-based sorting networks, e.g., AKS [2] or Zigzag sort [36] can sort n input elements in $O(n \log n)$ comparator operations. To the best of

[1] In some earlier works especially in the Oblivious RAM literature [14, 33, 46], the indivisible model is also referred to as the "balls-and-bins" model.

[2] Note that the indivisible model precludes certain types of algorithms, e.g., if the key is only 1-bit long, in the non-indivisible model, an algorithm could simply count the number of 0s and write out an answer; but this algorithm fails to move the balls into sorted order.

[3] In some literature, this notion is referred to as "data-oblivious" algorithms, but we use the word "oblivious".

our knowledge, it is unknown whether non-comparison-based techniques can asymptotically speed up sorting in these two models.

1.1 Main Results

We give a two-sided answer to the earlier questions. In this section we summarize our main results, stated using runtime (or circuit complexity) as a metric. Later in Section 1.2 we shall present additional results regarding metrics beyond the runtime — specifically we will describe how to optimize our algorithms' IO efficiency (i.e., the number of transfers between cache and memory) when executed on an Oblivious RAM.

Lower bounds for oblivious sorting. On the pessimistic side, we show that any oblivious algorithm that sorts k-bit keys such that $k = \Omega(\log n)$ in the indivisible model must incur at least $\Omega(n \log n)$ runtime — this lower bound holds even when the algorithm is non-comparison-based and moreover, even when allowing the algorithm to err with $O(1)$ probability on any input. Recall that the runtime in RAM model is the number of word operations. Our lower bound proof has a direct interpretation in the probabilistic circuit model mentioned above: we show that any probabilistic circuit family that sorts n balls each with an k-bit key must consume at least $\Omega(n \log n)$ selector-gates (for moving opaque balls around). Our lower bound results are stated informally below.

Theorem 1.1. (Informal: limits of oblivious sorting in the "indivisible" model). *For all $k \in \mathbb{N}$, any (possibly randomized) oblivious algorithm that sorts n balls each with an k-bit key must incur at least $\Omega(n \cdot \min(k, \log n))$ runtime, even when the algorithm is allowed $O(1)$ correctness failure on any input.*

Corollary 1.1. (Informal: limits of sorting with probabilistic circuits in the "indivisible" model). *For all $k \in \mathbb{N}$, any probabilistic circuit family that sorts n balls each with an k-bit key must consume at least $\Omega(n \cdot \min(k, \log n))$ selector-gates, even when we allow $O(1)$ correctness failure on any input.*

As mentioned in Section 2.7, our work demonstrates a new paradigm for lower bounding the complexity of oblivious algorithms — to the best of our knowledge, this is the only new paradigm since the well-known ORAM lower bound proof by Goldreich and Ostrovsky [32, 33][4].

Oblivious sorting in $o(n \log n)$ runtime for $o(\log n)$-bit keys. At the first sight, it might seem that we are at a dead end due to the aforementioned lower bounds imply that sorting $(\log n)$-bit keys takes time $\Omega(n \log n)$. However, we show that for smaller key sizes, non-comparison-based techniques can indeed help us defeat the $\Omega(n \log n)$ barrier for oblivious sorting. We prove the following result.

Theorem 1.2. (Informal: oblivious sorting for $o(\log n)$-bit keys). *There is a randomized oblivious algorithm which, except with negligible probability[5], correctly sorts n balls each with a key from the domain $[1..2^k]$ in running time $O(kn\frac{\log \log n}{\log k})$.*

As a special case, this implies that oblivious sorting of n balls each with a constant-sized key can be accomplished in $O(n \log \log n)$ time. Furthermore, when the key is $o(\log n)$ bits long (note that this means there must be many duplicate keys), our algorithm completes in $o(n \log n)$ time.

[4] Subsequently, Larsen and Nielsen [46] shows another ORAM lower bound using a novel technique "oblivious cell probe model".

[5] For simplicity, in the introduction, we assume that negligible failure is expressed in terms of n, i.e., the input length; although later in the paper we will use λ as the security parameter to explicitly distinguish from the input size.

To the best of our knowledge, Mitchell and Zimmerman [56] is the only known result in this vein[6] — they show how to obliviously sort 1-bit keys in $O(n \log \log n)$ runtime. Unfortunately there does not seem to be any straightforward way to extend their algorithm to even 2-bit keys. We also note that our result is tight in the indivisible model: in light of our lower bound, if the keys were $c \log n$ bits long for any arbitrarily small constant c, $\Omega(n \log n)$ runtime is a necessary price to pay for obliviousness. Moreover, for all k asymptotically smaller than $\log n$, our upper bound almost matches our lower bound, which differ only by a factor of $O(\log \log n)$.

Although in general oblivious algorithms may not have efficient circuit implementations (e.g., if they make data-dependent memory accesses [32,33,65,67,71]), all oblivious algorithms presented in this paper indeed access memory only in data-independent manners and thus can be easily implemented with probabilistic circuit families. We thus have the following corollary.

Corollary 1.2. (Informal: sorting with probabilistic circuits for $o(\log n)$-bit keys). *There exists a probabilistic circuit family which, except with negligible probability, correctly sorts n balls each with a key from $[1..2^k]$ consuming only $O(k^2 n \log \log n / \log k)$ boolean gates and $O(kn \log \log n / \log k)$ selector gates.*

Extension: sorting few distinct keys from a large space. Our results stated earlier can sort keys chosen from a small space in $o(n \log n)$ time. Note that one immediate implication that the keys are chosen from a small space is that the number of distinct keys is small. Thus a very natural question is whether our results would extend to the case when the keys are chosen from a large space but the number of distinct keys is small. We answer this question in the affirmative as informally stated in the following corollary.

Corollary 1.3. *For any positive $\alpha(n) := \omega(1)$, there is a randomized oblivious algorithm which, except with $n^{-\Omega(\alpha(n))}$ probability, correctly sorts n balls tagged with at most 2^k distinct keys in running time $O(n\alpha L + n\alpha^2 \log \log n + kLn \log \log n / \log k)$ where L denotes the number of words needed for storing each key.*

To achieve the above corollary, we devise efficient oblivious algorithms for estimating the number of distinct keys given an input array — these new building blocks might be of independent interest.

Limits of stability. Our sorting algorithm is *not* stable but as we show, this is in fact inherent. Recall that in the context of sorting, stability means that for any two balls whose keys are equal, the ordering of the balls in the output must agree with their relative order in the input. We prove a lower bound showing that even for 1-bit keys, *stable* oblivious sorting is impossible in $o(n \log n)$ time in the indivisible model — in Section 2.7, we argue that the following lower bound can be viewed as a strengthening of Goldreich and Ostrovsky's logarithmic ORAM lower bound.

Theorem 1.3 (Limits on stable oblivious sort). *Any (possibly randomized) oblivious algorithm that stably sorts n balls each with a 1-bit key must incur at least $\Omega(n \log n)$ runtime, even when we allow the algorithm to err with $O(1)$ probability on any input.*

Similar as before, our lower bound proof has a natural interpretation in a probabilistic circuit model, giving rise to the following corollary.

Corollary 1.4. (Limits on stable sort with probabilistic circuits). *Any probabilistic circuit family that stably sorts n balls each with a 1-bit key must incur at least $\Omega(n \log n)$ selector gates, even when we allow $O(1)$ correctness failure on any input.*

[6] Although Leighton et al.'s $O(n \log \log n)$ selection network [47] may seem close in nature, their definition of selection does not imply sorting. In fact, their construction is comparator-based and clearly cannot solve the 1-bit sorting problem or else it would violate the well-known 0-1 principle — see Section 1.3 for further discussions and the subsequent work of Peserico [59].

1.2 IO Efficiency on Oblivious RAMs and Applications to Open Problems

So far, we have solely focused on the algorithm's runtime (or circuit size). When oblivious sorting is implemented on a RAM, not only do we care about its runtime, IO performance is also a particularly important metric — this is exactly why there is a long line of research on external-memory and *cache-agnostic*[7] algorithms [25, 31, 35, 37].

We devise additional techniques to make our algorithms IO-efficient in the cache-agnostic model [25, 31]. The cache-agnostic model is an elegant and well-accepted notion first proposed by Frigo et al. [31] requiring that 1) an algorithm need not know the cache parameters and thus a single algorithm can readily execute on any target architecture without parameter tuning; and 2) when executed on any multi-level memory hierarchy, the algorithm minimizes cache misses on every level of the memory hierarchy simultaneously (including between cache and memory, between memory and disk, and between client and cloud). In this overview section, we present our IO efficiency results for the 1-bit special case, deferring the general statement of results for longer keys to Section 5. Interestingly, even *the 1-bit special case allows us to solve open questions* raised by prior work [35]. In the following, using standard notations adopted by the algorithms literature, we denote cache size as M and cache-line size as B (counted in the number of memory words).

Theorem 1.4. (Informal: IO-optimal oblivious sort for constant-sized keys). *There exists an oblivious and cache-agnostic algorithm that can sort $O(1)$-bit keys (except with negligible probability) in only $O(n/B)$ IOs (i.e., asymptotically optimal IO) under standard cache assumptions (i.e., tall cache and wide cache-line).*

Sorting 1-bit keys in optimal IO immediately implies *tight compaction* and *selection* in optimal IO — both are fundamentally important algorithmic abstractions that have been extensively studied in the algorithms literature [13, 35, 47, 56]. We now explain why our IO-efficient construction for obliviously sorting 1-bit keys solves open algorithmic challenges.

Tight compaction. Tight compaction is the following problem: given an input array I containing n real or dummy elements, output an array of equal size such that all real elements in I reside in the front of the array and remainder of the output array is padded with dummies. Earlier, Mitchell and Zimmerman [56] provides an oblivious tight compaction algorithm that runs in $O(n \log \log n)$ time, but takes $O(n)$ IO-cost (see Section 1.3 for more explanations) — as mentioned the optimal IO-cost would have been $O(n/B)$. In an elegant work by Goodrich [35], he phrased the following open challenge:

Can we achieve oblivious tight compaction in asymptotically optimal IO[8]*?*

As an immediate implication of our main theorem, we answer Goodrich's question in the affirmative — not only so, our algorithm is *cache-agnostic*. We stress that previously, it was *not even known how to design a cache-aware*[9] *oblivious algorithm* that achieves tight compaction in optimal IO.

Selection. Selection is the following problem: given an input array containing n opaque balls each associated with a key, output the $k \leq n$ smallest balls (tagged with their respective keys). Note that the parameters k and n are publicly known; and obliviousness requires that the algorithm does not

[7] In the standard algorithms literature this was typically referred to as the *cache-oblivious* model [25, 31], but we use the term cache-agnostic instead to avoid overloading the word "oblivious".

[8] Leighton et al.'s algorithm [47] does not achieve tight compaction for a similar reason why they cannot solve 1-bit-key sorting. See Section 1.3 for further discussions.

[9] We say that an algorithm is *cache-aware* if the algorithm takes as input the cache parameters.

reveal the keys (or balls) of the input array. Using small-key sorting as a building block, we show how to achieve oblivious selection in a cache-agnostic model in $O(n/B)$ IO-cost and $O(n \log\log n)$ runtime.

In comparison, the previous best known results for selection are the following: First, both Mitchell and Zimmerman [56] and Leighton et al. [47] achieve $O(n \log\log n)$ runtime but both approaches incur $O(n)$ IO-cost (see Section 1.3 for more explanations). Previously the most IO-efficient algorithm for oblivious selection in the *cache-agnostic* model is simply by applying Chan et al. [20] cache-agnostic oblivious sort algorithm which would require $\Omega((n/B)\log_{M/B}(n/B))$ IO-cost and $O(n \log n \log\log n)$ runtime. In the *cache-aware* model, a partial result exists, again by Goodrich [35], showing how to obliviously select the k-th smallest element in linear time and optimal IO. Unfortunately their technique is tailored for selection of *only* the k-th smallest element; directly using their algorithm to select *all* k smallest elements would incur another k blowup in performance; thus if $k = \Omega(\log n)$ the runtime would be $\Omega(n \log n)$.

Corollary 1.5. (Informal: IO-optimal tight compaction and selection). *There is a cache-agnostic, oblivious algorithm for tight compaction (or selection resp.) that completes in $O(n \log n \log n)$ runtime and with $O(n/B)$ IO-cost under standard cache assumptions (i.e., tall cache and wide cache-line).*

1.3 Related Works

Sorting. Sorting is a classical algorithmic abstraction, and the study of efficient sorting algorithms in both the circuit [2, 9, 36] and the Random Access Machine (RAM) [5, 40, 41, 43, 68] models of computation has been long-standing and extensive.

It is well-understood that there exist *deterministic*, comparator-based sorting networks that sort n elements consuming only $O(n \log n)$ comparators [2,36]. While such circuit-based techniques are perfectly oblivious, it is well-known that $\Omega(n \log n)$ comparison operations are necessary for any comparison-based sorting algorithm (not only for circuits but also for RAMs) [44]. On a RAM, it is also well-known that non-comparison-based techniques can sort n elements in $o(n \log n)$-time (e.g., Radix sort, counting sort, and others [5,40,41,43,68]). A subset (but not all) of these $o(n \log n)$-time algorithms [4,69] additionally allow each key to carry an opaque ball and the sorting algorithm will rearrange the balls according to the relative order of their keys — in our paper such an algorithm is said to support sorting in the indivisible model (i.e., balls cannot be split nor computed upon). To the best of our knowledge, most of the classical line of work on $o(n \log n)$-time non-comparator-based sorting [5, 40, 41, 43, 68] crucially relies on the ability to make data-dependent memory accesses.

Clarifications on the 0-1 principle. It is important to note the 0-1 principle for comparison-based sorting [34,36,44]: any (possibly probabilistic), comparison-based sorting algorithm (in either the circuit model or RAM model) that can correctly sort any input array consisting only of 0s and 1s (with high probability) can correctly sort any input array containing any set of comparable items (with high probability). One implication of this principle is that there cannot be any comparison-based algorithm (even not requiring obliviousness) that sorts even 0-1 arrays in $o(n \log n)$ time — thus a result of our nature must be non-comparison-based.

We stress that the 0-1 principle is applicable even when the algorithm is allowed to perform *arbitrary* computation *does not dependent* on the value of the keys — however, if the algorithm performs arbitrary computations that are dependent on the value of the keys, then the 0-1 principle is no longer applicable[10]. The algorithms described in our paper indeed require computations

[10] For simplicity, we consider only the described comparator model. The 0-1 principle is applicable on any Min-Max computation, which is a strictly stronger model compared to the comparator model [48].

dependent on the outcomes of the comparators, e.g., counting.

In earlier works and resources online, any sorting algorithm that invokes only comparison operations on the keys is often said to be *comparison-based* sorting — such a definition does not clearly articulate whether additional arbitrary computations are allowed. In this paper, we simply use the 0-1 principle as a criterion for classifying comparison-based vs. non-comparison-based sorting.

Oblivious RAM and oblivious algorithms. In the ground-breaking work by Goldreich and Ostrovsky [32, 33], they propose a new model of computation henceforth referred to as Oblivious RAM (ORAM) [32, 33, 65]. Specifically, an Oblivious RAM is a Random-Access Machine (RAM) in which the CPU's memory access patterns are computationally or statistically independent of the data that is input to the computation — to achieve this, we assume that the CPU can obtain and make use of private randomness.

Oblivious RAM (ORAM) is a rather intriguing model of computation. On one hand, we know that allowing randomness, any RAM program can be simulated with an ORAM with only $O(\log^2 n)$ blowup in runtime [21,71]. On the other hand, this does not mean that every algorithm must incur such a poly-logarithmic blowup when obliviously simulated — in fact, a recent line of research showed that a broad class of algorithms and data structures can be computed obliviously while incurring asymptotically smaller overhead than generic ORAM simulation [12,26,32,33,35,38,50,57].

Goldreich and Ostrovsky [32,33], and the more recent work by Larsen and Nielsen [46] (which comes subsequently to this work) prove that any ORAM algorithm must incur logarithmic blowup in comparison with the non-secure baseline. We give more detailed explanations of these known lower bounds and compare with our new oblivious algorithm lower bounds in Section 2.7.

IO efficiency of oblivious algorithms. Several works have considered the IO efficiency of oblivious algorithms in either a theoretical setting [7,20,35,37], or in practical implementations [29, 63, 64, 66, 74]. With the exception of Chan et al. [20], almost all known works consider the cache-aware setting, that is, the algorithm or implementation must be aware of the cache's parameters, e.g., the cache-line (or block) size, and the total size of the cache.

Goodrich's elegant work [35] constructs IO-optimal algorithms for compaction, selection, and sorting, but his techniques do not extend in any straightforward fashion to the cache-oblivious setting. Specifically, his algorithm relies on packing elements into cache-lines such that the algorithm can operate on cache-lines without incurring IO — thus the very core idea behind his algorithms relies on knowing explicitly the cache-line size and cache size. Leighton's oblivious selection algorithm [47] and Mitchell and Zimmerman's oblivious 1-bit-key sorting algorithm [56] both take $O(n \log \log n)$ runtime but consumes $O(n)$ IO-cost. There does not seem to be any straightforward method to make their algorithms more IO-efficient (recall that the optimal IO would have been $O(n/B)$). In particular, Leighton's algorithm [47] employs AKS sorting on polylogarithmically sized buckets; and additionally requires a global (non-oblivious) random permutation of the input array — both these steps do not exploit data locality and thus are not IO-efficient. Similarly, Mitchell and Zimmerman [56] requires randomly accessing a constant fraction of the input array which is expensive in IO.

Circuit complexity for sorting? It remains an open question whether there is a $o(w \cdot n \log n)$-sized circuit for sorting n words each w-bits long. To the best of our knowledge, no upper bounds or lower bounds are known regarding the (non-comparator-based) circuit complexity for sorting. Perhaps rather surprisingly, there does not even seem to be a partial answer for 1-bit words — note that the probabilistic selection circuit construction by Leighton et al. [47] is comparator-based and without introducing non-comparison-based techniques, their construction does not imply 1-bit sorting in $o(n \log n)$ time in any immediate, blackbox manner.

Our work may be regarded as some partial progress at understanding the (non-comparator-based) circuit complexity for sorting — however, our results apply only for a probabilistic circuit family model, where we consider a family of circuits and we sample a random circuit to give to an input (that is chosen by a possibly unbounded adversary who has not observed the circuit).

Estimating the number of distinct elements. There is a long line of research on algorithms that estimate the number of distinct elements in a *data stream* [28,42]. This line of work culminated in Kane et al. [42] who described a streaming algorithm that is tight in time and space — although it is not difficult to make their algorithm oblivious, to the best of our knowledge, their failure probability of correctness is $o(1)$ whereas we require negligibly small failure. In Appendix D, we describe a novel, almost linear-time oblivious algorithm for estimating the number of distinct elements in an array — in comparison, our algorithm need not be streaming in nature but must be oblivious. Thus our result here is related but incomparable to those achieved in the streaming algorithms literature.

Subsequent works. In a work by Chan et al. [18] that was recently released[11], the authors define the notion of (ϵ, δ)-differential obliviousness (i.e., (ϵ, δ)-differential privacy for access patterns). In the sorting context, (ϵ, δ)-differential obliviousness is strictly weaker than obliviousness. They show that for differential obliviousness one can sort k-bit keys in $O(kN(\log k + \log \log N))$ time for $\epsilon = \Theta(1)$ and δ being a suitable negligible function, and moreover with *stability*. As we show in this paper, stable oblivious sorting (in the indivisible model) is impossible without at least $\Omega(n \log n)$ runtime even for the 1-bit case. Thus Chan et al. [18]'s results show that with appropriate relaxations in privacy, one can overcome this stability lower bound. The new techniques for proving lower bounds described in this paper are also extended by Chan et al. [18] to prove more lower bounds for oblivious and differentially oblivious algorithms.

Peserico [59] shows a deterministic, oblivious algorithm such that performs tight compaction in linear time. It is not too difficult to extend his result to allow sorting of constant-sized keys (in the indivisible model) in linear time as well. Therefore, for constant-sized keys, Peserico's algorithm asymptotically improved our results. Interestingly, the optimal oblivious tight compaction of Peserico turned out to be a key building block in the construction of optimal ORAM in Asharov et al. [8].

2 Technical Roadmap

In this section, we present an informal technical roadmap on how we achieve the claimed results. For simplicity, we shall first explain our results focusing on only the runtime metric: We start from sorting, or partitioning, 1-bit keys in Section 2.1. Then, we sort "longer" keys by applying the 1-bit technique recursively in Section 2.2, improve the recursion in Section 2.3, and then extend it to even larger key space using hashing and distinct estimation in Section 2.4. Additional techniques are required to achieve IO efficiency and we defer the explanation of these techniques to Section 2.5. The lower bound proof is sketched and discussed in Section 2.6 and 2.7.

2.1 Partition 1-Bit Keys

We first show how to obliviously partition 1-bit keys: given an input array consisting n balls each with a 1-bit key, we want to partition the array such that 0-balls appear before the 1-balls.

1-bit sorting is not selection. Let us first consider the probabilistic selection circuit construction by Leighton et al. [47]. Specifically, Leighton et al. construct a *comparator-based* circuit of

[11] Although their work was released earlier, chronologically it actually happened subsequently to the writing of this paper.

$O(n \log\log n)$ size that can select and output the m smallest elements with very high probability when given an input array that has been randomly permuted. Interestingly, although at first sight, sorting 1-bit keys and selection seem like very similar problems, we stress that in fact our problem formulation (i.e., sorting 1-bit keys) is stronger than that of Leighton et al.'s [47] (i.e., selecting m smallest elements). We stress that there is no straightforward way (i.e., without relying on some kind of non-comparison-based techniques) of applying Leighton et al. [47]'s (comparator-based) selection network in a blackbox fashion to sort 0-1 sequences in $o(n \log n)$ time, or else it would clearly violate the 0-1 principle. (For example, it is not difficult to realize that the naïve approach of first selecting the smaller half and then selecting the larger half does not work — see remark below.)

Remark 1. *Interestingly, if one could reveal the number of* 0*s and* 1*s in the input, it would be trivial to rely on a selection network to realize* 1*-bit sorting; however, in our formulation, this count must be hidden.*

Our algorithm for partitioning 1-bit keys. Instead, we rely on the core ideas of Leighton et al. [47] in a non-blackbox fashion. Our 1-bit partitioning algorithm works as follows:

Partition(A):

1. Randomly permute the input elements in A using a linear-time implementation of the Fisher-Yates shuffle [27]. This random permutation is used only for load balancing and measure concentration and thus the permutation need not be secret.
2. Divide the permuted input array into bins of $Z = \log^6 \lambda$ in size where λ is the security parameter (i.e., we aim to achieve negligible in λ correctness failure).
3. Apply a sorting network such as Zigzag sort [36] to sort each bin. When each bin becomes sorted, we express all bins as a short-and-fat matrix denoted A' where each column represents a bin.
4. Our crucial observation is the following: in this matrix A', there must exist a set of at most $\log^4 \lambda$ consecutive rows henceforth called the *mixed stripe*, such that all elements above the mixed stripe (if any) are 0s, and all elements below (if any) are 1s.
5. Now, in one scan of the rows, we can identify the location of the mixed stripe.
6. Now, in one scan of the rows, we can copy the mixed stripe to a working buffer *without revealing where the mixed stripe resides* (see the simple Algorithm 7) for details on how to achieve this). We then call Zigzag sort to sort the working buffer. Finally, using oblivious sorting to cyclically shift the working buffer, combined with another scan of the rows, we can copy this working buffer back to where the mixed stripe was without revealing the location of the mixed stripe (see Section 4 for details).

The Partition algorithm is non-comparison-based. We stress that the above algorithm is non-comparison-based due to Steps 5 and 6. In these two steps, the algorithm makes use of the location of the mixed stripe, which is a variable that depends on the number of 0s and 1s in the input sequence. More specifically, if one were to implement as a circuit the oblivious procedure for copying the mixed stripe to the working buffer, such a circuit would require gates that take this mixed stripe location as input. As we clarified in Section 1.3 (the Related Work section), for such circuits the 0-1 principle for comparison-based sorting is not applicable (and thus such algorithms should be regarded as non-comparison-based).

2.2 Sorting Longer Keys: A Simple Algorithm

Our next step is to consider how to leverage our 1-bit partitioning building block to sort longer keys. For clarity, we will first describe a conceptually simple version of this reduction — this simple version already allows us to sort $o(\log n/\log\log n)$-bit keys in $o(n\log n)$ time. Later in Section 2.3, we describe how to reparametrize our recursion such that we can sort $o(\log n)$-bit keys in $o(n\log n)$ time — this is tight in the "indivisible" model in light of our lower bound in Theorem 1.3.

Consider an input array containing n balls each with a key from the domain $[1..K]$. Henceforth let $\mathsf{Sort}^K(A)$ denote an instance of our oblivious sorting algorithm capable of sorting keys from a domain $[a+1..a+K]$ of size K for some integer a when given an input array A. We assume that the input array has already been randomly permuted — if not, we can always permute it at random in linear time, e.g., using an efficient implementation of the Fisher-Yates shuffle [27]. As we shall see, this random permutation is used only for load balancing and thus the permutation need not be secret.

The algorithm. The algorithm $\mathsf{Sort}^K(A)$ breaks up a larger instance into a good half G and a bad half B, it calls itself on the good half, i.e., $\mathsf{Sort}^{\lceil K/2\rceil}(G)$; and calls $\mathsf{Sort}^K(B)$ on the bad half, where the domain of keys in G is either $[a+1..a+\lceil\frac{K}{2}\rceil]$ or $[a+\lceil\frac{K}{2}\rceil+1..a+K]$, and the domain of keys in B is always $[a+1..a+K]$. We describe the algorithm informally below, leaving a formal description to Section 5.2.

1. As base cases: 1) if the array A is less than $2Z$ in size, we simply apply the sorting network Zigzag sort [36] and output the result; and 2) if $K \leq 2$, we simply invoke our earlier $\mathsf{Partition}$ algorithm to complete the sorting. Otherwise we will continue with the following steps.
2. First, we divide the input array A into bins of size $Z = \log^6\lambda$ where λ is a security parameter — our algorithm should preserve correctness with $1-\mathsf{negl}(\lambda)$ probability where $\mathsf{negl}(\cdot)$ is a negligible function and moreover we assume that $n = \mathsf{poly}(\lambda)$ for some polynomial function $\mathsf{poly}(\cdot)$.
3. Next, we sort each Z-sized small bin using Zigzag sort, an $O(Z\log Z)$-sized sorting network by Goodrich [36] — in total this takes $O(n\log\log\lambda)$ time.
4. Now, imagine we express the outcome as a short-and-fat matrix, where each column, of height Z, is a sorted bin. Henceforth we refer to the middle $2\log^4\lambda$ rows of this matrix as the *crossover stripe*. Now we again rely on Zigzag sort to sort all elements in this crossover stripe (all elements in all bins in the crossover stripe are sorted altogether).

   Our key observation is the following: after this crossover stripe is sorted, except with negligible in λ probability, it holds that *any element in the top half of the matrix is no larger than even the minimum element in the bottom half.* We formally prove this fact later in Section 5.2. There are two direct implications of this observation — both of the following hold except with negligible probability:

   (a) First, either the top half or the bottom half can still have K distinct keys remaining *but not both.*

   (b) Second, one of the halves must have no more than $\lceil K/2\rceil$ distinct keys remaining.

5. Henceforth, the half that has more distinct keys remaining is referred to as the *bad half*, and the half that has fewer distinct keys remaining is referred to as the *good half.* If both halves have the same number of distinct keys remaining, we may break ties arbitrarily.

   Now, in one scan of each half, we can find the minimum and maximum keys of each half, and thus we can decide which is the good and which is the bad half[12].

6. Now in linear time, we can create an array where the good half is arranged before the bad half. To do this, suppose that the starting point is $[A_0, A_1]$ where A_0 is the top half and A_1 is the bottom half. First, we create two possible arrays in linear time: 1) $[A_0, A_1]$ and 2) $[A_1, A_0]$. Now, we can simply use a multiplexer to pick the right one in linear time. Let $X = [G, B]$ be the outcome of this step. It holds that except with negligible probability, B can have as many as K distinct keys and G can have no more than $\lceil K/2 \rceil$ distinct keys.
7. We next recurse on the bad half B by calling $\widetilde{B} \leftarrow \mathsf{Sort}^K(B)$. Further, we call $\widetilde{G} \leftarrow \mathsf{Sort}^{\lceil K/2 \rceil}(G)$ on the good half G — note that $\mathsf{Sort}^{\lceil K/2 \rceil}(\cdot)$ is an instance of the algorithm capable of sorting keys from the domain $[a'+1..a'+\lceil K/2 \rceil]$ for some a'.
8. Finally, we again use a multiplexer to select the correct arrangement among $[\widetilde{G}, \widetilde{B}]$ and $[\widetilde{B}, \widetilde{G}]$. Clearly this can be accomplished in linear time.

Obliviousness. The obliviousness of the algorithm is straightforward: the access patterns include a random permutation in the beginning, and then afterwards all access patterns are deterministic and data independent.

Runtime. The runtime of this algorithm can be analyzed using the following recurrence where $T(n, K)$ denotes the runtime for sorting n elements whose keys are from $[1..K]$:

$$T(n, K) = T(\lceil \frac{n}{2} \rceil, K) + T(\lceil \frac{n}{2} \rceil, \lceil K/2 \rceil) + O(n \log\log\lambda)$$

Further, we have the following base cases where the former is due to the calling Zigzag sort [36] for small enough bins, and the latter is due to applying our earlier $\mathsf{Partition}$ algorithm for the base case $K \leq 2$.

$$\begin{array}{lll} \text{For } n \leq 2Z: & T(n, K) & = O(n \log n) \\ \text{For } K \leq 2: & T(n, 2) & = O(n \log\log\lambda) \end{array} \tag{1}$$

It is not difficult to show that this recurrence solves to $T(n, K) = O(n \log K \log\log\lambda)$. At this moment, we have that for $o(\log n / \log\log n)$-bit keys, oblivious sorting can be accomplished in $o(n \log n)$ time. Our next section will describe how to optimize parameters of this recurrence to obtain a tighter upper bound, such that we can sort $o(\log n)$-bit keys in $o(n \log n)$ time.

2.3 A Better Recurrence

We observe that in fact, the method in the previous section can be generalized and the parameters improved. We describe our improved algorithm $\mathsf{Sort}^K(A)$ below where K denotes an upper bound on the number of distinct keys in the input array A.

1. We assume that the input A has been arranged in a matrix where each column represents a polylogarithmically-sized bin that has been Zigzag-sorted — this preprocessing step consumes $O(n \log\log\lambda)$ time.
2. Instead of dividing the matrix A into a good half and a bad half, we can divide it into $\log K$ pieces of equal size. We call the neighboring $2\log^4\lambda$ rows near every boundary two pieces a *crossover stripe*.
3. Now, call Zigzag sort to sort every crossover stripe.

[12]Note that here our algorithm is *not* comparison-based, since this step produces a bit that is dependent on the inputs and this bit will later be used in multiplexers for selecting the good and bad half respectively.

4. Let us examine the $\log K$ pieces of the resulting matrix A. We can now generalize our previous reasoning to conclude the following useful observation which holds except with negligible probability: for any $i < j$, any element in the i-th piece must be smaller than even the smallest element in the j-th piece. We will formally prove this later in Section 5.3.
5. Thus in one linear scan, we can write down (an upper bound on) the number of distinct elements in each piece. Specifically, in one linear scan, we can find the maximum and minimum key for each piece and their difference is clearly an upper bound on the number of distinct elements in the corresponding piece.
6. Now, in $O(\frac{n}{\log K} \cdot \log K \cdot \log\log K) = O(n \log\log K)$ time, we can Zigzag-sort all these pieces based on how many distinct keys each piece has, from small to large. Let $A_1, A_2, \ldots, A_k$ denote the resulting pieces where $k = \log K$ and A_i has a smaller number of distinct keys than A_j if $i < j$.

   Now, using similar reasoning as Section 2.2, we observe that A_k can have at most K distinct keys, A_{k-1} can have at most $\lceil K/2 \rceil$ distinct keys, A_{k-2} can have at most $\lceil K/3 \rceil$ distinct keys, A_{k-3} can have at most $\lceil K/4 \rceil$ distinct keys, and so on; finally, A_2 can have at most have at most $\lceil K/(k-1) \rceil$ distinct keys and A_1 can have at most have at most $\lceil K/k \rceil$ distinct keys where $k = \log K$.
7. Based on the above observation, we can make the following recursively calls to sort each piece: $\mathsf{Sort}^{\lceil K/k \rceil}(A_1)$, $\mathsf{Sort}^{\lceil K/(k-1) \rceil}(A_2)$, ..., $\mathsf{Sort}^{K}(A_k)$. We obtain k sorted pieces as a result; and finally, in $O(n \log\log K)$ time, we can Zigzag-sort all these pieces such that their keys are arranged from small to large.

The improved recurrence. The above re-parameterized variant yields the following recurrence:

$$
\begin{aligned}
T(n, K) = \quad & O(n \log\log K) + O(n \log\log \lambda) \\
& + \sum_{i=1}^{k} T\left(\frac{n}{\log K}, \left\lceil \frac{K}{k-i+1} \right\rceil\right),
\end{aligned}
$$

where the $O(n \log\log K)$ term is due to sorting the collection of pieces twice (first time by the number of distinct keys and the second time by the order of the keys), the $O(n \log\log \lambda)$ comes from Zigzag-sorting each bin in the preprocessing step, and all other operations that take linear time are asymptotically absorbed and not explicitly denoted. The base cases remain unchanged — see Equation (1). Solving this new recurrence is a bit more challenging, but it is not difficult to verify that $T(n) = O(n \log\log \lambda \log K / \log\log K)$ is a legitimate solution under the standard assumption that both n and $K \leq \mathsf{poly}(\lambda)$.

At this moment, it is not difficult to see that if $K = 2^{o(\log n)}$ we can obliviously sort in $o(n \log n)$ time.

2.4 Extension: Large Key Space but Few Distinct Keys

So far we have made a short-key assumption, i.e., each ball carries a key that is at most $o(\log n)$ bits long. Upon careful examination, in our earlier algorithm in Section 2.3 the only place where we needed the short-key assumption is due to how we estimated the number of distinct keys for each piece, i.e., by subtracting the minimum key of each piece from the maximum key.

We next propose an extension such that we can remove the short-key assumption, and instead rely on the weaker assumption that the number of distinct keys $\hat{K}$ in the input array is small, although each key can be from a large space. Henceforth we assume that an upper bound on the number of distinct keys denoted $\hat{K}$ is known a-priori to the algorithm. To achieve this we devise a new, almost linear-time oblivious algorithm for estimating the number of distinct keys for each piece — and this building block might be of independent interest in other applications. Our idea is the following.

First, consider a non-oblivious algorithm that estimates the number of distinct elements given an input array that makes use of a random oracle — we will remove the random oracle later using almost k-wise independent hash families; and we will also make the algorithm oblivious. We begin by first hashing all elements into $\log n$ bins using the random oracle — note that elements with the same key will land in the same bin. By averaging, there exists a bin whose load is at most $\frac{n}{\log n}$, and we can identify such bin in linear time. We then count the number of distinct elements D in this bin: it is not difficult to prove that the quantity $D \cdot \log n$ would be a constant-factor approximation of the number of distinct elements in the input array (except with negligible probability).

Our algorithm is based on this idea but we must additionally 1) remove the random oracle and replace it with almost k-wise independent hash families; and 2) make the algorithm oblivious while preserving efficiency. We thus devise the following algorithm where A is an input array provided to the algorithm:

1. Select a random hash h from a k-wise ϵ-independent hash family for appropriate choices of k and ϵ to be specified in later in Appendix D.3.
2. For each element x in the input array A, tag the element with its hash $h(x)$ that is $\log\log n$-bits long.
3. Let B_1 be A to start with. Henceforth assume that each element is tagged with its hash.

   For $j = 1, 2, \ldots, \log\log n$,

   - Obliviously partition B_j based on the j-th bit of the hashes of all elements in B_j — using the algorithm described in Section 2.1, this can be accomplished in $O(|B_j| \log\log\lambda)$ runtime.
   - After this partitioning, either the first half of the resulting array contains the 0-th partition or the second half of the resulting array contains the 1-st partition. Use a multiplexer to select the half of the array that contains either the 0-th partition or the 1-st partition.
   - Assume that the 1st (or 2nd) half of the array is selected w.l.o.g.: now in one linear scan of the outcome, overwrite any element that does not belong to the 0-th partition (or the 1-st partition).
   - Let the resulting array be B_{j+1} — note that the length of B_{j+1} is exactly $|B_j|/2$.
4. Finally, use an oblivious sorting algorithm to count the number of distinct items in $B_{\log\log n}$, and let D be the outcome. Output $D \cdot \log n$ as an estimate of the number of distinct items in A.

In Appendix D.3, we will show that there is a way to concretely instantiate the k-wise, ϵ-independent hash family using a construction proposed by Meka et al. [54] such that the above algorithm completes in $O(|A| \cdot \alpha^2 \log\log\lambda)$ time and provides an estimate of the number of distinct keys in A that is accurate up to a small constant factor except with $\lambda^{-\Omega(\alpha)}$ probability — moreover, this can be accomplished assuming 1) that the RAM's word is large enough to store a ball and its key; and 2) word-level multiplications can be performed in unit cost on a RAM (whereas our earlier algorithms for short keys only required word-level addition, subtraction, and comparison in unit cost).

In Appendix D.2, we will further remove the requirement that the RAM's word is large enough to store the key — to achieve this, we rely on a collision-resistant compression function (which can be instantiated from a 2-wise independent hash family and the Merkle-Damgård construction [23,55]) to compress each possibly long key before embarking on the aforementioned algorithm. We defer the details to Section D. Further, we will also show that using our new distinct key estimation algorithm in lieu of the earlier approach of computing the expression (maximum key - minimum key), we can sort arbitrary-length keys in $O(n\alpha L + n\alpha^2 \log\log n + kLn \log\log n/\log k)$ time where 2^k denotes an upper bound on the number of distinct keys, L denotes the number of RAM words required for storing each key, and α is a term related to the security failure probability $n^{-\Omega(\alpha)}$.

2.5 IO Efficiency in the Cache-Agnostic Model

So far, we have focused entirely on running time. We now show how to improve the IO efficiency of our construction in a cache-agnostic model. Goodrich [35] phrased the open question: can we achieve a linear-IO tight compaction scheme? We show that our IO-efficient scheme solves this open question in the 1-bit special case — previously it was not known how to do this even in the *cache-aware* model. Further, our result implies an IO-efficient selection algorithm (for selecting all m smallest elements) in the cache-agnostic model (and this question was implicitly left open by Goodrich [35] as well).

Random permutation of inputs lacks IO efficiency. The reason our algorithms thus far were not IO-efficient mainly arises from the need to randomly permute the input array during a preprocessing stage. This random permutation is necessary to defeat adversarially chosen inputs, such that for every input, except with negligible probability the output is correct. Aggarwal and Vitter [1] proved that any algorithm that randomly permutates n elements must incur at least $\min\{n, \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\}$ IOs. To asymptotically improve the IO efficiency of our algorithm, we show that a weaker primitive called "bin-wise independent shuffle" is sufficient rather than a full random permutation.

Bin-wise independent shuffle. We observe that a complete random permutation of the inputs is an overkill; and all we need to do is to permute the inputs barely enough, such that our probabilistic analysis will nonetheless hold in the same manner as if the data were permuted completely at random. Interestingly, to realize our *bin-wise independent shuffle* idea, we make use of an additional building block, i.e., a cache-agnostic matrix transpose procedure [31] which is well-known in the algorithms literature — in fact, our algorithm uses this matrix transposition building block in several other places to be able to collect both the rows and columns of a matrix in an IO-efficient manner.

We now elaborate: abstracting away other details of our algorithm, our goal is to divide the input elements into bins such that each bin will have roughly the same fraction of 0s and 1s as the input array. To achieve this, the idea is to view the input array as a short-and-fat matrix, perform a random, cyclic shift of each row (which, as we show, can be accomplished in an IO-efficient manner); and finally relying on a cache-agnostic matrix transpose operation, transform each column of the matrix into a bin. It is not difficult to observe that the Z random coins for each bin are independent where Z denotes the bin's size — hence the name "bin-wise independent shuffle". In Section 4, we formally state and prove the statistical properties we need from this random bin assignment process.

2.6 Lower Bounds for Oblivious Sorting

The following is a proof sketch showing that any algorithm that obliviously sorts $(\log n)$-bit keys must incur $\Omega(n \log n)$ runtime, a special case of Theorem 1.1. The full proof — see Appendix B — and the lower bound of stable oblivious sorting (Theorem 1.3) are both extensions of this sketch. Recall that we consider sorting algorithms in the indivisible model, where algorithms move opaque balls between memory and CPU registers (i.e. bins). Our lower bound counts only the number of such moves, while the CPU is allowed to perform arbitrary computation on the keys.

Let $\mathcal{S}$ be any oblivious algorithm in the indivisible model such that correctly sorts $(\log n)$-bit keys. For simplicity, assume perfect obliviousness and perfect correctness for the time being. Fixing an input and a random tape, $\mathcal{S}$ has a fixed pattern of moves of balls between CPU registers and memory. Let G be a directed acyclic graph that represents such pattern of moves constructed as follows: Consider each memory location or the CPU register is a bin; Each vertex in G represents a (bin, time step) pair, and each edge in G represents an *admissible* move from a bin corresponding to time step t to another bin in time step $t' > t$. By perfect obliviousness, if G is encountered on one input with non-zero probability, it must be encountered for every input with non-zero probability. By perfect correctness and the indivisible nature of the algorithm, for each input, there exists a vertex-disjoint routing from input vertices to output vertices in G such that the routing sorts the input — we henceforth say that G *explains* the input. Specifically, G explains all n cyclic shifts of the input sequence $(1, \ldots, n)$, and thus every input vertices in G is routed to a distinct output vertex in each of the shifts. In the beautiful work of Pippenger and Valiant [60], they show that such graph G must have at least $\Omega(n \log n)$ edges. It follows that $\mathcal{S}$ must take time $\Omega(n \log n)$ because each time step creates at most 4 edges in G. To strengthen the proof to algorithms with $O(1)$ correctness, we apply an averaging argument, which is deferred to Appendix B.2.

2.7 Comparison with Known ORAM Lower Bounds

We remark that neither of our lower bounds (Theorem 1.1 and Theorem 1.3) is implied from Goldreich and Ostrovsky's ORAM lower bound [32, 33] or its proof techniques; nor implied by the more recent ORAM lower bound by Larsen and Nielsen [46] (which is in fact subsequent to our work).

Background on ORAM lower bounds. Both works [33, 46] argue that any ORAM scheme must incur logarithmic blowup; but Goldreich and Ostrovsky's lower bound is applicable to only the "balls-and-bins" model and statistical security; whereas Larsen and Nielsen remove these restrictions, and prove the ORAM lower bound for even "non-balls-and-bins" algorithms, and computational security (where the balls-and-bins is the same as the indivisible model). On the other hand, Goldreich and Ostrovksy's lower bound applies even to offline ORAMs, whereas Larsen and Nielsen's result applies only to online ORAMs (i.e., the memory requests are determined in an online fashion).

Comparison with our lower bounds. Both of the aforementioned ORAM lower bounds state that a logarithmic blowup is necessary to compile a *generic* RAM program to an oblivious one. This means that there exists a non-oblivious algorithm Alg running in time $T(n)$ on an input memory array of size n, such that any oblivious algorithm functionally equivalent to Alg must incur at least $T(n) \cdot \Omega(\log n)$ runtime. These lower bounds do not imply that for *every* algorithm this logarithmic blowup is necessary. As the simplest example, counting the number of 0s in an array can be done obliviously in linear time, and thus the oblivious simulation overhead is $O(1)$.

Even more specifically, these known ORAM lower bounds do not seem to directly imply a general sorting (i.e., sorting n-bit keys) lower bound either. For example, Goldreich and Ostrovksy's lower bound, imprecisely speaking, says that any oblivious algorithm, that has to make logical accesses in an order prescribed by some non-oblivious algorithm, must incur logarithmically higher runtime than the non-oblivious algorithm. Thus to directly apply their ORAM lower bound to argue that any generic oblivious sorting algorithm (in the indivisible model) must incur $\Omega(n \log n)$ runtime, it seems necessary to argue that *any* non-oblivious sorting algorithm must access memory in sufficiently many permutations upon different inputs. While this may not be impossible, such an extension would at least be somewhat non-trivial.

On the other hand, our Theorem 1.3 in fact implies Goldreich and Ostrovsky's ORAM lower bound: to prove the ORAM lower bound it suffices to find one task whose oblivious implementation must be at least $\Omega(\log n)$-factor more expensive than a non-oblivious implementation. Let 1-bit-key stable sorting be this task: there is a trivial linear-time, non-oblivious algorithm that stably sorts 1-bit keys in the indivisible model; and Theorem 1.3 shows that any oblivious implementation must be at least $\Omega(\log n)$-factor more expensive. Note that just like Goldreich and Ostrovsky's lower bound, our lower bound is also in the "balls-and-bins" model and applies only to statistical security.

Technically, our proofs introduce a new paradigm where we lower bound the complexity of *access pattern graphs* of oblivious algorithms — to achieve this, we draw connections to the study of non-blocking graphs in the classical algorithms literature [60]. We believe that these techniques can spur new applications: for example, they were adopted in the subsequent work where Chan et al. [18] investigate the complexity of differentially oblivious algorithms.

3 Definitions and Preliminaries

Negligible functions. A function $\epsilon(\cdot)$ is said to be *negligible* if for every polynomial $p(\cdot)$, there exists some λ_0 such that $\epsilon(\lambda) \leq \frac{1}{p(\lambda)}$ for all $\lambda \geq \lambda_0$.

Statistical indistinguishability. For an ensemble of distributions $\{D_\lambda\}$ (parametrized with λ), we denote by $x \leftarrow D_\lambda$ a sampling of an instance according to the distribution D_λ. Given two ensembles of distributions $\{X_\lambda\}$ and $\{Y_\lambda\}$, we say that the two ensembles are statistically indistinguishable, often written as $\{X_\lambda\} \overset{\epsilon(\lambda)}{\equiv} \{Y_\lambda\}$, iff for any unbounded adversary $\mathcal{A}$,

$$\left| \Pr_{x \leftarrow X_\lambda} \left[\mathcal{A}(1^\lambda, x) = 1 \right] - \Pr_{y \leftarrow Y_\lambda} \left[\mathcal{A}(1^\lambda, y) = 1 \right] \right| \leq \epsilon(\lambda).$$

3.1 Oblivious Algorithms on a RAM

In this paper, we consider a Random-Access Machine (RAM) where a CPU interacts with a memory to perform computation. In every step of the computation, the CPU can read and/or write a memory location, perform computation over words stored in its CPU registers, update the values stored in (a subset of) its CPU registers.

Assumptions. We assume that the CPU has only $O(1)$ registers, and it also has access to a private random string that is unobservable by the adversary. Unless otherwise noted, we assume that word-level additions and comparisons can be performed in unit cost on the RAM, where word size is $\Theta(\log \lambda)$ bits — for our algorithms for short keys (Sections 4 and 5), this is sufficient; however, our algorithm in Appendix D that support arbitrary key spaces but few distinct keys additionally require that word-level multiplication be computed in unit cost.

Oblivious algorithms. We formally define what it means for a RAM program to be oblivious.

Definition 3.1 (Oblivious algorithm)**.** *We say that a (possibly randomized) algorithm* Alg *is* oblivious*, iff the following holds: for any inputs* $I_0, I_1 \in \{0,1\}^*$ *such that* $|I_0| = |I_1|$*,* $\mathsf{Alg}(1^\lambda, I_0)$*'s access patterns to memory and* $\mathsf{Alg}(1^\lambda, I_1)$*'s access patterns to memory are identically distributed.*

Throughout this paper, all of our algorithms are *perfectly oblivious* but may suffer negligibly small *statistical failure probability in terms of correctness.* (If the output is not correct, we can then trade security for correctness by checking and running a non-oblivious but efficient sorting.)

In this work, both the upper bound and lower bound results in the oblivious algorithms model hold also in a stronger model of probabilistic circuits (please see Appendix A.1 for the definition and efficiency metrics).

3.2 External-Memory and Cache-Agnostic Algorithms

Not only do we care about the runtime of a RAM program, we also care about minimizing IO-cost. To characterize IO efficiency, we adopt the same (possibly multi-level) cache-memory model as adopted in the standard, *external-memory* algorithms literature [1, 25, 30, 31, 70]. In the external-memory model, besides the CPU, the storage hierarchy is implemented by a *cache* and an *external-memory.* As before, in each step of execution, a CPU performs some computation over its internal registers, and then makes a `read` or a `write` request to the storage hierarchy.

Memory requests are served in the following manner:

- If the address requested is already in the cache, the CPU then interacts with the cache to complete the read or write request and thus no external memory read or write is incurred;
- Else if the address requested does not exist in the cache: 1) first, a *cache-line* containing the requested address is copied into the cache from external memory possibly evicting some existing cache-line from the cache in the process where the evicted cache-line is written back to memory; and 2) then the CPU interacts with the cache to complete the read or write request. Thus, a *cache-line* defines the atomic unit of access between the cache and the external memory.

Notation. Throughout this paper, we use the notation M to denote the cache size, i.e., the number of words the cache can contain; and we use the notation B to denote a cache-line size, i.e., the number of words contained in a cache-line.

An algorithm's IO efficiency. In such an external-memory model, an algorithm's *IO-cost* is the number of times a cache-line is transferred between the memory and the cache (thus IO-cost characterizes the number of cache misses).

Cache-aware vs. cache-agnostic algorithms. As in the standard algorithms literature, if an external-memory algorithm must know a-priori the parameters of the cache (i.e., M and B), it is said to be *cache-aware*. If an external-memory algorithm need not know the cache parameters, it is said to be *cache-agnostic* [25, 31]. As is well-known in the algorithms community, cache-agnostic algorithms offer numerous compelling advantages over cache-aware ones: first, any cache-agnostic algorithm with optimal IO performance can readily run on any architecture with unknown architectural parameters and achieve optimal IO performance; second, on any multi-level memory hierarchy, optimality is achieved in between any two adjacent levels (e.g., between the cache and memory, between the memory and disk, and between the local disk and the cloud server).

Standard cache assumptions. Our assumptions about the cache are standard and well-accepted in the algorithms literature. We assume that the cache has full associativity and moreover implements an optimal replacement policy — the justifications of such assumptions have been clearly articulated in the algorithms literature: although common architectures in practice may have different realizations, costs in this ideal cache model can be easily translated to costs on common practical architectures (see Appendix A.2 for details).

Our IO-efficiency-related upper bound results assume that $M \geq \log^{1.1} n$ — this is satisfied if we assume the following standard cache assumptions that are widely adopted in the external-memory algorithms literature [6, 10, 11, 15, 25, 31, 35, 53, 58, 61, 62, 72, 75]:

- *Tall cache.* A tall cache assumption states that the cache is *taller* than it's width; that is, the number of cache line, M/B, is greater than the size of one cache line, B, where M is the size of cache; or simply stated, $M \geq B^2$.
- *Wide cache-line.* The wide cache-line assumption states that $B \geq \log^c n$ where c is a constant (typical works assume that $c \geq 0.55$) and n is space consumed by the algorithm.

Oblivious algorithms in the external-memory model. Oblivious algorithms in the external-memory model is similarly defined as in Definition 3.1. We stress that even in the external-memory model, for our obliviousness definition we assume that the adversary can observe full memory addresses being requested in every CPU step — our definition is stronger than those adopted by some earlier works on external-memory oblivious algorithms [35, 37, 39] such as Goodrich et al. [35] — earlier works assume that the adversary can only observe the accesses between the cache and the memory but not the accesses between the CPU and the cache. Chan et al. [20] were in fact the first to phrase this notion of obliviousness for external-memory algorithms — they referred to this stronger notion as *strong obliviousness*, and the weaker notions adopted earlier [35] as *weak obliviousness*. They also argue that this strong notion is desired in practical applications such as oblivious algorithms for secure processors such as the popular Intel SGX [3, 22, 52] — since strong obliviousness defends against a well-known cache-timing attack whereas the weak obliviousness provides no such defense.

3.3 Building Blocks

3.3.1 Cache-Agnostic Algorithms

Cache-agnostic oblivious matrix transpose. Frigo et al. [31] provides a matrix transposition with optimal runtime and IO-cost, and it is also oblivious.

Lemma 3.1 (Theorem 3, [31])**.** *There is an cache-agnostic, oblivious algorithm* Transpose *such that given an $m \times n$ matrix A stored in row-major layout, the algorithm computes the transpose A^T in runtime $O(mn)$ and IO-cost $O(\frac{mn}{B})$.*

Cache-agnostic non-oblivious random cyclic-shift. Given an input array I of n elements, a *random cyclic-shift* algorithm outputs an array O such that $O[i] := I[(i+r) \mod n]$ for all i, where r is uniformly sampled at random from $[n]$. The naïve cache-agnostic algorithm shift runs in time $O(n)$ and IO-cost $O(\lceil n/B \rceil)$ as follows: sample r uniformly from $[n]$; for i from 0 to $n-1$, move $I[(i+r) \mod n]$ to $O[i]$. Note that shift reveals randomness r (and hence is not oblivious w.r.t. r).

3.3.2 Oblivious Sorting

Sorting circuits. Ajtai et al. [2] (AKS) and Goodrich [36] (Zigzag sort) show that sorting circuits with $O(n \log n)$ comparators can be constructed. Later, we will make use of such sorting circuits as a building block (particularly on problems of small sizes).

Funnel oblivious sort. Though Zigzag and AKS sort are asymptotically efficient in runtime, they are not as good in terms of IO-cost. Chan et al. [20] devised an *IO-optimal* (randomized) oblivious sorting algorithm in the *cache-agnostic* model. At a high level, their algorithm invokes an instance of funnel sort [31] in a non-blackbox fashion to randomly permute inputs, and then invokes another instance of funnel sort [31] (this time in a blackbox fashion) to sort the permuted inputs — hence we call their algorithm "funnel oblivious sort".

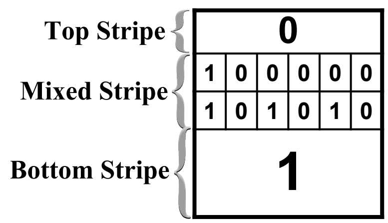

Figure 1: Matrix layout and example matrix after transpose

In our paper, we rely on a further improved version of Chan et al. [20]'s algorithm henceforth denoted FunnelOSort — in particular, we substitute the bitonic sort in Chan et al. [20]'s construction with an improved building block called DeterministicPart (i.e., deterministic partitioning) as described in Section 4.2. We state the improved result in the following lemma.

Lemma 3.2. (Theorem 5.7, [20] with improvements described in Section 4.2). *Assuming $M = \Omega(B^2)$, and $B = \Omega(\log^{0.55} \lambda)$, there exists a cache-agnostic, oblivious sorting algorithm henceforth denoted* FunnelOSort*, which, except with negligible in λ probability, correctly sorts n elements in $O(n \log n \log\log \lambda)$ time, $O(n)$ space and $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ IO-cost.*

Note that the work of Chan et al. claims the runtime $O(n \log n (\log\log \lambda)^2)$, and thus our improvements in Section 4.2 improves a $\log\log \lambda$ factor.

4 Partitioning 1-Bit Keys

Our first step is to realize 1-bit partitioning: given n balls each tagged with a 1-bit key, output an array containing the same balls (tagged with keys) according to the relative ordering of their keys. The algorithm need not be stable — as shown in Theorem 1.3, no oblivious algorithm in the indivisible model can realize 1-bit *stable* partitioning in $o(n \log n)$ time.

Assumptions. Throughout this section, we shall assume that the RAM's word size is large enough to store each ball as well as its key. We assume that word-level addition, subtraction, and comparison operations can be computed in unit cost (but we do not need word-level multiplication in this section).

4.1 Intuition

The high-level idea of our 1-bit partitioning scheme (the IO-inefficient version) was described in Section 2.1. To quickly recap, the idea is to randomly permute the input balls and divide them into $\log^6 \lambda$-sized bins. Now we arrange each bin as the column of a matrix, and sort each column. It thus holds that except with negligible in λ probability, there exists a small *mixed stripe* containing at most $\log^4 \lambda$ rows such that all rows above (called the top stripe) contain 0s and all rows below (called the bottom stripe) contain 1s (see Figure 1). The formal proof of this statement will be presented later in Lemma 4.1. Now, our algorithm simply obliviously sorts (e.g., using Zigzag sort) the mixed stripe, and outputs the top stripe, mixed stripe, and bottom stripe in this order.

To make the above algorithm IO-efficient, first, we need to instantiate the oblivious sort with an IO-efficient construction. In our case, we only need to sort 1-bit keys — we thus devise an IO-efficient, deterministic 1-bit sorting algorithm, DeterministicPart, in Section 4.2. Secondly, as mentioned earlier in Section 2.5, a novel idea we have is to replace the random permutation in the preprocessing stage with a weaker primitive that is IO-efficient and still sufficient for our statistical

Algorithm 1 Partitioning 1-bit keys

procedure: $\mathsf{Partition}(A)$ *// We parse A as a $Z \times \frac{n}{Z}$ matrix where $Z = \log^6 \lambda$*
For each row in A, perform a $\mathsf{RandCyclicShift}$.
Sort *columns*:
$A^T \leftarrow \mathsf{Transpose}(A)$. Use $\mathsf{DeterministicPart}$ to sort each row of A^T. $A \leftarrow \mathsf{Transpose}(A^T)$.
In one scan, identify the first row i with mixed 0s and 1s, and rows $[i, \ldots, i+\delta]$ are said to be the *mixed stripe* where $\delta := \log^4 \lambda$.
Move the mixed stripe to a working buffer using $\mathsf{MoveStripe}(A, i, i+\delta)$, where the working buffer is represented as a $\delta \times \frac{n}{Z}$ matrix.
Run $\mathsf{DeterministicPart}$ to sort the working buffer.
Obliviously cyclic-shift the working buffer by $(i \mod \delta)$ rows using $\mathsf{FunnelOSort}$.
Move the working buffer back to the original location in A using the reversed $\mathsf{MoveStripe}$.
return A

guarantees. Thus, we perform a "bin-wise independent shuffle" operation, where we express the initial array A as a short-and-fat matrix of $\log^6 \lambda$ height. We then perform an independent random cyclic shift of each row (the random shift offset need not be secret), and then we call each column a bin. In our formal algorithm description to follow, we show how to combine this idea with a cache-agnostic matrix transpose operation to achieve IO efficiency.

4.2 New Building Blocks

We will need two new building blocks, $\mathsf{MoveStripe}$ and $\mathsf{DeterministicPart}$.

Oblivious deterministic partitioning. We devise an IO-efficient algorithm (in the cache-agnostic model) called $\mathsf{DeterministicPart}$ that is capable of sorting balls tagged with 1-bit keys in $O(n \log n)$ runtime and $O(\frac{n}{B} \log_M n)$ IO-cost. We defer the details of this algorithm to Appendix C.1.

Obliviously copying a stripe to a working buffer. Given a matrix A written down in a row-major manner in memory, a *stripe* is defined as a set of consecutive rows from l to r. The $\mathsf{MoveStripe}(A, l, r)$ algorithm copies all elements in the rows from l to r to a working buffer of equivalent size (in any order), without revealing the location of the stripe, i.e., l, r, and A. The detailed algorithm (deferred to Appendix C.2) runs in $O(n)$ time and $O\left(\frac{n}{B}\right)$ IO-cost, where n is the number of entries in the matrix A.

4.3 Algorithm for Partitioning 1-Bit Keys

We now describe the detailed $\mathsf{Partition}$ (Algorithm 1) that sorts balls tagged with 1-bit keys, and it is IO-efficient in the cache-agnostic model. In $\mathsf{Partition}$, all matrices are row-major in memory, which matters when discussing IO efficiency. Let δ be $\log^4 \lambda$ and Z be $\log^6 \lambda$.

4.4 Analysis

Correctness. Let the *minimum mixed stripe* be the minimum set of rows such that all rows above it contain 0s and all rows below it contain 1s. The number of rows contained in the minimum mixed stripe is said to be the *height* of the minimum mixed stripe.

Lemma 4.1 (Mixed stripe is small)**.** *There exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any input key assignment, except with* $\mathsf{negl}(\lambda)$ *probability, the height of the minimum mixed stripe in the* Partition *algorithm is upper bounded by* $\delta = \log^4 \lambda$.

Proof. Sorting a row in A^T is equivalent to sorting the corresponding column in A, and hence, in this proof, we consider columns in A. Consider a fixed column in A after the Partition algorithm performs the RandCyclicShift — recall that such a column is called a bin, and its size is $Z = \log^6 \lambda$.

Let bin B be a fixed column. Let X_i be the bit associated with the i-th ball in bin B. We observe that X_i is picked at random from the i-th row of the original matrix A (before the RandCyclicShift line). Note that X_i draws from Bernoulli distribution $B(1, p_i)$, where p_i is the fraction of 1 in i-th row. Further, for any $i \neq j$, X_i and X_j are independent. Let $X := \sum_{i=1}^{Z} X_i$ be the total number of 1s in B. We next observe that, for *every bin* B, the expectation equals to $\mathbf{E}[X] = \sum_{i=1}^{Z} p_i$. By Hoeffding's inequality, for every bin B, we have

$$\Pr(|X - \mathbf{E}[X]| \geq \frac{1}{2} \log^4 \lambda) \leq 2e^{-2\frac{(\log^4 \lambda)^2}{4Z}} = 2e^{-\frac{1}{2}\log^2 \lambda}.$$

It follows that, for each bin, except with probability $\epsilon(\lambda) = 2e^{-0.5 \log^2 \lambda}$, the total number of 1s must be within the range $R = (\mathbf{E}[X] - \frac{1}{2}\delta, \mathbf{E}[X] + \frac{1}{2}\delta)$. Let Y_k be the total number of 1s in the k-th bin. By union bound, the probability that there exists k such that $Y_k \notin R$ is upper bounded by $\mathsf{negl}(\lambda) = \frac{n}{Z}\epsilon(\lambda)$. It follows that, after sorting each row in A^T and transposition, the height of the minimum mixed stripe is at most δ except with $\mathsf{negl}(\lambda)$ probability. □

Corollary 4.1. *Given any input array with* n *elements and each with a 1-bit key,* Partition *correctly sorts it except with* $\mathsf{negl}(\lambda)$ *probability.*

Proof. Note that all 0s and 1s that are not in the mixed stripe are located in the correct position after sorting each row in A^T and transposition. By Lemma 4.1, except with negligible probability, there exists some mixed stripe of fixed size R that contains the *minimum mixed stripe*. Conditioning on such good event, the remaining procedures of Partition finds the minimum mixed stripe and correctly sorts elements in the *mixed stripe*, which implies the correctness of the final output. □

Time complexity.

Lemma 4.2. *Given input* A *and security parameter* λ*, where* n *denotes number of elements in* A*,* Partition *completes in time* $O(n \log\log\lambda)$.

Proof. RandCyclicShift and Transpose run in $O(n)$ time. For each column of A, DeterministicPart takes time $O(\log^6 \lambda \log\log\lambda)$, summing up to $O(n \log\log\lambda)$. MoveStripe takes time $O(n)$, and then sorting the working buffer of $\frac{n}{\log^2 \lambda}$ elements (using FunnelOSort) takes time $O(\frac{n}{\log^2 \lambda} \log n \log\log\lambda)$, which is $o(n)$ as $n = \mathsf{poly}(\lambda)$. Hence, the total runtime is $O(n \log\log\lambda)$. □

IO-cost. We now analyze the scheme's IO efficiency.

Lemma 4.3. *The overall IO-cost of* RandCyclicShift *in Line 2 is* $O(\frac{n}{B})$.

Proof. Considering the input size n and the cache-line size B, the bound holds for both cases:

1. $\frac{n}{Z} > B$, each RandCyclicShift instance takes at most $2(\frac{n}{ZB} + 1)$ IOs. Further, there are Z such instances, thus the overall cost is $O(\frac{n}{B})$.

2. $\frac{n}{Z} \leq B$, with each cache load, we load $\lfloor \frac{B}{n/Z} \rfloor$ rows in the matrix. Since we need to load Z rows in total, the total number of loads is $\frac{Z}{\lfloor \frac{B}{n/Z} \rfloor} \leq 2\frac{Z}{\frac{B}{n/Z}} = O(\frac{n}{B})$.

□

Lemma 4.4. *The algorithm takes $O(\lceil \frac{n}{B} \rceil \log_M \log \lambda)$ IO-cost.*

Proof. By Lemma 4.3, Line 2 takes $O(\frac{n}{B})$ IOs. For Line 3, Transpose takes $O(\frac{n}{B})$ IOs by Lemma 3.1. After Transpose, each column in A is consecutive in memory, so DeterministicPart runs in $O(\frac{n}{Z} \cdot \lceil \frac{Z}{B} \rceil \log_M Z)$ IOs if $Z > M$ (and $O(\lceil \frac{n}{M} \rceil)$ otherwise). Then, MoveStripe performs linear scan and costs $O(\lceil \frac{n}{B} \rceil)$ IOs. Sorting the working buffer and cyclic shifting run in $O(\lceil \frac{n}{B} \rceil)$ as the problem size is only $O(\frac{n}{\log^2 n})$. In summary, the overall IO-cost is $O(\lceil \frac{n}{B} \rceil \log_M \log \lambda)$. □

It follows from Lemmas 4.2 and 4.4:

Theorem 4.1. *There exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all input A,* Partition *is a cache-agnostic and oblivious algorithm, correctly sorts A in time $O(n \log \log \lambda)$ and IO-cost $O(\lceil \frac{n}{B} \rceil \log_M \log \lambda)$ except with probability* $\mathsf{negl}(\lambda)$*. Assuming wide cache-line and tall cache (hence $M \geq \log^c \lambda$ for some constant $c > 0$), the IO-cost is $O(\lceil \frac{n}{B} \rceil)$.*

5 Sorting Short Keys

Our earlier algorithm Partition allows us to sort balls carrying 1-bit keys. In this section, we show how to use Partition as a starting point and recursively sort larger keys. For simplicity, we first describe a simple reduction in Section 5.2 that allows us to sort only $o(\log n / \log \log n)$-bit keys in $o(n \log n)$ time. Later in Section 5.3, we will show how to improve the recurrence to sort $o(\log n)$-bit keys in $o(n \log n)$ time.

Assumptions. Throughout this section, we shall assume that the RAM's word size is large enough to store each ball as well as its key. We assume that word-level addition, subtraction, and comparison operations can be computed in unit cost (but we do not need word-level multiplication in this section).

Notation. The notation "domain $[K]$" is abused to denote any set $\{a, a+1, \ldots, a+K-1\}$ of size K for some integer a — in other words, our SortSmall algorithm can sort not just integer keys from $\{1, \ldots, K\}$, but in fact any contiguous domain of size K as long as each key and ball can be stored in a single memory word.

5.1 Intuition

Earlier we described the intuition how to leverage a 1-bit partitioning building block to solve the problem of sorting longer keys. We briefly recap at this moment.

Suppose we would like to sort keys from some contiguous domain $[K]$. The idea is to still sort each column of the matrix where each column is a bin of size $Z = \log^6 \lambda$. Now, we make a critical observation: if we sort the middle $2 \log^4 \lambda$ rows of this matrix, then, except with negligible in λ probability, it must be that any element in the top half of the matrix is no larger than even the minimum element in the bottom half.

Two useful implications come out of this observation. First, at most one half (either the top half or the bottom half but not both) can still have K distinct keys left. Second, the instance that has fewer number of distinct keys left has at most $\lceil \frac{K}{2} \rceil$ distinct keys. These two observations allow us to break the problem apart into 1) an instance of half the size of sorting keys from the same domain $[K]$ as where we started, and 2) an instance of half the size, but sorting keys from the domain $[\lceil \frac{K}{2} \rceil]$ which is roughly half the size of the initial domain. Note that to realize this divide-and-conquer strategy obliviously requires using multiplexers to select a problem instance from two possibilities, and then for the outcome we again need to apply a multiplexer to select an answer from two possible answers — we defer these somewhat more tedious details to Section 5.2.

In our full algorithm description in Section 5.2, we also aim to optimize the IO efficiency of the algorithm. The techniques for IO efficiency here are the same as our techniques for the 1-bit partitioning algorithm earlier in Section 4: essentially, we use bin-wise independent shuffling rather than a full random permutation in the preprocessing stage; and further, we rely on cache-agnostic matrix transposition several times to be able to operate on either the rows or the columns of a matrix in an IO-efficient manner.

5.2 Warmup Scheme

Notations. Let $\delta = \log^4 \lambda$. For simplicity, we use "element" and "key of element" interchangeably in the following.

New building block. The building block Selection is an oblivious algorithm that finds the element of rank r in input an array, and its runtime and IO-cost is asymptotically the same as Partition — in fact, in Appendix E.2, we show that we can select all r smallest elements (not just the r-th element) in almost linear time and linear IO; and just this building block alone solves an open problem phrased by Goodrich [35].

Detailed algorithm. In Algorithm 2, we describe a warmup algorithm, SortSmall, that can sort $o(\log n / \log \log n)$-bit keys in $o(n \log n)$ time. The algorithm also describes several optimizations that are necessary for IO efficiency.

The procedure of SortSmall is similar to Partition (Algorithm 1). Hence, the correctness follows in a similar way: after sorting columns and sorting crossover stripe (Line 5 and 6), with overwhelming probability, compared to the median key, the top half consists of only smaller keys and the bottom half consists of only larger keys. We defer the detailed analysis of probability, the recursive analysis of time and IO-cost to Appendix F.1.

Theorem 5.1 (Sorting small keys)**.** *Assuming tall cache and wide cache-line, there exists a negligible function* $\mathsf{negl}(\cdot)$ *and a cache-agnostic, oblivious algorithm (with an explicit construction) which, except with* $\mathsf{negl}(\lambda)$ *probability, correctly sorts an input array containing n (key, value) pairs where the keys take value in the domain $[1..K]$ in running time $O(n \log K \log \log \lambda)$ and IO-cost $O(\lceil \frac{n}{B} \rceil \log K)$.*

5.3 Improved Recurrence

To sort even longer keys in $o(n \log n)$ time, we further divide the matrix into more pieces (rather than only two pieces), and thus the key domain of each piece is further reduced except for the worst piece. To achieve obliviousness, we perform an additional oblivious sort to permute pieces, so the access pattern is independent from the real key domain of pieces. The algorithm Sort is presented in Algorithm 3, where the differences between Sort and SortSmall (Algorithm 2) are marked in blue. The building block osort is an oblivious sorting algorithm, which is instantiated differently to achieve either runtime or IO efficiency.

In Algorithm 3, to get the best runtime, we shall instantiate each osort with AKS or Zigzag sort [2, 36], which yields the following runnning time. We defer the analysis of correctness and the

Algorithm 2 Sort keys from $[K]$

1: **procedure:** $\mathsf{SortSmall}(A, K)$ *// We parse A as a $Z \times \frac{n}{Z}$ matrix where $Z = \log^6 \lambda$.*
2: **if** $|A| < 2Z$ **then return** $\mathsf{FunnelOSort}(A)$
3: **if** $K \leq 2$ **then return** $\mathsf{Partition}(A)$
4: Run $\mathsf{RandCyclicShift}$ on each row and transpose the resulting matrix.
5: **Sort *columns*:** Let ranks $r_1 := Z/2 - \delta,\ r_2 := Z/2 + \delta$. For each column of A, find elements a_1, a_2 of ranks r_1, r_2 using $\mathsf{Selection}$, and then partially sort the column (using $\mathsf{Partition}$) such that: (a) a_1, a_2 are the r_1, r_2-th elements, (b) all elements located above a_1 are taking value at most a_1, (c) all elements between a_1 and a_2 are in the domain $[a_1, a_2]$, (d) and all below a_2 are at least a_2. Transpose the resulting matrix.
6: **Sort *crossover stripe*:** Let the middle 2δ rows be the crossover stripe. Run $\mathsf{FunnelOSort}$ on the crossover stripe.
7: In one scan, find the minimum and maximum key for each half.
8: Let the number of distinct keys in each half be (max key $-$ min key $+ 1$).
9: **if** top half has $\leq \lceil \frac{K}{2} \rceil$ distinct keys **then** *// Swap, $\mathsf{DummySwap}$ implement a multiplexer*
10: $\quad$ $\mathsf{Dummyswap}$(top half, bottom half).
11: $\quad$ **Sort *sub-problems*:** $\mathsf{SortSmall}$(top half, $\lceil \frac{K}{2} \rceil$), $\mathsf{SortSmall}$(bottom half, K).
12: $\quad$ $\mathsf{Dummyswap}$(top half, bottom half).
13: **else**
14: $\quad$ Swap(top half, bottom half).
15: $\quad$ **Sort *sub-problems*:** $\mathsf{SortSmall}$(top half, $\lceil \frac{K}{2} \rceil$), $\mathsf{SortSmall}$(bottom half, K).
16: $\quad$ Swap(top half, bottom half).
17: **return** the resulting matrix A

calculation of recursive running time to Appendix F.2.

Theorem 5.2 (Sorting keys in domain K)**.** *There exists a negligible function* $\mathsf{negl}(\cdot)$ *and a cache-agnostic, oblivious algorithm (with an explicit construction) which, except with* $\mathsf{negl}(\lambda)$ *probability, correctly sorts an input array containing n (key, value) pairs where the keys take value in the domain $[1..K]$ in running time $O(n \frac{\log K}{\log \log K} \log \log \lambda)$.*

IO-efficient instantiation. To get the best IO efficiency in the cache-agnostic model, in the algorithm Sort, we instantiate every osort with $\mathsf{FunnelOSort}$ except for sorting the pieces (i.e., Line 13). To sort $p = \log K$ pieces in Line 13, arrange the memory layout such that for any i, the i-th element in piece j are packed together for all $j \in [p]$, and then perform $\mathsf{FunnelOSort}$ on each pack of p elements for every $i \in [n/p]$. Note that $p < M$ as $M = \Omega(\log^{1.1} \lambda)$ by wide cache-line and tall cache assumptions. The correctness follows by the same proof of Theorem 5.2 (see Appendix F.2), and the time and IO-cost is stated as follows.

Corollary 5.1. *Assuming tall cache and wide cache-line, the aforementioned IO-efficient instantiation of* Sort *runs in time $O(n \frac{\log K}{\log \log K} \log^2 \log \lambda)$ and IO-cost $O(\lceil \frac{n}{B} \rceil \frac{\log K}{\log \log K})$.*

Proof. Sorting the columns, the crossover stripes, and the pieces take $O(n \log Z \log \log \lambda)$, $O(n \log \log \lambda)$, and $O(n \log p \log \log \lambda)$ respectively. The recursion solves to $O(n \frac{\log K}{\log \log K} \log^2 \log \lambda)$.

The IO-cost to sort columns is $O(\frac{n}{Z} \lceil \frac{Z}{B} \rceil \log_{\frac{M}{B}} \frac{Z}{B})$, to sort crossover stripes is $O(\lceil \frac{n}{B} \rceil)$, to sort pieces is $O(\frac{n}{p} \lceil \frac{p}{B} \rceil)$ as $p < M$, and the base case $\mathsf{SortSmall}$ is $O(\lceil \frac{n}{B} \rceil \log K)$. Hence, the recursion solves to $O(\lceil \frac{n}{B} \rceil \frac{\log K}{\log \log K})$ as the dominating factor $\log_{\frac{M}{B}} \frac{Z}{B}$ of sorting columns is bounded by a constant assuming tall cache and wide cache-line. $\square$

Algorithm 3 Sort $o(\log n)$-Bit Keys

procedure: $\mathsf{Sort}(A, K)$ *// The input A is an array of n elements, each element is a key in a domain $[K]$. Parse the array into a $Z \times \frac{n}{Z}$ matrix, where $Z = \log^6 \lambda$.*

if $|A| < 2Z$ **then**

    return $\mathsf{osort}(A)$

if $K < 64$ **then**

    return $\mathsf{SortSmall}(A, K)$

Perform $\mathsf{RandCyclicShift}$ on each row of A.

Sort *columns*: $A^T \leftarrow \mathsf{Transpose}(A)$. osort each row of A^T. $A \leftarrow \mathsf{Transpose}(A^T)$.

Let $p := \lfloor \log K \rfloor$, $q := \frac{Z}{p}$. Parse matrix A as p *pieces*, $A_1, \ldots, A_p$, where each piece A_j is a $q \times \frac{n}{Z}$ sub-matrix of A.

for i from 1 to $p - 1$ **do**

    Sort *crossover stripes*: osort the boundary between A_i and A_{i+1}. That is, sort all elements between row $iq - \delta$ and row $iq + \delta$.

for i from 1 to p **do**

    Find the maximum key x and minimum key y in A_i. Let $K_i := x - y + 1$.

Sort *all pieces* $\{A_i\}_{i \in [p]}$ obliviously in increasing order of K_i: $\{B_i\}_{i \in [p]} \leftarrow \mathsf{osort}(\{A_i\}_{i \in [p]})$. We assume each B_i remembers its original piece index.

for i from 1 to p **do**

    Sort *sub-problems*: $\mathsf{Sort}(B_i, \lceil \frac{K}{p-i+1} \rceil)$.

Obliviously sort $\{B_i\}_{i \in [p]}$ by their original piece indexes: $\{C_i\}_{i \in [p]} \leftarrow \mathsf{osort}(\{B_i\}_{i \in [p]})$.

return $\{C_i\}_{i \in [p]}$.

6 Open Questions

Our paper raises several interesting open questions:

- Can we achieve similar results as in our paper but with *deterministic* algorithms?
- In a non-indivisible model, can we overcome the $n \log n$ barrier for oblivious sorting and/or tight, order-preserving compaction with non-comparison-based techniques?
- In our algorithm for sorting $o(\log n)$-bit keys as well as the prior work by Chan et al. [20], to get the best IO efficiency results would introduce an extra $\log \log$ factor to the runtime (relative to the best known algorithm optimized for runtime). Can we obtain the same IO efficiency results without trading off runtime?

Acknowledgments

We would like to acknowledge helpful discussions with Elette Boyle, Hubert Chan, Kai-Min Chung, Yue Guo, Robert Kleinberg, Ilan Komargodski, Dexter Kozen, Bruce Maggs, Moni Naor, and Rafael Pass. We appreciate the useful comments from the reviewers of STOC'18, FOCS'18, and SODA'19.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An O(N Log N) sorting network. In *STOC*, 1983.

[3] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.

[4] A. Andersson. Faster deterministic sorting and searching in linear space. In *FOCS*, 1996.

[5] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, August 1998.

[6] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[7] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Oblivious computation with data locality. Cryptology ePrint Archive, Report 2017/772, 2017. `http://eprint.iacr.org/2017/772`.

[8] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, and Elaine Shi. Optorama: Optimal oblivious ram. Cryptology ePrint Archive, Report 2018/892, 2018. `https://eprint.iacr.org/2018/892`.

[9] K. E. Batcher. Sorting Networks and Their Applications. AFIPS '68 (Spring), 1968.

[10] M. Bender, E. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341–358, January 2005.

[11] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[12] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 207–218. ACM, 2013.

[13] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973.

[14] Elette Boyle and Moni Naor. Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ITCS '16, pages 357–368, New York, NY, USA, 2016. ACM.

[15] Gerth Stolting Brodal and Rolf Fagerberg. Lower Bounds for External Memory Dictionaries. In *SODA*, 2003.

[16] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[17] L. Elisa Celis, Omer Reingold, Gil Segev, and Udi Wieder. Balls and Bins: Smaller Hash Families and Faster Evaluation. In *FOCS*, 2011.

[18] T-H. Hubert Chan, Kai-Min Chung, Bruce Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. `https://eprint.iacr.org/2017/1033.pdf`.

[19] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Asiacrypt*, 2017.

[20] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, 2018.

[21] T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying computationally and statistically secure ORAMs and OPRAMs. In *TCC*, 2017.

[22] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

[23] Ivan Bjerre Damgård. A Design Principle for Hash Functions. In *CRYPTO*, 1989.

[24] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, August 2014. USENIX Association.

[25] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[26] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[27] R.A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, 1975.

[28] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, October 1985.

[29] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[30] Robert W. Floyd. Permuting Information in Idealized Two-Level Storage. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 105–109. Springer US, 1972. DOI: 10.1007/978-1-4684-2001-2_10.

[31] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[32] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[33] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[34] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA*, 2010.

[35] Michael T. Goodrich. Data-oblivious External-memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 379–388, New York, NY, USA, 2011. ACM.

[36] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *STOC*, 2014.

[37] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[38] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[39] Michael T Goodrich and Joseph A Simons. Data-oblivious graph algorithms in outsourced external memory. In *International Conference on Combinatorial Optimization and Applications*, pages 241–257. Springer, 2014.

[40] Yijie Han. Deterministic sorting in o($n$loglog$n$) time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[41] Yijie Han and Mikkel Thorup. Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In *FOCS*, 2002.

[42] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An Optimal Algorithm for the Distinct Elements Problem. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 41–52, New York, NY, USA, 2010. ACM.

[43] David G. Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. Technical report, 1981. University of British Columbia.

[44] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[45] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[46] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

[47] Tom Leighton, Yuan Ma, and Torsten Suel. On probabilistic networks for selection, merging, and sorting. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 106–118, 1995.

[48] Tamir Levi and Ami Litman. The strongest model of computation obeying 0-1 principles. *Theory of Computing Systems*, 48(2):374–388.

[49] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, March 2015.

[50] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *IEEE S & P*, 2015.

[51] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM CCS*, 2013.

[52] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[53] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear i/o. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 723–735, London, UK, UK, 2002. Springer-Verlag.

[54] Raghu Meka, Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Fast pseudorandomness for independence and load balancing. In *ICALP*, 2014.

[55] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[56] John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In *STACS*, pages 554–565, 2014.

[57] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[58] Rasmus Pagh and Srinivasa Rao Satti. Secondary indexing in one dimension: Beyond b-trees and bitmap indexes. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 177–186, New York, NY, USA, 2009. ACM.

[59] Enoch Peserico. Deterministic oblivious distribution (and tight compaction) in linear time. *CoRR*, abs/1807.06719, 2018.

[60] Nicholas Pippenger and Leslie G. Valiant. Shifting graphs and their applications. *J. ACM*, 23(3):423–432, July 1976.

[61] Harald Prokop. *Cache-Oblivious Algorithms.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[62] Sanguthevar Rajasekaran and Sandeep Sen. Optimal and practical algorithms for sorting on the pdm. *IEEE Trans. Comput.*, 57(4):547–561, April 2008.

[63] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, pages 415–430. USENIX Association, 2015.

[64] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[65] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Asiacrypt*, 2011.

[66] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S & P*, 2013.

[67] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *ACM CCS*, 2013.

[68] Mikkel Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42(2):205–230, 2002.

[69] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *FOCS*, 1975.

[70] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, June 2001.

[71] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[72] Zhewei Wei, Ke Yi, and Qin Zhang. Dynamic external hashing: The limit of buffering. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 253–259, New York, NY, USA, 2009. ACM.

[73] Wikipedia. Cache replacement policies. `https://en.wikipedia.org/wiki/Cache_replacement_policies`.

[74] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM CCS*, 2012.

[75] Ke Yi and Qin Zhang. On the Cell Probe Complexity of Dynamic Membership. In *SODA*, 2010.

Appendices

A Additional Preliminaries

A.1 Probabilistic Circuits for Sorting in the Indivisible Model

Both our upper bound and lower bound results have natural interpretations for probabilistic circuit families that realizes sort.

Consider a family of circuits $\mathcal{C}$ for sorting. Each circuit is allowed to have two types of gates: 1) arbitrary *boolean gates* with constant fan-in and constant fan-out; and 2) selector gates, each of which takes in a bit b and two balls, and outputs one of the balls.

Definition A.1. (Probabilistic circuit family for sorting in the indivisible model). *We say that the circuit family $\mathcal{C}$ is an (n, K, ϵ)-sorting circuit family in the indivisible model, iff for any input $\vec{X}$ containing n balls each assigned with a key from the domain $[K]$,*

$$\Pr_{C \leftarrow_\$ \mathcal{C}}\left[C \text{ correctly sorts } \vec{X}\right] \geq 1 - \epsilon.$$

Metrics for probabilistic sorting circuits. Given a probabilistic circuit family $\mathcal{C}$, we define the following metrics where S is referred to as the circuit complexity of $\mathcal{C}$ and S_{sel} is referred to as the selector complexity which accounts for the number of atomic operations on opaque balls.

$$S(\mathcal{C}) := \max_{C \in \mathcal{C}} (\# \text{ gates in } C)$$
$$S_{\text{sel}}(\mathcal{C}) := \max_{C \in \mathcal{C}} (\# \text{ selection gates in } C)$$

Although in general, a sorting algorithm in the Oblivious RAM model may not have an efficient probabilistic-circuit realization, for our concrete constructions this is true. As explained later, our upper bound results also imply the existence of an $(n, K, \mathsf{negl}(n))$-sorting circuit family for $K = 2^{o(\log n)}$ and a suitable negligible function $\mathsf{negl}(\cdot)$, with $o(n \log n)$ selector-gates, and $o(k \cdot n \log n)$ circuit complexity where $k := \log K$ denotes the word-width for storing a from the domain $[K]$ — the $O(k)$ factor arises from implementing word-level addition, subtraction, and comparison operations with $O(k)$ boolean gates.

On the other hand, our lower bounds are stated in terms of the number of ball movements incurred in the Oblivious RAM model — thus our lower bound results immediately imply lower bounds in terms of the number of selector-gates for probabilistic sorting circuit families.

A.2 Standard Assumptions of External-Memory Model

Cache associativity and replacement policy. The design of the cache can affect the IO-cost of an external-memory algorithm. In the *fully-associative* model, each cache-line from the memory can be placed in any of the $\frac{M}{B}$ slots in the cache. In an r-way associative model, the cache is divided into clusters each containing r cache-lines, and any cache-line can be placed in only one cluster (but can be placed anywhere within that cluster).

If there is no valid slot in the relevant cluster (or the entire cache in the case of full associativity), some cache-line will be evicted from the cluster back to the memory to make space — which cache-line is evicted is decided by what we call a "replacement policy". Common replacement policies in practical systems include Least Recently Used (LRU) and First-In-First-Out (FIFO) [25, 73].

Ideal cache assumptions and justifications. The IO-cost of external-memory algorithms (including cache-agnostic algorithms) depend on the design of the cache, including its associativity and replacement policy. Throughout this paper, we adopt the standard practice in the literature [6, 10, 25, 31, 61] and analyze our algorithms assuming an "ideal cache" that adopts an optimal replacement policy and is fully associative. It is important to justify why these assumptions extend to realistic storage architectures, despite the fact that realistic storage architectures are not "ideal". These justifications are standard and well-accepted by the algorithms community [6, 10, 25, 31, 61]. Specifically, Frigo et al. [31, 61] justify the ideal-cache model by proving that ideal-cache algorithms can be simulated on realistic storage hierarchies with degraded runtime — but the slowdown is only a constant factor in expectation. Henceforth, we omit these justifications.

B Oblivious Sorting Lower Bounds in the Indivisible Model

In this section, we prove lower bounds for oblivious sorting in the "indivisible" model. In this model, we would like to obliviously sort n opaque balls each carrying a k-bit key by the relative order of the keys. Our lower bounds work for a probabilistic Oblivious RAM model with the following assumptions:

- There are $O(1)$ number of CPU registers;
- The CPU is allowed to perform arbitrary computation on the keys (and moreover such computation is for free in our lower bound which makes our lower bound stronger);
- Whenever the CPU visits a memory location, it may (but does not have to) read the ball in the memory location into some CPU register, and/or write a ball from some (possibly different) CPU register into the memory location. Each such memory operation will count towards cost in our lower bound.

B.1 Preliminaries on Routing Graph Complexity

Let $G = (V, E)$ be a directed acyclic graph. Suppose that G has n inputs and n outputs every an input node is one that has in-degree 0 and an output node is one with out-degree 0. We say that A is an assignment from I to O iff A is a bijection from nodes in I to nodes in O. Henceforth, we also say that G is a routing graph from I to O. Pippenger and Valiant proved the following useful theorem [60].

Definition B.1. (Routing graph "implementing" some assignment). *Let G be a routing graph from I to O where $|I| = |O| = n$, and let A be an assignment from I to O. We say that G* implements *an assignment A iff there exist n vertex-disjoint paths from I to O.*

Theorem B.1 (Pippenger and Valiant [60])**.** *Let $\mathbf{A} := (A_1, A_2, \ldots, A_K)$ denote a set of assignments from I to O where $|I| = |O| = n$, such that each input in I is assigned to K different outputs in O by the K assignments in $\mathbf{A}$. Let G be a graph that implements every A_i for $i \in [K]$. It holds that the number of edges in G must be at least $3n \log_3 K$.*

In our lower bound proof, we shall make use of this theorem to reason about access pattern graphs of an Oblivious RAM machine.

B.2 Limits of Oblivious Sorting in the Indivisible Model

Theorem B.2. (Oblivious sorting k-bit keys in the indivisible model must take $\Omega(nk)$ time for any $k = O(\log n)$). *Fix any integer $k = O(\log n)$. Any (possibly probabilistic) oblivious sorting algorithm in the indivisible model which correctly sorts any input sequence n balls each carrying a k-bit key must incur at least $\Omega(nk)$ expected runtime. Further, this lower bound holds even if for any input, the algorithm is allowed to err with probability $O(1)$.*

Proof. For simplicity, we first prove the following: assuming that the algorithm $\mathcal{S}$ achieves perfect correctness for any input sequence, then it must incur $\Omega(nk)$ runtime with probability 1. Without loss of generality, henceforth we may assume a RAM with two CPU registers — but our proof easily extends to RAMs with $O(1)$ CPU registers since clearly any such $O(1)$-register RAM can be simulated by a 2-register RAM with constant blowup in runtime. Without loss of generality, assume n is a power of 2, $k \leq \log n$, and hence $K = 2^k$ divides n.

Access pattern as a routing graph. Imagine that there are n balls in memory, and consider any fixed initial key assignment $\vec{X} \in [K]^n$. Recall that the execution of the algorithm $\mathcal{S}$ may be probabilistic, and thus we consider a specific sample path for input $\vec{X}$ that occurs with non-zero probability. Let G be the sequence of physical access patterns observable in this sample path. Henceforth we will think of G as a routing graph in which each node is labeled by one or more pairs of the form (a, t) where a denotes the (physical) memory address and t denotes the time step in which this node is created. We assume w.l.o.g. that at every time step, the CPU reads one memory address, then writes the same memory address, and there is at most 1 occupied register at the end of each step. Henceforth for convenience we will assume that the occupied CPU register resides at memory address 0 and all other memory locations reside at addresses 1 or greater. In this graph G, there are n input nodes corresponding to the n balls initially in memory each labeled with $(a, 0)$ for $a \in [n]$, and there are n output nodes corresponding to the sorted array at the end of the algorithm $\mathcal{S}$ each labeled with (a, T) where $a \in [n]$ and T denotes the runtime of $\mathcal{S}$ in this sample path — note that without loss of generality, we may assume that the output array is copied to addresses $1..n$. In every time step $t \in [T]$ of the execution (recall that the CPU accesses exactly one memory location in every time step):

- Create a new node $(0, t)$ denoting a copy of the CPU's register at the end of this time step;
- If the CPU accesses some memory location a during this step (there is exactly one such a), then we create also a node (a, t); further, we draw the following directed edges:

$$\begin{aligned} &(0, t-1) \to (0, t), \quad (0, t-1) \to (a, t), \\ &(a, t-1) \to (a, t), \quad (a, t-1) \to (0, t) \end{aligned}$$

- If the CPU does not access a memory location a during this step, then we henceforth use notation (a, t) as an alias for $(a, t-1)$, i.e., without actually creating a new node called (a, t).

We stress that the same node may have multiple labels of the form (a, t), since if the node (corresponding to some memory address) did not get accessed in some time step, there is no need to create a new copy of this address.

Since the algorithm has perfect obliviousness, if G is encountered with non-zero probability for some input (or initial key assignment) $\vec{X} \in [K]^n$ it must be encountered with non-zero probability for every input $\vec{X}' \in [K]^n$. Now since the algorithm also has perfect correctness and is in the indivisible model, it must be that for every input $\vec{X}' \in [K]^n$, there exists a permutation of $\vec{X}'$ that makes it sorted, and G implements the assignment naturally implied by this permutation — henceforth if the above is satisfied for some G and some input $\vec{X}' \in [K]^n$, we say that G *explains* the input $\vec{X}'$. More intuitively, G explains $\vec{X}'$ if subject to the access pattern G, there is a way to route input balls to output positions such that the output becomes sorted. We thus have the following fact:

Fact 1. *For an algorithm that has perfect correctness and perfect obliviousness, it must be that if G is enountered with non-zero probability for some input $\vec{X} \in [K]^n$ then G must explain every input $\vec{X}' \in [K]^n$.*

Complexity of G. We now prove that an access pattern G encountered with non-zero probability must be sufficiently large in size.

Lemma B.1. *Let $I = \{(i, 0) : i \in [n]\}$ and $O = \{(i, T) : i \in [n]\}$ be input and output nodes of G. Suppose that an access pattern G can explain any input sequence $\vec{X}' \in [K]^n$. Then, there exists a set of assignments $\mathbf{A} := (A_1, A_2, \ldots, A_K)$ from I to O such that G implements every $A_s \in \mathbf{A}$, and every input in I is assigned to a distinct output in O in each of the K assignments in $\mathbf{A}$.*

Proof. The goal is to identify a set of K inputs denoted $\{\vec{X}_s\}_{s \in [K]}$, and show that every $\vec{X}_s$ gives rise to some assignment A_s that G implements, such that the assignments $\{A_s\}_{s \in [K]}$ satisfy the distinctness condition (i.e., every input is mapped to a distinct output in each A_s).

Intuitively, we consider n/K groups each of size K, and every group takes on a distinct key from $[K]$. We now form the K inputs by each time shifting these K groups by one group. More formally, for any $z \in \mathbb{N}$, let $[z]_K$ be the smallest non-negative integer that equals to $z \mod K$. Let $\vec{X}_s$ be the input sequence $(a_0, a_1, \ldots, a_{n-1})$, where a_i is tagged with the key $[\left\lfloor \frac{i}{n/K} \right\rfloor + s]_K$; that is, $\vec{X}_s$ consists of K groups of size n/K, each group is tagged with a unique key in $[K]$, and the groups are shifted by s. Now since G can explain $\vec{X}_s$, it holds that there must exist a permutation of $\vec{X}_s$ that makes it sorted, and G can implement the natural assignment A_s defined by this permutation.

It remains to show that the set of assignments $\{A_s\}_{s\in[K]}$ must satisfy the aforementioned distinctness condition — this is straightforward by the definition of these assignments. □

By Lemma B.1, the assignments $\mathbf{A} := (A_1, A_2, \ldots, A_K)$ satisfy the premise of Theorem B.1, and thus we conclude that the graph G must have at least $3n \log_3 K$ edges. Observe that each time step of $\mathcal{S}$ incurs at most 4 edges in the construction of G. Therefore it must be that the runtime of this sample path $T \geq \frac{3}{4} n \log_3 K = \Omega(nk)$. Since this must hold for any sample path of non-zero probability given any fixed input sequence, it holds that with probability 1, the program's execution time is $\Omega(nk)$.

The case of imperfect correctness. It suffices to describe how to extend this proof for the case when, given any input sequence, the algorithm $\mathcal{S}$ succeeds with $p = O(1)$ probability. Without loss of generality, we assume that $p = \frac{1}{2}$ since we can easily readjust the parameters of the proof for other constants. Since $\mathcal{S}$ has perfect obliviousness, we have that for any input sequence the distribution of the access pattern graph G is the same — and let $\mathcal{D}$ denote this distribution on G.

Let $\mathbf{X} = \{\vec{X}_s\}_{s\in[K]}$ denote the set of K input sequences defined in Lemma B.1. Since for any input $\vec{X}$, $\mathcal{S}$ succeeds with at least p probability, we have that

$$\forall \vec{X} : \quad \Pr_{G \leftarrow_\$ \mathcal{D}} \left[G \text{ can explain } \vec{X} \right] \geq \frac{1}{2}$$

which implies that

$$\mathbf{E}_{\vec{X} \leftarrow_\$ \mathbf{X}} \mathbf{E}_G \left[I(G, \vec{X}) \right] \geq \frac{1}{2} \tag{2}$$

where $I(G, \vec{X})$ is the indicator function denoting whether G can explain $\vec{X}$. It suffices to prove that

$$\Pr_{G \leftarrow_\$ \mathcal{D}} \left[G \text{ can explain at least } \frac{1}{4} \text{ fraction of } \mathbf{X} \right] \geq \frac{1}{10}.$$

For the sake of contradiction, suppose that

$$\Pr_{G \leftarrow_\$ \mathcal{D}} \left[G \text{ can explain at least } \frac{1}{4} \text{ fraction of } \mathbf{X} \right] < \frac{1}{10}.$$

We have that

$$\mathbf{E}_G \mathbf{E}_{\vec{X} \leftarrow_\$ \mathbf{X}} \left[I(G, \vec{X})) \right] \leq \frac{1}{10} \cdot 1 + \frac{9}{10} \cdot \frac{1}{4} < \frac{1}{2}$$

This contradicts Equation (2). □

B.3 Limits of Stability

As mentioned, our oblivious sorting algorithm is not stable. As the following theorem states, this is in fact inherent since one cannot hope to achieve $o(n \log n)$-time *stable* oblivious sort even for 1-bit keys in the indivisible model.

Theorem B.3. (Stable oblivious sorting of even 1-bit keys in the indivisible model must consume $\Omega(n \log n)$ runtime). *Any (possibly probabilistic) oblivious sorting algorithm in the indivisible model which correctly and stably sorts any input sequence of n balls each carrying 1-bit key must incur at least $\Omega(n \log n)$ expected runtime. Further, this lower bound holds even if for any input, the algorithm is allowed to err with probability $O(1)$.*

Proof. The proof proceeds in almost identical manner as Theorem B.2. The only modification necessary is that we now have to prove an equivalent of Lemma B.1 for the case of stable, 1-bit sorting. This is not difficult to construct by considering the following subset of up to $n-1$ number of 0-1 input sequences: in the i-th such sequence where $i \in [n]$, the last i balls are marked with 0 and the remaining balls are marked with 1. We note that this claim no longer holds if the sort is not required to be stable (and that is why our Partition algorithm, which is not stable, can possibly work). □

B.4 Implications for Probabilistic Sorting Circuits in the Indivisible Model.

Note that a lower bound in such a probabilistic Oblivious RAM model immediately implies a lower bound in a probabilistic circuit model as described in Section A.1. We thus conclude with the following immediate corollaries.

Corollary B.1. (Lower bound for probabilistic sorting circuits). *Let $0 < \epsilon < 1$ be a constant and $K = O(n)$; for any (n, K, ϵ)-sorting circuit family $\mathcal{C}$, it must be that $S_{\mathrm{sel}}(\mathcal{C}) \geq \Omega(n \log K)$.*

We may now define a family of circuits for stably sorting 1-bit keys in the indivisible model in a similar fashion as Definition A.1. We also obtain the following immediate corollary for the selector-complexity of any circuit family that stably sorts 1-bit keys.

Corollary B.2. (Lower bound for probabilistic 1-bit stable sorting circuits). *For any constant $0 < \epsilon < 1$, for any circuit family $\mathcal{C}$ that stably sorts n balls with 1-bit keys with correctness error ϵ, it must be that $S_{\mathrm{sel}}(\mathcal{C}) \geq \Omega(n \log n)$.*

C New Building Blocks

C.1 Deterministic Partitioning

In this section, we describe a *deterministic* algorithm in the cache-agnostic model for performing partitioning of 1-bit keys. As mentioned earlier, this can be used as a building block in some of our algorithms and to improve the runtime of the FunnelOSort algorithm of Chan et al. [20] by a $\log \log n$ factor.

C.1.1 Intuition

Given an input array containing n balls each tagged with a 1-bit key, the algorithm DeterministicPart performs following steps:

- Divide the array into $\sqrt{n}$ blocks each of size $\sqrt{n}$.
- Recurse on each $\sqrt{n}$-sized block to partition each block using our DeterministicPart algorithm itself — let $A_1, A_2, \ldots, A_{\sqrt{n}}$ denote the outcome blocks.

- Let $(P, M) \leftarrow \mathsf{PurifyHalf}(B_0, B_1)$ be an algorithm that upon receiving two sorted blocks B_0 and B_1, outputs two blocks P and M such that $P \cup M = B_0 \cup B_1$, and moreover, 1) P must be *pure*, i.e., it contains either all 0s or all 1s; and 2) M is sorted. Notice that from $B_0 \cup B_1$ it is guaranteed that we can extract at least one pure block.

- Now leverage this building block PurifyHalf such that we emit $\sqrt{n} - 1$ pure blocks and at most 1 unpure block. To achieve this, do the following steps. Let $M = A_1$. For $i = 2$ to $\sqrt{n}$, let $(P_{i-1}, M) \leftarrow \mathsf{PurifyHalf}(M, A_i)$. At this point, all of $P_1, \ldots, P_{\sqrt{n}} - 1$ are pure, and the M block is the only unpure block.

- Since the $\sqrt{n} - 1$ blocks $P_1, \ldots, P_{\sqrt{n}-1}$ are pure, we can write each of these pure blocks as the row of a matrix. We now recursively call DeterministicPart itself to sort all rows of this matrix — let the outcome of this step be A'.

- Finally, use an efficient procedure called PartBitonic to combine the sorted outcome M with the earlier outcome A' into a single sorted array and output the result.

  The PartBitonic algorithm realizes the following abstraction: given a bitonically partitioned (i.e., bitonically sorted) array, it outputs a fully partitioned (i.e., sorted) array. Here we say that an array is bitonically partitioned iff either all the 0s are at the beginning or the end; or all the 1s are at either the beginning or the end of the array.

To fully instantiate the above algorithm, we also need to instantiate the building blocks PurifyHalf and PartBitonic. It turns out that it is not difficult to construct an IO-efficient algorithm for these building blocks in the cache-agnostic model — and in fact in our instantiation later, PurifyHalf in turn relies on PartBitonic as a building block. The details are described in the full algorithm description in the next subsection.

C.1.2 Algorithm

We use capital letters $A, B, C, \ldots$ to denote arrays, subscripts A_i denote the i-th subarray of array$A$, and $A[i]$ denotes the i-th element of array A, where indexes start from 1. In addition, $A||B$ denotes the concatenation of two arrays A, B, and $|A|$ denotes the number of elements in A. To move elements, the following oblivious operations and their corresponding dummy operations have identical memory-access pattern. Henceforth, we use them in *if-else* branches without specifically stating their dummy counterparts.

- To (or not to) *oblivious compare-and-swap* two elements, take two elements as input, compare two keys, and swap two elements if the first key is greater than the second one.

- To (or not to) *oblivious swap* two consecutive subarrays A_1, A_2 possibly of different lengths, copy $A_2||A_1$ to a scratch array B and then copy B back and overwrites A_1, A_2. By copying, the access pattern depends only on the total length of $A_1||A_2$.

- The standard array Reverse algorithm is oblivious.

We say an array A of n elements is *partitioned* if for all $i < j \in [n]$, $A[i] \leq A[j]$. Also, an array A is *pure* if all elements $A[i] \in A$ have the same key. We say an array A of n elements is *bitonically partitioned* iff (a) all the 0s are either at the beginning or the end; or (b) all the 1s are either at the beginning or the end of the array.

Our IO-efficient deterministic partitioning algorithm is described in Algorithms 4, 5, and 6.

Algorithm 4 Deterministic Partition

procedure DeterministicPart(A) *// The input A is an array of n elements*
    if $n \leq 2$ **then**
        If $n = 1$, **return** A. Else, **return** the result of oblivious compare-and-swap $A[0], A[1]$.
    Parse A as subarrays of size q, denoted as $A_1, \ldots A_p$, where $q = \lfloor \sqrt{n} \rfloor$ and $p = \lfloor n/q \rfloor$. Parse the remainder as A_{p+1}.
    $T \leftarrow$ DeterministicPart(A_1)
    for i from 2 to p **do**
        $T' \leftarrow$ DeterministicPart(A_i)
        $(B_{i-1}, T) \leftarrow$ PurifyHalf(T, T') *// Put majority to B_{i-1} and the remaining to T*
    Let $B_{p+1} \leftarrow$ DeterministicPart(A_{p+1}). Let $C_p \leftarrow$ PartBitonic($T||$Reverse(B_{p+1})).
    Let k_i be the key of the pieces B_i. Transpose the memory layout of $\{B_i\}_{i \in [p-1]}$ such that for every $j \in [q]$, the elements $\{B_i[j]\}_{i \in [p-1]}$ are packed contiguously in memory.
    Sort each pack $\{B_i[j]\}_{i \in [p-1]}$ by $\{k_i\}_{i \in [p-1]}$ using DeterministicPart.
    Transpose back the memory layout, let the result be C.
    return PartBitonic($C||$Reverse(C_p)).

Algorithm 5 Purify Half

procedure PurifyHalf(A, B) *// A, B are arrays such that $|A| = |B| = n$ and both A, B are sorted.*
    Count the number of 0 and 1 keys in all elements of $A||B$. Let b be the majority.
    if b is 1 **then** oblivious Reverse both A, B.
    Parse A as $A_1||A_2$, B as $B_1||B_2$, where $|A_i| = |B_i| = n/2$. Let $C \leftarrow A_1||B_1$.
    return (C, PartBitonic($A_2||$Reverse(B_2))).

Algorithm 6 Partition a Bitionically Partitioned Array

procedure PartBitonic(A) *// A is a bitonically partitioned array of n elements.*
    if $n = 1$ **then return** A.
    if n is odd **then** let n' be $n-1$, A' be the subarray of A with first $n-1$ elements
    else let n' be n, A' be A.
    Parse A' as $A_1||A_2$, where $|A_1| = |A_2| = n'/2$.
    for i from 0 to $n'/2 - 1$ **do**
        Oblivious compare-and-swap $A_1[i], A_2[i]$ iff $k(A_2[i]) < k(A_1[i])$.
    Scan A_1, get the maximum and minimum keys (m_1, n_1). Also get (m_2, n_2) from A_2 similarly.
    if $m_1 - n_1 < m_2 - n_2$ **then**
        Oblivious swap A_1, A_2, $A_1 \leftarrow$ PartBitonic(A_1), oblivious swap A_1, A_2.
    else $A_1 \leftarrow$ PartBitonic(A_1) *// Perform dummy swap before and after this operation*
    Let B be $A_1||A_2$.
    if n is odd **then** insert $A[n]$ to B by oblivious compare-and-swap all elements in B.
    return B.

C.1.3 Analysis

Correctness. We first argue that PartBitonic correctly outputs a partitioned array by induction. Observe the for-loop at Line 6 yields two bitonically partitioned arrays A_1 and A_2. Afterwards, either A_1 is pure with all 0s or A_2 is pure with all 1s, and we find it by the maximum and minimum keys at Line 8. The correctness follows by the induction hypothesis that the impure subarray is then sorted by PartBitonic recursively.

To see DeterministicPart is correct for all $|A| = n$, assume by induction that it is correct for all A' such that $|A'| < n$. Note that, after the loop at Line 6, each B_i is pure for all $1 \le i < p - 1$ by induction hypothesis. Then, both C and C_p are also sorted by induction. It follows by the correctness of PartBitonic and $C||\mathsf{Reverse}(C_p)$ is bitonically partitioned.

Running time. The running time of PartBitonic is $O(n)$ as it performs linear operations and recurses on the subproblem of half length. Then, denote the running time of DeterministicPart as $T(n)$. There are $2\sqrt{n}$ recursive calls with at most $\sqrt{n}$ elements, and other procedures are linear time. Hence, for recursion depth k, there are total $2^k n^{1-2^{-k}}$ instances, and each instance costs $O(n^{2^{-k}})$ time. The total cost is the sum of all $\log\log n$ depths,

$$T(n) = \sum_{k=0}^{\log\log n} 2^k n = O(n \log n).$$

IO-cost. Recall that M is the cache size, B is the size of a cache line. Observe that PartBitonic takes $O(\lceil n/B \rceil)$ IO by the argument similar to the running time. The IO-cost of DeterministicPart is $O(\lceil \frac{n}{B} \rceil \log_M n)$ by solving the summation

$$C(n) = \sum_{k=0}^{\log\log n - \log\log M} 2^k \lceil \frac{n}{B} \rceil = O(\lceil \frac{n}{B} \rceil \log_M n).$$

Lemma C.1. DeterministicPart *is a cache-agnostic and oblivious algorithm such that correctly sorts 1-bit keys in time $O(n \log n)$ and IO-cost $O(\lceil \frac{n}{B} \rceil \log_M n)$.*

C.2 Move Stripe to Working Buffer

Given a matrix A written down in a row-major manner in memory, a *stripe* is defined as a set of consecutive rows from l to r. The following MoveStripe algorithm can copy all elements in the rows from l to r to a working buffer of equivalent size (in any order), without revealing the location of the stripe.

Algorithm 7 MoveStripe

procedure MoveStripe(A, l, r) *// This function is an oblivious and cache-agnostic procedure, it will move items in the interval $[l, r)$ to a temp memory.*
  Let M be the temporary memory to hold the mixed stripe.
  Parse A as $Z \times \frac{n}{Z}$ matrix, where $Z = \log^6 \lambda$.
  Move ptr to the first row of M.
  for i from 0 to Z **do**
    if $i \in [l, r)$ **then** *// inside the region*
      $M[\mathsf{ptr}] \leftarrow A[i]$, where $X[t]$ denotes the t-th row of matrix X.
    Move ptr to next row. If it reaches the end of M, move it to the beginning.
  return M

D Sorting Arbitrary-Length but Few Distinct Keys

So far, our algorithms assumed that the key is short in order to overcome the $n \log n$ barrier. In this section, we show how to relax this assumption on the key length as follows. In Section D.1, we present a simple building block to count few number of distinct keys, and then, using k-wise ϵ-independent hash family, we extend the counting algorithm to estimate the number of distinct keys in Section D.2. Then, in Section D.3, we show how to overcome the $n \log n$ barrier for arbitrary-length keys but assuming that the number of distinct keys is $2^{o(\log n)}$.

Assumptions. In this section, we shall assume that the RAM's word size is $\Theta(\log n)$ bits, and that each key may be large and require L words to store. We assume that word-level addition, subtraction, comparison, and *multiplication* operations can be computed in unit cost (*cf.* in earlier sections, we did not need unit-cost word-level multiplication).

D.1 Counting Few Number of Distinct Keys

If an input array contains only polylogarithmically many distinct keys, we can count the number of distinct keys in $O(n \log \log \lambda)$ time as described in the following algorithm. The idea is to maintain a working buffer whose size $\mathsf{bucketSize}$ is polylogarithmic. Each time we read in the next batch of polylogarithmically many elements, union it with the working buffer, and obliviously sort to move elements of the same key together. Then in one linear scan, for each unique key, we mark only the first occurrence as distinguished and the remaining as dummy. With another oblivious sort, we can extract the first up to $\mathsf{bucketSize}$ elements with distinct keys (and if there are not enough the result is padded to $\mathsf{bucketSize}$ with dummies).

Finally, the algorithm either outputs FAIL if the working buffer is full or it outputs the number of distinct, non-dummy elements in the working buffer.

Algorithm 8 Cardinality estimation for small cardinality

---

procedure $\mathsf{FewDistinct}[\mathsf{bucketSize}](A)$ *// Input A is an array. This procedure outputs the number of distinct keys of A correctly if A contains less than* $\mathsf{bucketSize}$ *elements. Otherwise it outputs FAIL.*

  Initialize two empty arrays B, C of length $\mathsf{bucketSize}$ elements.

  for i from 1 to $\left\lceil \frac{|A|}{\mathsf{bucketSize}} \right\rceil$ **do**

    Copy from A to C the i-th sub-array of length $\mathsf{bucketSize}$.

    Concatenate arrays B and C, eliminate duplicate elements by sorting, output smallest distinct elements to B. (Both B and C are padded with dummy elements to achieve obliviousness).

  Let count be the number of distinct elements in B.

  if $\mathsf{count} = \mathsf{bucketSize}$ **then return** *FAIL* **else return** count

---

We mainly use $\mathsf{FewDistinct}$ with $\mathsf{BucketSize} = \log^5 \lambda$. When instantiating $\mathsf{FewDistinct}[\log^5 \lambda]$ using $\mathsf{ZigZagSort}$, it runs in time $O(n \log \log \lambda)$. To be IO-efficient, $\mathsf{FewDistinct}[\log^5 \lambda]$ is instantiated with $\mathsf{FunnelOSort}$, which runs in IO-cost $O(\lceil \frac{n}{B} \rceil \log_{M/B} \frac{\log^5 \lambda}{B})$ but time $O(n \log^2 \log \lambda)$.

D.2 Counting Distinct Keys

Recall that in our earlier algorithms, we needed the short-key assumption because we used the expression (maximum key $-$ minimum key $+$ 1) to estimate the number of distinct keys in each piece. In this section, we will propose a new oblivious algorithm for estimating the number of distinct keys accurate up to a constant factor (except with negligible probability).

As mentioned, we assume each key is of L-words such that $L \leq \mathsf{poly}(\lambda)$ (rather than 1-word as in previous sections). We use the notation $F_0(A)$ (or F_0 for short) to denote the number of distinct keys in the array A as in the literature [28,42]. Our algorithm estimates F_0 in $O(n\alpha L + n\alpha^2 \log\log\lambda)$ time with negligible failure probability assuming $n := |A| \leq \mathsf{poly}(\lambda)$.

D.2.1 Preliminaries

We define the problem of cardinality estimation (i.e., estimating the number of distinct keys given an input array) as follows.

Definition D.1. (Cardinality estimation problem). *Given an array of elements $x_1, x_2, ..., x_n$ with repetitions, we use the notation $F_0 = |\{x_1, x_2, ..., x_n\}|$ to denote the number of distinct elements in the array. For constants $a, b > 0$, we say $\tilde{F_0}$ is an* $[a, b]$-estimation *of F_0 iff $aF_0 \leq \tilde{F_0} \leq bF_0$.*

k-wise ϵ-independent hash family. We will leverage a k-wise ϵ-independent hash family to sub-sample keys (Meka et al. [54]), and the tail bound of k-wise ϵ-almost independent variables (Celis et al. [17]) is needed in the analysis.

Definition D.2. (k-wise ϵ-almost independent hash function). *A hash family $\mathcal{H} = \{h : \{0,1\}^p \to \{0,1\}^q\}$ is said to be k-wise ϵ-almost independent if for any k fixed inputs, their (joint) output distribution is ϵ-close to uniform (in statistical distance, where the probability is over the choice of a function from the family).*

Lemma D.1 (Construction 1, [54])**.** *Let $k > 1$, $2^{\ell_1} \geq k\ell_2/\epsilon$, and $\mathbb{F}$ be the field $GF(2^{\ell_1})$. Then, there exists a k-wise ϵ-almost independent hash family $\mathcal{H} = \{h : \mathbb{F} \to \{0,1\}^{\ell_2}\}$ such that every $h \in \mathcal{H}$ can be evaluated in $O(\log(k\ell_2))$ field operations, where each field operation is either an addition or a multiplication in $\mathbb{F}$, and the seed of h is two elements sampled at random from $\mathbb{F}$.*

Lemma D.2 (Lemma 2.2, [17])**.** *Let $X_1, \ldots, X_m \in \{0,1\}$ be k-wise ϵ-almost independent random variables for some $k/2 \in \mathbb{N}$ and $0 \leq \epsilon < 1$. Let $X = \sum_{i=1}^m X_i$ and $\mu = \mathbf{E}[X]$. Then, for any $t > 0$, it holds that*

$$\Pr[|X - \mu| > t] \leq 2\left(\frac{mk}{t^2}\right)^{k/2} + \epsilon\left(\frac{m}{t}\right)^k.$$

Counting few distinct elements. As described in Appendix D.1, the algorithm $\mathsf{FewDistinct}$ counts the number of distinct elements in an array efficiently and obliviously.

Lemma D.3. *For all $m, n \in \mathbb{N}$, let A be an array of length n such that consists of at most $m-1$ distinct elements. The algorithm $\mathsf{FewDistinct}[m]$ is oblivious and computes $F_0(A)$ in time $O(n \log m)$ for all A.*

Collision-resistant compression family. Let constants $t, p \in \mathbb{N}$. We say a family of function $\mathcal{G} := \{g : \{0,1\}^{tp} \to \{0,1\}^p\}$ is a *collision-resistant compression* iff for all $x \neq y \in \{0,1\}^{tp}$,

$$\Pr_{g \in \mathcal{G}}[g(x) = g(y)] \leq t2^{-p}.$$

The following is a family of functions constructed from the pairwise independent hash family $\mathcal{H}_2 = \{h : \{0,1\}^{2p} \to \{0,1\}^p\}$ using the Merkle-Damgård construction [23, 55].

- For $t > 1$, define a family of functions $\mathcal{G} := \{g_{h_1,\ldots,h_{t-1}} : \{0,1\}^{tp} \to \{0,1\}^p \mid h_1,\ldots,h_{t-1} \in \mathcal{H}_2\}$, where
$$g_{h_1,\ldots,h_{t-1}}(x) := h_{t-1}(h_{t-2}(\ldots h_1(x_1||x_2)||\ldots x_{t-1})||x_t),$$
where x is parsed as $x_1||x_2||\ldots x_t$ and $x_i \in \{0,1\}^p$ for all $i \in [t]$.

Lemma D.4. *The family $\mathcal{G}$ is collision-resistant compression family.*

Proof. Fix any pair $x \neq y \in \{0,1\}^{tp}$, there exists $i \in [t]$ such that $x_i \neq y_i$ and for all $j \in [i+1,t]$, $x_j = y_j$. If $g(x) = g(y)$, then there exists $i' \in [i,t]$ such that

$$\begin{aligned} &h_{i'-1}(h_{i'-2}(\ldots h_1(x_1||x_2)\ldots)||x_{i'}) = \\ &\qquad h_{i'-1}(h_{i'-2}(\ldots h_1(y_1||y_2)\ldots)||y_{i'}) \end{aligned}$$

but $h_{i'-2}(\ldots h_1(x_1||x_2)\ldots)||x_{i'} \neq h_{i'-2}(\ldots h_1(y_1||y_2)\ldots)||y_{i'}$. That is, i' is the first collision of h after x_i and y_i. By union bound, $i \leq t$, and $\mathcal{H}_2$ is pairwise independent, the collision probability is at most $t2^{-p}$. $\square$

A standard instantiation of the pairwise hash family $\mathcal{H}_2$ was described by Carter and Wegman [16]. Hence, in the RAM model, to compress a long input of L words, it takes $O(L)$ words to write down one such function $g \in \mathcal{G}$, and applying g to an input is by construction oblivious and completes in $O(Lp/w)$ runtime as field multiplication, where w is the word size.

D.2.2 Algorithm for Estimating Number of Distinct Keys

To estimate F_0 of the array A (of length n), the idea is to put keys into $\log n$ bins by the hash value of each key. There exists a small bin such that consists of at most $n/\log n$ (possibly duplicated) keys, and then we can sort and count keys in such small bin in time $O(n)$. Observing that all identical keys are put into the same bin, the counting of the small bin is a good approximation of $F_0/\log n$ if F_0 is large enough and the hash is random enough. When F_0 is small, we can simply use FewDistinct (Algorithm 8) and obtain a exact number.

Algorithm 9 obliviously searches for the small bin of size at most $n/\log n$ and performs the estimation. We instantiate the k-wise ϵ-almost hash family $\mathcal{H}$ of Meka et al. (Lemma D.1), where the parameters are $k := 2\frac{\log\lambda}{\log\log\lambda}\alpha$, $\epsilon := 2^{-4\alpha\log\lambda}$, $\ell_1 := 6\alpha\log\lambda$, $\ell_2 := \log\log n$, and $\mathbb{F} = GF(2^{\ell_1})$, where $\alpha(\lambda) := \omega(1)$ is a super constant such that $1 < \alpha \leq \log\lambda$. Note that $2^{\ell_1} \geq k\ell_2/\epsilon$ holds for all $\lambda \geq 2^8$ and $\alpha > 1$. The collision-bounded compression $\mathcal{G} := \{g_{h_1,\ldots,h_{t-1}} : \{0,1\}^{tp} \to \{0,1\}^p \mid h_1,\ldots,h_{t-1} \in \mathcal{H}_2\}$ (see Lemma D.4) is instantiated with parameters $p := \ell_1$ and $t := L/p$, where $\mathcal{H}_2 := \{\{0,1\}^{2p} \to \{0,1\}^p\}$ is the standard pairwise independent hash family of Carter and Wegman [16]. The above $h \in \mathcal{H}$ and $g \in \mathcal{G}$ take space $O(\alpha)$ and $O(L)$ words respectively, and their sampling and evaluation are fast and oblivious.

D.2.3 Analysis

To show correctness, note that there exists no collision for all i, i' such that $a_i \neq a_{i'}$ but $g(a_i) = g(a_{i'})$ except with negligible probability by Lemma D.4 and union bound. Hence, it suffices to show the following holds had we generated H_i as $H_i \leftarrow h(a'_i)$ at Line 4, where $a'_i \in \{0,1\}^p$ for all i.

Lemma D.5. *There exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all A,* $\mathsf{Distinct}(\mathsf{InitHash}(A))$ *outputs a $[\frac{1}{2},\frac{3}{2}]$-estimation except with probability* $\mathsf{negl}(\lambda)$.

Algorithm 9 Cardinality estimation

procedure $\mathsf{InitHash}(A)$ *// The input A is an array of n keys, where each key $a_i \in A$ is L-word long. This procedure pre-calculates all hash values that will be used in the* $\mathsf{Distinct}$ *procedure.*

  Sample at random functions h from $\mathcal{H}$, g from $\mathcal{G}$.

  For each $a_i \in A$, let $G_i = g(a_i)$ if $L \geq 12\alpha$, $G_i = a_i$ otherwise.

  Let $H_A := \{(a_i, H_i, G_i)\}_{i\in[n]}$, where $H_i \leftarrow h(G_i)$ is $(\log\log n)$-bit for each G_i.

  return H_A.

procedure $\mathsf{Distinct}(H_A)$ *// The input $H_A := \{(a_i, H_i, G_i)\}_{i\in[n]}$ is an array of n elements that is generated by* $\mathsf{InitHash}(A)$*. It outputs an estimation of $F_0(A)$.*

  Let B_1 be the array $((H, G))_{(a,H,G)\in H_A}$, i.e., using only the latter two entries.

  for j from 1 to $\lceil \log\log n \rceil$ **do**

    $\mathsf{Partition}(B_j)$ according to $H_{i,j}$ for all $(H_i, G_i) \in B_j$, where $H_{i,j}$ is the j-th bit of H_i.

    Count the number of 0s and 1s in $(H_{i,j})_{i\in[n/2^j]}$, let b be the bit with count $\leq n/2^j$.

    Let B_{j+1} be an array of length $n/2^j$. Obliviously copy into B_{j+1} each pair $(H_i, G_i) \in B_j$ such that $H_{i,j} = b$ (and pad B_{j+1} with dummies for obliviousness).

  Let $\bar{G}$ be the array consists of second entries of $B_{\lceil \log\log n\rceil+1}$, i.e., $\bar{G} = (G)_{(H,G)\in B_{\lceil \log\log n\rceil+1}}$. Sort $\bar{G}$ using $\mathsf{FunnelOSort}$.

  Let est be the number of elements in $\bar{G}$ (using a linear scan).

  Let G' be the array $(G_i)_{(a_i,H_i,G_i)\in H_A}$.

  Let $\mathsf{ans} := \mathsf{FewDistinct}[\log^5 \lambda](G')$; return ans if $\mathsf{ans} \neq \mathit{FAIL}$; else return $\mathsf{est} \cdot \log n$.

Proof. If $F_0 < \log^5 \lambda$, then $\mathsf{Distinct}$ outputs an exact number. Otherwise, the algorithm is equivalent to distribute F_0 balls into $\log n$ bins using hash values H_i. We claim that for every bin Z, the number of balls in Z is in the range $[\frac{1}{2}\mu, \frac{3}{2}\mu]$, except with negligible probability, where $\mu = F_0/\log n$ is the expectation. We note that the bound holds for every bin, and thus how does $\mathsf{Distinct}$ choose a bin doesn't affect correctness.

To prove the claim, fix a bin Z, let $X_1, \ldots, X_{F_0}$ be random variables such that $X_j = 1$ iff the j-th distinct key hashes to bin Z. Let $X = \sum_{i=1}^{F_0}$. Note that $\{X_j\}_{j\in[F_0]}$ are ϵ-almost k-wise independent random variables, where $k = 2\frac{\log\lambda}{\log\log\lambda}\alpha$, $\epsilon = 2^{-4\alpha\log\lambda}$. By Lemma D.2, plugging in $m = F_0$, $\mu = F_0/\log n$, $t = \frac{1}{2}\mu$, we have

$$
\begin{aligned}
\Pr[|X - \mu| > t] &\leq 2\left(\frac{mk}{t^2}\right)^{k/2} + \epsilon\left(\frac{m}{t}\right)^k \\
&= 2\left(\frac{4k\log^2 n}{F_0}\right)^{k/2} + \epsilon(2\log n)^k \\
&\leq 2^{-\Omega(\alpha\log\lambda)} + 2^{-\Omega(\alpha\log\lambda)},
\end{aligned}
$$

where the last inequality follows by $F_0 \geq \log^5 \lambda$ and $n = \mathsf{poly}(\lambda)$. Choosing $\mathsf{negl}_1(\lambda)$ be the RHS, we have the claim holds for every bin except with probability $\mathsf{negl}(\lambda) := (\log n)\mathsf{negl}_1(\lambda)$ by union bound. $\square$

The time complexity of $\mathsf{InitHash}$ is dominated by evaluating g and h. Recall that $w = \Omega(\log\lambda)$ is the length of a word, and the field $\mathbb{F} = GF(2^{6\alpha\log\lambda})$ is used in both g and h, where a field operation takes time $(6\alpha\log\lambda/w)^2 = O(\alpha^2)$ as multiplication. As evaluating g takes $O(t)$ and

h takes $O(\log(k\ell_2))$ field operations, $\mathsf{InitHash}$ runs in time $O(n\alpha L + n\alpha^2 \log\log\lambda)$. The the runtime of $\mathsf{Distinct}$ depends on $\beta := \min(L, \alpha)$, which is dominated by $\mathsf{Partition}$, $\mathsf{FunnelOSort}$, and $\mathsf{FewDistinct}[\log^5 \lambda]$, which are all $O(\beta n \log\log\lambda)$. Also, both $\mathsf{InitHash}$ and $\mathsf{Distinct}$ are oblivious as the access pattern doesn't depend on A or randomness except for the input length n and security parameter λ.

Summarizing the above, we conclude with the following theorem.

Theorem D.1. *Let A an array of n keys, where each key has L words. There exists a negligible function $\mathsf{negl}(\cdot)$ such that for $n, L \le \mathsf{poly}(\lambda)$, the algorithm $\mathsf{InitHash}$ runs in time $O(n\alpha L + n\alpha^2 \log\log\lambda)$, $\mathsf{Distinct}$ runs in time $O(\min(L,\alpha) n \log\log\lambda)$, and the pair $(\mathsf{InitHash}, \mathsf{Distinct})$ is oblivious and outputs a $[\frac{1}{2}, \frac{3}{2}]$-estimation except with probability $\mathsf{negl}(\lambda)$.*

IO-cost. We now analyze the IO-efficiency of the instantiation using the IO-efficient $\mathsf{FewDistinct}$.

Corollary D.1. *Assuming tall cache and wide cache-lines, $\mathsf{InitHash}$ consumes $O(\lceil \frac{nL}{B} \rceil)$ IO-cost, and $\mathsf{Distinct}$ consumes $O(\lceil \frac{\beta n}{B} \rceil)$ IO-cost and time $O(\beta n \log^2 \log\lambda)$, where $\beta = \min(L, \alpha)$.*

Proof. By tall cache and wide cache-line assumption, we choose $\alpha \le \log^{0.5}\lambda = o(B)$. The IO-cost of $\mathsf{InitHash}$ is dominated by evaluating g on L words as each evaluation of h needs only $O(\alpha)$ words, which fits into the cache of size M.

The IO-cost follows the fact that $\mathsf{Partition}$, $\mathsf{FunnelOSort}$ the array $B_{\lceil \log\log n \rceil + 1}$ of length $\frac{n}{\log n}$, and $\mathsf{FewDistinct}$ are all $O(\lceil \frac{\beta n}{B} \rceil)$ under the tall cache and wide cache-line assumption, where β in logarithmic factors are canceled under the assumption. The time complexity follows by the dominating term of $\mathsf{FewDistinct}$, $O(\beta n \log^2 \log\lambda)$. ☐

D.3 Sorting Arbitrary-Length but Few Distinct Keys

In the previous Sort algorithm (Algorithm 3), we assumed that keys are short integers in the domain $[K]$. In this section, we relax this assumption — instead of assuming that keys are short, we assume that the number of distinct keys is small relative to n but the keys can be from a large domain. Henceforth let $\hat{K}$ denote the an upper bound on F_0, i.e., the number of distinct elements in the input — we assume that the algorithm knows $\hat{K}$. To achieve this, we rely on our earlier distinct element estimation algorithm $\mathsf{Distinct}$ — we will show that a constant-factor approximation suffices.

Detailed algorithm. Before calling $\mathsf{SortArbitrary}$ as shown in Algorithm 10, we begin by performing the following initialization procedure. Given an input array A, we begin by 1) applying the collision-resilient compression function to compress each long key; and 2) applying a k-wise, ϵ-independent hash function to each compressed key. Note that the above initialization procedure is performed *only once upfront, and the resulting compressed keys and their hash values will be used by all instances of* $\mathsf{Distinct}$ *algorithms subsequently.* Therefore, in the description of $\mathsf{SortArbitrary}$, we simply assume that each input element is already tagged with and a compressed key and its hash value.

After this initialization procedure, we proceed with the main algorithm (i.e., $\mathsf{SortArbitrary}$) as follows. As before, we will divide the input into polylogarithmically sized bins and obliviously sort each bin. We now write the bins as columns and divide the resulting matrix into pieces. We then apply the $\mathsf{Distinct}$ algorithm to estimate the number of distinct elements in each piece and obliviously sort pieces in increasing order of the estimation. To upper-bound the real number of distinct elements in each piece, the upper bound of the previous Sort is multiplied by 3 as $\mathsf{Distinct}$ outputs $[\frac{1}{2}, \frac{3}{2}]$-estimations (except with negligible probability). Now we recurse on each piece using

Algorithm 10 Sort Arbitrary-Length but Few Distinct Keys

```
1: procedure SortArbitrary(A, K̂)        // The input A is an array of
   n elements, each element is a tuple (key, H, G), where key is L-word long and F_0(A) is at most K̂, and
   H, G are the hashes generated by InitHash. Parse the array as a Z × n/Z matrix, where Z = log^6 λ.
2:     if |A| < 2Z then
3:         return ZigZagSort(A)
4:     if K̂ ≤ 2 then return Partition(A)
5:     Perform RandCyclicShift on each row of A.
6:     Sort columns: A^T ← Transpose(A). ZigZagSort each row of A^T. A ← Transpose(A^T).
7:     Let p := 2 if K̂ < 64, otherwise p := ⌊log K̂⌋. Let q := Z/p. Parse matrix A as p pieces,
       A_1, ..., A_p, where each piece A_j is a q × n/Z sub-matrix of A.
8:     for i from 1 to p − 1 do
9:         Sort crossover stripes: ZigZagSort the boundary between A_i and A_{i+1}. That is, sort
           all elements between row iq − δ and row iq + δ.
10:    for i from 1 to p do
11:        Estimate the number of distinct keys in A_i:
               K̃_i ← { FewDistinct[64](A_i)   if K̂ < 64
                      { Distinct(A_i)          otherwise.
12:    Sort all pieces {A_i}_{i∈[p]} obliviously in increasing order of K̃_i:
       {B_i}_{i∈[p]} ← ZigZagSort({A_i}_{i∈[p]}). We assume each B_i remembers its original piece index.
13:    for i from 1 to p do
14:        Sort sub-problems: { SortArbitrary(B_i, ⌈K̂/(p−i+1)⌉)                 if K̂ < 64
                              { SortArbitrary(B_i, min(3⌈K̂/(p−i+1)⌉, K̂))      otherwise.
15:    Obliviously sort {B_i}_{i∈[p]} by their original piece indexes: {C_i}_{i∈[p]} ← ZigZagSort({B_i}_{i∈[p]}).
16:    return {C_i}_{i∈[p]}.
```

this estimated distinct count (multiplied by 3). Finally, the base cases of $\hat{K} < 64$ are implemented with the exact distinct counting algorithm FewDistinct (see Appendix D.1).

In our detailed algorithm description (Algorithm 10) ZigZagSort by default sorts in increasing order of *key* if not specified.

Correctness. We first argue that in the execution of SortArbitrary, all instances of Distinct output $[\frac{1}{2}, \frac{3}{2}]$-estimation except with negligible probability. For each Distinct, the failure probability is negligible by Theorem D.1. We take *union bound* over polynomially many Distinct, and it follows the total failure probability is still negligible. Note the union bound holds even though the probabilities of two Distinct are not independent (as they depend on the same g and h, sampled once upfront).

The following variant to bound the number of distinct keys in pieces follows from a proof that is similar to Lemma F.6 — thus we state the following lemma without repeating the proof.

Lemma D.6. *Let A be an array of at most $\hat{K}$ distinct keys, $\mathbf{A} := \{A_1, \ldots, A_p\}$ be a piecewise-ordered partition of A. Then, for every $i \in [p]$, there exist at least i pieces of $A' \in \mathbf{A}$ such that A' has at most $\lceil \frac{\hat{K}}{p-i+1} \rceil$ distinct keys.*

Compared to Sort integers (Algorithm 3), the only difference is that $\mathsf{SortArbitrary}$ works on estimated values $\tilde{K}_i$ when $\hat{K} \geq 64$, and it suffices to show the relaxed sub-problem $\mathsf{SortArbitrary}(B_i, \min(3\lceil \frac{\hat{K}}{p-i+1} \rceil, \hat{K}))$ is correctly solved except with negligible probability. Suppose that A has at most $\hat{K}$ distinct keys. Condition on the *good event* such that $\{A_1, \ldots, A_p\}$ is a piecewise-ordered partitioning of A, and all $\tilde{K}_i$ are $[\frac{1}{2}, \frac{3}{2}]$-estimation of the real $F_0(A_i)$. It suffices to observe the following fact, and thus, by union bound, $\mathsf{SortArbitrary}$ is correct except with probability negligible in λ.

Fact 2. *Conditioning on the good event stated above, for any $i \in [p]$, if the real $F_0(A_i)$ ranks j among all real $\{F_0(A_t) : t \in [p]\}$ and the estimation $\tilde{K}_i$ ranks $\tilde{j}$ among all estimation $\{\tilde{K}_t : t \in [p]\}$ such that $\tilde{j} < j$, then $F_0(A_i) \leq 3\lceil \frac{\hat{K}}{p-\tilde{j}+1} \rceil$.*

Proof. Assume for contradiction that $F_0(A_i) > 3\lceil \frac{\hat{K}}{p-\tilde{j}+1} \rceil$. By $[\frac{1}{2}, \cdot]$-estimation, $\tilde{K}_i > \frac{3}{2}\lceil \frac{\hat{K}}{p-\tilde{j}+1} \rceil$. Also, as $\tilde{K}_i$ ranks $\tilde{j} < j$, by pigeon hole principle, there exists a piece A_{i^*} such that has real rank $\leq \tilde{j}$ but estimated rank $> \tilde{j}$. This cannot happen: by ranking $\leq \tilde{j}$ and Lemma D.6, it holds that $F_0(A_{i^*}) \leq \lceil \frac{\hat{K}}{p-\tilde{j}+1} \rceil$, which implies $\tilde{K}_{i^*} \leq \frac{3}{2}\lceil \frac{\hat{K}}{p-\tilde{j}+1} \rceil$ by $[\cdot, \frac{3}{2}]$-estimation; it follows $\tilde{K}_{i^*} < \tilde{K}_i$ and contradicts that the estimation $\tilde{K}_{i^*}$ ranks $> \tilde{j}$. $\square$

Running time. The overall sorting consists of one $\mathsf{InitHash}$ and one $\mathsf{SortArbitrary}$. To $\mathsf{InitHash}$, it takes $O(n\alpha L + n\alpha^2 \log\log\lambda)$ time. To $\mathsf{SortArbitrary}$, it takes $\hat{c}_1 = c_1 nL \log\log\lambda$ to perform column-wise $\mathsf{ZigZagSort}$ for some constant c_1, and then the recursion of $\mathsf{SortArbitrary}$ is

$$T(n, \hat{K}) = \sum_{i=1}^{p} T\left(\frac{n}{p}, \min(3\lceil \frac{\hat{K}}{p-i+1} \rceil, \hat{K})\right) + \hat{c}_1,$$

where the base cases are: if $|A| < 2Z$, then it is $\mathsf{ZigZagSort}$ and runs in time $O(nL\log n)$; if $\hat{K} < 64$, then it is exactly $\mathsf{SortSmall}$ and runs in time $O(n \log \hat{K} \log\log\lambda)$. The recurrence and hence the solution are identical to that of Sort (Lemma F.7) except with a larger constant. Following the same induction and induction hypothesis, we have the inequality

$$T(n, \hat{K}) \leq (cnL \log\log\lambda) \frac{\log \hat{K} - \log p + \log(3)}{\log\log(3\hat{K}/p)} + \hat{c}_1.$$

Choosing sufficiently large $c \geq c_1/0.16$ such that $cn\frac{\log \hat{K}}{\log\log \hat{K}} \log\log\lambda$ bounds both base cases, we have

$$T(n, \hat{K}) \leq O\left(nL \frac{\log \hat{K}}{\log\log \hat{K}} \log\log\lambda\right)$$

by the fact that $\frac{\log \hat{K} - \log p + \log(3)}{\log\log(3\hat{K}/p)} \leq \frac{\log \hat{K}}{\log\log \hat{K}} - 0.16$ for all $\hat{K} \geq 64$.

Summarizing the above, we conclude with the following theorem.

Theorem D.2. *Let A an array of n keys and at most $\hat{K}$ distinct keys, where each key has L words. There exists a negligible function $\mathsf{negl}(\cdot)$ for all A such that $n, \hat{K}, L \leq \mathsf{poly}(\lambda)$, the pair $(\mathsf{InitHash}, \mathsf{SortArbitrary})$ obliviously sorts A in time $O(n\alpha L + n\alpha^2 \log\log\lambda + nL\frac{\log \hat{K}}{\log\log \hat{K}} \log\log\lambda)$ except with probability $\mathsf{negl}(\lambda)$.*

Remark 2. *Compared to the earlier* Sort *algorithm in Section 5, we remark that* SortArbitrary *has a couple more advantages (even when the key space is small): 1)* Sort *requires that the subtraction operation be well-defined on the key space (w.r.t. the ordering) whereas* SortArbitrary *requires only a comparison operator on the key space (and yet* SortArbitrary *is not comparison-based sorting since it needs to evaluate hash functions over keys); and 2) even when the key space K is small, if the number of distinct keys $\hat{K}$ is much smaller than the key space K,* SortArbitrary *can be more efficient than the earlier* Sort *algorithm.*

If keys are L-word long and L can be greater than cache size M, then just comparing two keys costs L/B IOs. Hence, the IO-cost is simply runtime divided by cache-line size B in the cache-agnostic model.

E Applications to Other Open Problems

E.1 Tight Compaction

Tight compaction is the following problem: given an input array containing n real or dummy elements, output an array where all the dummy elements are moved to the end of the array[13]. Oblivious compaction is a fundamental building block — in fact, the large majority of existing oblivious algorithms [50, 57] as well as ORAM/OPRAM schemes [7, 19, 32, 33, 37, 45] have adopted compaction as a building block. Goodrich phrased an open question in his elegant paper [35]:

> *Can we achieve tight compaction in linear IO?*

As mentioned earlier, the probabilistic selection network construction by Leighton et al. [47] can easily be extended to perform tight compaction in $O(n \log\log n)$ time — however the resulting algorithm would not have good IO efficiency when implemented on a RAM machine. It is obvious that IO-efficient oblivious sorting algorithm for the 1-bit-key special case could answer Goodrich's open question as stated in the following corollary.

Corollary E.1. (Tight compaction in almost linear time and linear IO). *There exists a negligible function* $\mathsf{negl}(\cdot)$ *and a cache-agnostic, oblivious algorithm (with an explicit construction) which, except with* $\mathsf{negl}(\lambda)$ *probability, correctly achieves tight compaction over n real or dummy elements in $O(n \log\log \lambda)$ time and with $O(n/B)$ IOs assuming the cache size $M \geq \log^{1.1} \lambda$ (which is satisfied under the standard tall cache and wide cache-line assumptions).*

Goodrich stated a notion of "order-preserving" for tight compaction which is the same as the standard stability notion for sorting. Our tight compaction algorithm is not stable but our 1-bit-key stable sorting lower bound (Theorem B.3) rules out the existence of any $o(n \log n)$-runtime oblivious algorithm for tight stable compaction in the indivisible model (since from tight stable compaction one can easily construct oblivious sorting for 1-bit keys).

E.2 Selection

We consider the selection problem studied frequently in the classical algorithms literature [13, 47]: given an input array containing n number of (key, value) pairs, output the $k \leq n$ pairs with smallest keys. We note that Goodrich [35] considered a much weaker selection abstraction in which only

[13]This formulation is slightly stronger than Goodrich's [35]. Unlike Goodrich's formulation, here our algorithm is not aware of the number of real elements.

the k-th smallest element is required to be output (henceforth we refer to his variant as the weak-selection problem). Goodrich [35] showed that weak-selection can be realized with $O(n/B)$ IOs and almost linear time if the algorithm knows the cache's parameters M and B. It is not too difficult to extend Goodrich's algorithm to realize (strong) selection for $k = o(n/\log^2 n)$ — however, for larger k values, e.g., $k = O(n)$, a straightforward extension of their algorithm would result in $\Omega(n \log n)$ since they lack an efficient tight compaction building block (which they phrased as an open question in their paper).

Using our oblivious tight compaction algorithm, we can easily devise a linear IO, almost linear-time selection algorithm that not only selects *all* k smallest elements for any $k \leq n$, but also is cache-agnostic. The idea is simple:

1. First, as we explain in more detail below, we rely on an oblivious variant of Demaine's median algorithm [25] (which is a modification of the classical, linear-time median algorithm by Blum et al. [13]) to find the element of the k-th rank[14].

2. Next, in one scan, we mark all elements greater than k as dummy, and we call tight compaction to suppress all dummies.

It thus suffices to describe how to find the element of the k-th rank. We first describe Demaine's cache-agnostic selection algorithm which proceeds as follows. First, divide the input array into groups of 5 and find the median of every 5 elements. Then the algorithm recurses and finds the median of the medians. Then in a partitioning step, the elements are partitioned into a set smaller than this median (of the medians) and a set that is larger than or equal to this median. Finally, the algorithm recurses on the partition which contains the element of the desired rank. This algorithm is *not* oblivious due to two reasons:

1. First, Demaine's implementation of the partitioning step is not oblivious.
2. Second, we cannot reveal how many elements fall on each side of the median (of the medians).

In light of these issues, we make the following modifications to Demaine's algorithm.

- First, we rely on our new $\mathsf{Partition}$ algorithm to perform the partitioning step in an oblivious manner.
- Second, since we cannot reveal how many elements are on each side of the median of the median, no matter which side we recurse on, we always overapproximate and use $\frac{7}{10}n$ as the length of the array to recurse on — note that the algorithm guarantees that the number of elements on either side must be bounded by $\frac{7}{10}n$.
- Third, due to the conditions necessary for our probabilistic analysis, whenever the problem size is smaller than $\log^6 \lambda$, we stop the recursion and simply rely on $\mathsf{FunnelOSort}$ to find the element of the desired rank.

It is not difficult to see that this modified algorithm achieves $O(n \log \log n)$ runtime and consumes $O(n/B)$ IOs whenever $M \geq \log^{1.1} \lambda$ (which is satisfied under the standard tall cache and wide cache-line assumptions). This gives rise to the following corollary:

[14] Alternatively, the following variant of our $\mathsf{Partition}$ will also work: sort all buckets, then sort the stripe covering the k-th element, and then the median of the sorted stripe is the global k-th element with overwhelming probability; moreover, the first piece (before the k-th element) contains all k smallest element with overwhelming probability.

Corollary E.2. (Selection in almost linear time and linear IO). *There exists a negligible function* $\mathsf{negl}(\cdot)$ *and a cache-agnostic, oblivious algorithm (with an explicit construction) which, except with* $\mathsf{negl}(\lambda)$ *probability, correctly achieves selection over* n *elements in* $O(n \log\log\lambda)$ *time and with* $O(n/B)$ *IOs assuming the cache size* $M \geq \log^{1.1}\lambda$ *(which is satisfied under the standard tall cache and wide cache-line assumptions).*

E.3 Additional Applications

Although oblivious algorithms aroused significant interest in the security and architecture communities due to emerging applications such as cloud outsourcing [24, 37, 66, 74] and secure processor design [29, 49, 51, 64], the available toolbox for designing oblivious algorithms in practical applications is in fact embarrassingly small in comparison with our rich body of knowledge on classical algorithms for (non-oblivious) RAMs.

Our work also enriches the toolbox for oblivious algorithms. For example, Goodrich and Mitzenmacher [37] and others [50, 57] have shown that any algorithm that has an efficient representation in a MapReduce model (with a streaming reduce function) can be obliviously computed asymptotically faster than the best known ORAM scheme [21, 45, 71]. Goodrich and Mitzenmacher's compiler [37] that converts a streaming MapReduce program to an oblivious form makes use of oblivious sort to aggregate keys of the same value. Thus, for cases where the key space is small, our results will immediately improve both the runtime and IO efficiency of such a compiler. This observation has numerous applications and we give some examples below:

- *Histogram.* Histogram is efficiently expressible in the streaming-MapReduce framework, our results immediately imply that there exists a $o(n \log n)$-time oblivious algorithm for evaluating histograms with $2^{o(\log n)}$ distinct bins.
- *K-means.* K-means is a well-known algorithm for performing clustering of data. Each iteration of the K-means algorithm reassigns each data point to the nearest cluster, and then updates the cluster centers by averaging. Our results imply that for K-means where $K = O(\log\log n)$, each iteration of the algorithm can be obliviously evaluated in $O(n \log\log n)$ time and $O(n/B)$ IOs (assuming standard cache assumptions) where n denotes the number of data points.

F Deferred Proofs

F.1 Proof of Theorem 5.1

Below, we prove the correctness of $\mathsf{SortSmall}$ (Algorithm 2) in Lemma F.2, analyze the running time in Lemma F.3 and the IO-cost in Lemma F.4.

Lemma F.1 (Piecewise-ordered partition)**.** *Let* m *be the median key in* A*. There exists a negligible function* $\mathsf{negl}(\cdot)$ *such that after we sort the crossover stripe, the maximum element in the top half is at most* m *and the minimum element in the bottom half is at least* m *except with probability* $\mathsf{negl}(\lambda)$*.*

Proof. We prove the statement of the top half, and the bottom half follows symmetrically. Let A be the $Z \times \frac{n}{Z}$ matrix after $\mathsf{RandCyclicShift}$. For each element A_{ij}, define the random variable X_{ij} as

$$X_{ij} := \begin{cases} 1 & \text{if } A_{ij} \leq m \\ 0 & \text{if } A_{ij} > m \end{cases} .$$

Let random variable $X_j := \sum_{i=0}^{Z-1} X_{ij}$ be the number of keys in column j that is at most m. For every column j, observe that the expectation $\mathbf{E}[X_j]$ is the same value $\mu = (\sum_i \sum_j X_{ij})/n$. For every j, the following Hoeffding's inequality holds (similar to Lemma 4.1):

$$\Pr(|X_j - \mu| \geq \frac{1}{2}\delta) \leq 2e^{-2\frac{(\log^4 \lambda)^2}{4Z}} = 2e^{-0.5 \log^2 \lambda}.$$

Considering random variables $Y_{\max} := \max_j\{X_j\}$ and $Y_{\min} := \min_j\{X_j\}$, we have $Y_{\max} - Y_{\min} < \delta$ except with probability $\mathsf{negl}(\lambda) := \frac{n}{Z} 2e^{-0.5 \log^2 \lambda}$ by union bound. We say that the *conceptual mixed stripe* (after sorting columns) is the rows between $Y_{\min}$ and $Y_{\max}$. Conditioning on the event that $Y_{\max} - Y_{\min} < \delta$, i.e., the height of the mixed stripe is less than δ, we consider the location of the conceptual mixed stripe in the following cases.

- If the mixed stripe is totally in the bottom half, i.e., $Y_{\min} \geq \frac{Z}{2}$, then there are at least $\frac{Z}{2}$ elements greater than m for every column of A. It follows that after sorting the crossover stripe, every element is at most m in the top half.
- If the mixed strip is on the boundary of the top and bottom half, i.e., $\frac{Z}{2} - \delta \leq Y_{\min} < \frac{Z}{2}$, then the mixed stripe of height δ is totally covered by the crossover stripe of height 2δ. Note that any element a such that $a > m$ and in the top half must be in the crossover stripe by $\frac{Z}{2} - \delta \leq Y_{\min}$ and Line 5(c). Hence, after sorting the crossover stripe, any $a > m$ must be in the bottom half because there are at most $n/2$ such a. The statement of the top half follows by the contraposition.

Note that the mixed stripe cannot be totally in the top half ($Y_{\min} < \frac{Z}{2} - \delta$) while conditioning on $Y_{\max} - Y_{\min} < \delta$ (otherwise $Y_{\max} < \frac{Z}{2}$, and then m cannot be the median). It follows that every element in the top half is at most m. □

After sorting the crossover stripe, we say the top and bottom half pieces form a *piecewise-ordered partition* of A iff the partition satisfies the median property as stated in the above Lemma F.1. Now, we prove the correctness of SortSmall using the following fact and union bound.

Fact 3. *If the top and bottom halves is a piecewise-ordered partition of A, then at least one in the two pieces has at most $\lceil \frac{K}{2} \rceil$ distinct keys.*

Lemma F.2. *There exists a negligible function $\mathsf{negl}(\cdot)$ such that for any input array of n elements and keys from the domain $[K]$, SortSmall correctly sorts the array with probability $1 - \mathsf{negl}(\lambda)$.*

Proof. If all partitions of A in the recursive SortSmall are piecewise-ordered, and all base-case Partitions are correct, then SortSmall outputs correctly by Fact 3. By taking union bound over Lemma F.1 and Theorem 4.1, the bad event happens with probability negligible in λ as there are at most n crossover stripes and K Partitions. □

Lemma F.3. *The algorithm runs in time $O(n \log K \log\log \lambda)$ for all $n \geq \lambda$.*

Proof. Let c_0, c_1 be constants such that FunnelOSort runs in time $c_0 n \log n \log\log \lambda$ and Partition runs in time $c_1 n \log\log \lambda$. Denote $T(n, K)$ as the runtime of SortSmall on n elements in the domain $[K]$. Observe that sorting a column takes a constant number of Partition as Selection is implemented by Partition. Hence, the recursion of runtime is

$$T(n, K) = T\left(\frac{n}{2}, K\right) + T\left(\frac{n}{2}, \left\lceil \frac{K}{2} \right\rceil\right) + c_1' n \log\log \lambda,$$

where $c_1' > c_1$ is the constant absorbing the runtime of all other linear-time operations. We prove by induction that the property $T(n, K) \le c \cdot n \log K \log\log \lambda$ holds for all positive integer n and $K \ge 2$, where $c := \frac{c_1'}{\log 1.3}$ is a constant. For two base cases,

1. If $n < 2Z$, the runtime is *not* bounded by $c \cdot n \log K \log\log \lambda$. We set $T(n, K) = 1$ in the recursion and will sum up this term later.

2. If $K = 2$, the runtime is $c_1 n \log\log \lambda \le c \cdot n \log K \log\log \lambda$.

Assuming the induction hypothesis holds: for all $n' < n$, $K' \le K$, $T(n', K') \le c \cdot n' \log K' \log\log \lambda$. To prove that $T(n, K) \le c \cdot n \log K \log\log \lambda$ holds, we substitute the recursion,

$$\begin{aligned} T(n,K) &\le c \cdot \tfrac{n}{2} \log K \log\log \lambda \\ &\quad + c \cdot \tfrac{n}{2} \log \lceil \tfrac{K}{2} \rceil \log\log \lambda + c_1' n \log\log \lambda \\ &\le c \cdot n \log K \log\log \lambda \\ &\quad - (\log 1.3) c \cdot n \log\log \lambda + c_1' n \log\log \lambda \end{aligned}$$

where the second inequality holds by $\lceil K/2 \rceil \le K/1.3$ for all $K \ge 2$. By $c = c_1'/(\log 1.3)$, it follows that $T(n, K) \le c \cdot n \log K \log\log \lambda$. Now, the base case, $n < 2Z$, which takes time $O(2Z(\log 2Z) \log\log \lambda)$. If $\log K \ge \log\log \lambda$, then, there are at most $n/(2Z)$ such base cases, and thus the total time of all such base cases is $O(n \log^2 \log \lambda)$, which is dominated by $O(n \log K \log\log \lambda)$. Otherwise, $\log K < \log\log \lambda$, then most elements go to the $K = 2$ base case, and only a small fraction of elements go to $n < 2Z$ base case: it takes $(\log n - \log 2Z)$ recursive calls to reach $n < 2Z$, there are at most $\log K - 1$ such call has K divided by 2 (it is $K = 2$ otherwise), and thus the number of such base cases is

$$\begin{aligned} \sum_{i=0}^{\log K - 1} \binom{\log n - \log 2Z}{i} &\le \binom{\log n - \log 2Z}{\log K - 1} \log K \\ &\le \left(\frac{(\log n - \log 2Z)e}{\log K - 1} \right)^{\log K - 1} \log K, \end{aligned}$$

where the RHS is at most $(e \log n)^{\log K - 1} \log K$. Given that $n = \lambda^{c'}$ for some constant c', it follows that the multiplication of $(e \log n)^{\log K - 1} \log K$ and $O(2Z(\log 2Z) \log\log \lambda)$ is bounded by $O(n \log K \log\log \lambda)$. Hence, $T(n, K) = O(n \log K \log\log \lambda)$. ☐

Lemma F.4. *Assuming tall cache and wide cache-line,* SortSmall *runs in IO-cost* $O(\lceil \frac{n}{B} \rceil \log K)$.

Proof. The IO-cost has a similar recursion,

$$C(n, K) = C(\frac{n}{2}, K) + C(\frac{n}{2}, \frac{K}{2}) + O(\lceil \frac{n}{B} \rceil \log_M \log \lambda),$$

where the base case $n < 2Z$ is $O(\lceil \frac{Z}{B} \rceil \log_{\frac{M}{B}} \lceil \frac{Z}{B} \rceil)$ by FunnelOSort, and the base case $K \le 2$ is $O(\lceil \frac{n}{B} \rceil \log_M \log \lambda)$ by Partition. Hence, the total IO-cost is $O(\lceil \frac{n}{B} \rceil (\log K \log_M \log \lambda + \log_{\frac{M}{B}} \lceil \frac{\log^6 \lambda}{B} \rceil))$. Recall that wide cache-line assumes $B \ge \log^c n$ for $c \ge 0.55$ and tall cache assumes $M \ge B^2$, the IO-cost is bounded by $O(\lceil \frac{n}{B} \rceil \log K)$. ☐

This concludes the proof of Theorem 5.1.

F.2 Proof of Theorem 5.2

We show the correctness by proving the following Lemma F.5 and F.6. Then, the running time is analyzed in Lemma F.7 by solving the recursion.

Correctness. The following Lemma is the counterpart of Lemma F.1, and their proofs are identical except for that the element of rank $\frac{tn}{p}$ is used instead of the median. Thus the proof of the following lemma is omitted.

Lemma F.5. *For every $t \in \{1, 2, \dots, p-1\}$, let m_t be the element of rank $\frac{tn}{p}$ in A (which should be sorted to the $\frac{tZ}{p}$-th row of A). After sorting the crossover stripes, let Top_t be the piece above the $\frac{tZ}{p}$-th row, and Bottom_t be the piece below the row. There exists a negligible function $\mathsf{negl}(\cdot)$ such that the maximum element in Top_t is at most m_t and the minimum element in Bottom_t is at least m_t except with probability $\mathsf{negl}(\lambda)$.*

After sorting the crossover stripes, we extend the terminology and say that $A_1, \dots, A_p$ is a *piecewise-ordered partition* of A iff for every $t \in \{1, 2, \dots, p-1\}$, m_t partitions A correctly as stated in Lemma F.5. Now, conditioning on the event that $\{A_i\}_{i\in[p]}$ is piecewise-ordered, we show that there exist many pieces such that consists of significantly reduced key domain.

Lemma F.6. *Let A be an array of key in the domain $[K]$, $\mathbf{A} := \{A_1, \dots, A_p\}$ be a piecewise-ordered partition of A. Then, for every $i \in [p]$, there exist at least i pieces of $A' \in \mathbf{A}$ such that the domain of A' is at most $\lceil \frac{K}{p-i+1} \rceil$.*

Proof. Define the set $\mathbf{A}_i := \{A' \in \mathbf{A} : A'$ has at most $\lceil \frac{K}{p-i+1} \rceil$ distinct keys$\}$ for every $i \in [p]$. Equivalently, the lemma states that $|\mathbf{A}_i| \geq i$. It suffices to prove the property holds had $\{A_i\}_{i\in[p]}$ was totally sorted as the domain of each A_i are identical. We prove that $|\mathbf{A}_i| \geq i$ holds for every $i \in [p]$ by induction.

For $i = 1$, let K_j be the domain of the piece A_j, $K_{\min} := \min_j\{K_j\}_{j\in[p]}$. Then, the domain of $\mathbf{A}$ is at least $pK_{\min} - (p-1)$, where $p-1$ deducts the double counting on $(p-1)$ boundaries. Solving $pK_{\min} - (p-1) \leq K$, we have $K_{\min} \leq \lceil K/p \rceil$, and hence the property holds for $i = 1$. For any $i > 1$, assume by induction hypothesis that the property holds for $i-1$, i.e., $|\mathbf{A}_{i-1}| \geq i-1$. Considering the $p-i+1$ pieces of largest domain, the union domain of all such pieces is at most K. Hence, there must exists a piece $A' \notin \mathbf{A}_{i-1}$ such that has domain at most $\lceil \frac{K}{p-i+1} \rceil$ by the same argument as $i = 1$. Observing $\mathbf{A}_{i-1} \subseteq \mathbf{A}_i$ and $|\mathbf{A}_{i-1}| \geq i-1$, it follows $|\mathbf{A}_i| \geq i$. ☐

By Lemma F.5 and Lemma F.6, we conclude that Sort is correct except with negligible probability.

Time complexity. Recall that in Algorithm 3 osort is instantiated with Zigzag sort [36]. We now analyze the algorithm's runtime for this specific instantiation.

Lemma F.7. *The algorithm Sort runs in time $O(n \frac{\log K}{\log\log K} \log\log\lambda)$.*

Proof. Let c_0, c_1 be constants such that Zigzag sort runs in time $c_0 n \log n$ and $\mathsf{SortSmall}$ runs in time $c_1 n \log K \log\log\lambda$. Denote $T(n, K)$ as the runtime of Sort on n elements in the domain $[K]$. Observe that all operations are linear-time except for recursion, sorting columns, and sorting pieces. Sorting columns takes time $6c_0 n \log\log\lambda$, and sorting pieces takes time $2c_0 n \log p \leq 2c_0 n \log\log K \leq 4c_0 n \log\log\lambda$ for $K \leq \lambda^{c_2}$, where c_2 is a constant. (Recall that the key length is at most word length, which is $O(\log\lambda)$, which implies $K \leq \lambda^{c_2}$.) Hence, the recursion of runtime is

$$T(n, K) = \left(\sum_{i=1}^{p} T(\frac{n}{p}, \frac{K}{i}) \right) + c_0' n \log\log\lambda,$$

where $c_0' \geq 10c_0 + 1$ is the constant absorbing the runtime of all other linear-time operations. We prove by induction that the property $T(n, K) \leq c \cdot n \frac{\log K}{\log \log K} \log \log \lambda$ holds for all positive integer n and $K \geq 64$, where $c := \max(\frac{c_0'}{0.39}, 3c_1)$ is a constant. The property holds for two base cases:

1. If $n < 2Z$, the runtime is $c_0 n \log n < c_0 n(\log 2Z) \leq c \cdot n \frac{\log K}{\log \log K} \log \log \lambda$ (recall that $Z = \log^6 \lambda$).
2. If $K \leq 64$, the runtime is $c_1 n \log K \log \log \lambda \leq c \cdot n \frac{\log K}{3} \log \log \lambda \leq c \cdot n \frac{\log K}{\log \log K} \log \log \lambda$.

Assuming the induction hypothesis holds: for all $n' < n$, $K' \leq K$, $T(n', K') \leq c \cdot n' \frac{\log K'}{\log \log K'} \log \log \lambda$. To prove that $T(n, K) \leq c \cdot n \log K \log \log \lambda$ holds, we substitute the recursion,

$$\begin{aligned} T(n, K) = & \left(\sum_{i=1}^{p} c \cdot \frac{n}{p} \frac{\log (K/i)}{\log \log (K/i)} \log \log \lambda \right) + \hat{c} \\ \leq & \left(\frac{cn}{p \log \log(K/p)} \log \log \lambda \sum_{i=1}^{p} \log (K/i) \right) + \hat{c}, \end{aligned}$$

where $\hat{c}$ is $c_0' n \log \log \lambda$ for short. Note that $\sum_{i=1}^{p} \log (K/i) = p \log K - \log(p!) \leq p(\log K - \log p)$ by $\log(p!) \geq p \log p$. By $p = \log K$, we have

$$T(n, K) \leq cn(\log \log \lambda) \frac{\log K - \log \log K}{\log(\log K - \log \log K)} + \hat{c}.$$

Note the fact that $g(x) := \frac{x}{\log x} - \frac{x - \log x}{\log(x - \log x)} > 0.39$ for all $x \geq 6$ (as the derivative is positive). By $K > 64$, it follows that $\frac{\log K - \log \log K}{\log(\log K - \log \log K)} < \frac{\log K}{\log \log K} - 0.39$, and hence the induction holds for all c such that $0.39c \geq c'$. $\square$

This concludes the proof of Theorem 5.2.