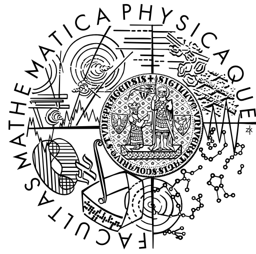

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

DOCTORAL THESIS

Karel Král

Complexity of dynamic data structures

Computer Science Institute of Charles University

Supervisor of the doctoral thesis: prof. Mgr. Michal Koucký, Ph.D.
Study programme: Computer Science
Study branch: Theory of Computing, Discrete Models and Optimization

Prague 2021

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my parents who have been there for me my whole life. Huge thanks goes to Veronika Slívová for not only being with me in my personal life but also for working with me on our research.

I would also like to thank all co-authors as they have contributed a lot to the results contained in this thesis, namely Pavel Hubáček and Michael Saks. I would also like to thank all other collaborators with whom I have been working on results not contained in this thesis. Last but not least I would like to thank my advisor, Michal Koucký, for all the advice and time spent with me. Without him this thesis would have never existed.

This research was supported in part by the Grant Agency of the Czech Republic under the grant agreement no. 19-27871X, by the Charles University projects PRIMUS/17/SCI/9, GAUK 1568819, and Charles University grant SVV-2017-260452. At the beginning of my studies I have been funded from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 616787.

Title: Complexity of dynamic data structures

Author: Karel Král

Institute: Computer Science Institute of Charles University

Supervisor: prof. Mgr. Michal Koucký, Ph.D., Computer Science Institute of Charles University

Abstract: Sorting is one of the fundamental problems in computer science. In this thesis we present three individual results.

Asymptotically optimal sorting networks have been created by Ajtai et al. [1983]. But Asharov et al. [2021] have observed that boolean circuits based on sorting networks are not optimal for sorting short integers. We present a construction of even smaller boolean circuits for sorting short integers.

Lower bounds for offline Oblivious RAM have been connected to lower bounds for sorting circuits by Boyle and Naor [2016]. Larsen and Nielsen [2018] have shown a lower bound for online Oblivious RAM. We provide a lower bound for online Oblivious RAM in a more general model.

Finally we provide an algorithm with expected running time $\mathcal{O}(n \log \log(n))$ for sorting integers on random access machine with word length between $\log(n)$ and $\log(n)$ cubed. This algorithm does not match the expected running time of the algorithm of Han and Thorup [2002], but our algorithm is much easier to implement and analyse.

Keywords: Sorting, Boolean Circuit, online Oblivious RAM

Contents

Introduction **3**

1 History and Previous Results **5**
1.1 Classical Sorting Algorithms . . . 5
1.2 RAM . . . 6
1.3 Sorting Networks . . . 6
1.4 Sorting Boolean Circuits . . . 6
1.5 Turing Machines . . . 7
1.5.1 Upper Bounds on Turing Machines . . . 7
1.5.2 One Tape Turing Machines . . . 7
1.5.3 Balls and Bins Model Lower Bounds . . . 8
1.6 ORAM . . . 8
1.6.1 Constructions . . . 9
1.6.2 Lower-bounds . . . 9
1.7 Connections . . . 10
1.7.1 Super-concentrators . . . 10
1.7.2 Non-adaptive Hellman Attack . . . 10
1.7.3 Network Coding Conjecture . . . 10

2 Boolean Circuits for Sorting and Counting Short Integers **11**
2.1 Our Results . . . 11
2.1.1 Our Techniques . . . 12
2.2 Notation . . . 13
2.3 Preliminaries . . . 14
2.4 Sorting Circuits Based on AKS Network . . . 18
2.5 Sorting n Binary Strings of Length m . . . 20
2.6 Partial Sorting by the First k Bits in Poly-logarithmic Depth . . . 23
2.7 Conclusion . . . 29

3 Oblivious RAM **30**
3.1 Our Results . . . 30
3.1.1 Our Techniques . . . 31
3.2 Preliminaries . . . 32
3.2.1 Online ORAM . . . 33
3.3 Dense Graphs . . . 36
3.4 ORAM Lower Bound . . . 37
3.5 Alternative Definitions for Oblivious RAM . . . 42

4 Sorting on RAM **46**
4.1 Overview of Our $\mathcal{O}(n \log\log(n))$ Algorithm . . . 46
4.2 Coupon Collector . . . 46
4.3 The Algorithm . . . 47
4.4 Practical Implementation . . . 54
4.5 Short Word Lengths . . . 55
4.6 Open Questions . . . 55

Conclusion **56**
4.6.1 Acknowledgements 56
Bibliography **57**
List of Figures **64**
List of Publications **65**

Introduction

Sorting undoubtedly plays a central role in computer science. Many problems can be solved using sorting as a subcomponent. There are many practical variants of sorting based either on what we sort (integers, rational numbers, strings, etc.) or how we sort (in parallel, in distributed fashion, in external memory, etc.). Despite the fact that sorting has been and still is studied extensively there are still many basic questions about sorting unanswered. This thesis contributes three drops into the wast sea of results about sorting.

This thesis is based on the papers Hubáček et al. [2019] presented in Chapter 3 and Koucký and Král [2021] presented in Chapter 2. This chapter contains abstracts of those papers and Chapter 1 extends the previous work chapters by other sorting directions, namely those connected to our $\mathcal{O}(n \log\log(n))$ expected time sorting algorithm presented in Chapter 4.

Chapter 1 is without a doubt an incomplete overview of research connected to sorting. If the author has omitted a result it is not because it was not interesting, but because of the sheer volume of literature. Notation is provided on per chapter basis as the intersection of notations consists just of the set of natural numbers.

Circuits

Sorting networks have been extensively studied. The celebrated result of Ajtai et al. [1983] shows a construction of asymptotically optimal sorting network. However the famous $\Omega(n \log(n))$ lower bound for comparator based sorting applies to sorting networks. A natural question is if we can build smaller boolean circuits than what we would get from implementing the Ajtai et al. [1983] sorting network.

We build boolean circuits of size $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$ for sorting m-bit integers. We build also circuits that sort m-bit integers according to their first k bits that are of size $\mathcal{O}(nmk(1 + \log^*(n) - \log^*(m)))$ and depth $\mathcal{O}(\log^3(n))$. This improves on the results of Asharov et al. [2021] and resolves some of their open questions. See Chapter 2 for more details.

Oblivious RAM

Oblivious RAM (ORAM), introduced in the context of software protection by Goldreich and Ostrovsky [1996], aims at obfuscating the memory access pattern induced by a RAM computation. Ideally, the memory access pattern of an ORAM should be independent of the data being processed. Since the work of Goldreich and Ostrovsky [1996], it was believed that there is an inherent $\Omega(\log(n))$ bandwidth overhead in any ORAM working with memory of size n. Boyle and Naor [2016] proved that any super-constant lower bound for *offline* ORAM, i.e., an ORAM that can process its inputs simultaneously, implies super-linear lower bounds on size of sorting circuits – which would constitute a major breakthrough in computational complexity. Larsen and Nielsen [2018] were the first to give a general $\Omega(\log(n))$ lower bound for any *online* ORAM, i.e., an ORAM that must process its inputs in an online manner.

In Chapter 3, we revisit the lower bound of Larsen and Nielsen [2018], which was proved under the assumption that the adversarial server knows exactly which server accesses correspond to which input operation. We give an $\Omega(\log(n))$ lower bound for the bandwidth overhead of any online ORAM even when the adversary has no access to this information. For many known constructions of ORAM this information is provided implicitly as each input operation induces an access sequence of roughly the same length. Thus, they are subject to the lower bound of Larsen and Nielsen [2018]. Our results rule out a broader class of constructions and specifically, they imply that obfuscating the boundaries between the input operations does not help in building a more efficient ORAM.

As our main technical contribution and to handle the lack of structure, we study the properties of *access graphs* induced naturally by the memory access pattern of an ORAM computation. We identify a particular graph property that can be efficiently tested and that all access graphs of ORAM computation must satisfy with high probability. This property is reminiscent of the Larsen and Nielsen [2018] property but it is substantially less structured; that is, it is more generic.

Simple $\mathcal{O}(n \log\log(n))$ Expected Time Algorithm

In Chapter 4 we present an $\mathcal{O}(n \log\log(n))$ expected time sorting algorithm. This construction does not match the expected $\mathcal{O}(n\sqrt{\log\log(n)})$ algorithm of Han and Thorup [2002]. On the other hand our algorithm is simple to analyse and implement. The algorithm uses rather simple techniques requiring knowledge of only basic data structures and algorithms and an easy result about the coupon collector model. One needs to know just hash maps, binary search, resizeable arrays (their amortized analysis), and any $\mathcal{O}(n \log(n))$ sorting algorithm.

1. History and Previous Results

The research mentioned here is recalled in order to put our results in perspective and to mention connections to previous work. We are certain that we will omit some results due to the sheer volume of interest in sorting both from the practical and theoretical point of view. A reader who is interested in history may wish to consult the classical book Knuth [1997] for more historical notes.

The classical comparison based sorting takes time $\mathcal{O}(n \log(n))$ when sorting n integers. Well known lower bound postulates that this is optimal for comparison based sorting. However, this is a great over-simplification and the picture is much more nuanced: sorting integers from a domain of size M can be done using binary search trees or bucket sort in time $\mathcal{O}(n \log(|M|))$, thus sorting for example m-bit integers only needs $\mathcal{O}(nm)$ comparisons. Such an algorithm can be implemented for instance on a pointer machine. We discuss even faster algorithms in Section 1.2.

1.1 Classical Sorting Algorithms

One could argue that sorting was amongst the first problems that appeared in data processing systems. Knuth [1997] mentions that the sorting box of Herman Hollerith was used in the 1890 USA census. The machine had both electrical and mechanical components and was fed punched cards. The company founded by Hollerith was later merged with other companies and after few years renamed to IBM. The book of Campbell-Kelly and Aspray provides more details of this history.

The mergesort algorithm is attributed to John von Neumann as early as 1945 or with detailed analysis in 1948 in a report by Goldstine and Neumann (see for instance Knuth [1997] and Katajainen and Träff [1997]). We cite the result of mergesort as Knuth [1997] even though it was analysed by von Neumann much earlier and its variants have been analysed by many others in the following decades. Our reasoning is that the readers are probably already familiar with mergesort analysis and if not they are better with a textbook of their choice than with the original source. A reader interested in history can consult the references given in Knuth [1997] and Katajainen and Träff [1997] to find references to the original work.

The quicksort algorithm invented by Hoare [1961] became perhaps one of the most studied and used sorting algorithms. Quicksort became a part of the C standard library (`qsort`, see Bentley and McIlroy [1993]) a variant of it is used in Java 7 library Wild et al. [2013] to name just two popular programming languages. Theoretical properties of quicksort and its variants have been also extensively studied. Sedgewick [1975] resolved many problems that were open until then. There are copious other classical algorithms that could be mentioned. But we try to keep the list limited to the algorithms directly used in this thesis. In Chapter 4 we directly use any algorithm with running time $\mathcal{O}(n \log(n))$ where both quicksort and mergesort are natural candidates.

1.2 RAM

The *random access machine* model (see Hopcroft et al. [2001] for a formal definition) allows much more than just to compare two inputs. For instance radixsort can be traced back to machines of Hollerith (mentioned in the previous section).

One of the first advances was van Emde Boas tree van Emde Boas [1975] which if used with a hash-table can sort n numbers from the range $0, \ldots, K-1$ in time $\mathcal{O}(n \log\log(K))$. A beautiful series of results followed with the works of Willard [1983], Kirkpatrick and Reisch [1983], culminating with the $\mathcal{O}\left(n\sqrt{\log\left(\frac{m}{\log(n)}\right)}\right)$ expected time algorithm of Han and Thorup [2002], where m is the length of the words we are sorting. When $m = \Omega(\log^2(n)\log\log(n))$ one can sort in expected linear time using the algorithm of Belazzougui et al. [2014].

1.3 Sorting Networks

Batcher [1968] proposed *sorting networks*, an oblivious comparison based parallel model of computation intended for sorting. Numbers in a sorting network are thought of as signals which can only be compared. Sorting networks have became an important part of graphics processing units (Owens et al. [2008]).

The seminal paper by Ajtai et al. [1983] gives an asymptotically optimal sorting network of logarithmic depth and thus having $\mathcal{O}(n\log(n))$ comparators matching the comparison based lower bound. The AKS network has immense applications in theoretical computer science, and we use it in Chapter 2. Since the multiplicative constant in the construction of Ajtai et al. [1983] is extremely large there have been subsequent work attempting constructions with smaller asymptotic constants. Paterson [1990] improves the constant significantly but the multiplicative constant of the depth still prevents practical use. Goodrich [2014] presents a sorting network with $\mathcal{O}(n\log(n))$ comparators which is much smaller than previous constructions, however the networks depth is $\mathcal{O}(n\log(n))$.

1.4 Sorting Boolean Circuits

Another oblivious model of computation heavily used throughout theoretical computer science are boolean circuits. One can turn the AKS sorting network into a circuit sorting n integers each of m-bits of size $\mathcal{O}(nm\log(n))$ and depth $\mathcal{O}(\log(m)\log(n))$ (see Section 2.4). However, when building boolean circuits for sorting it is not clear whether one can take any advantage of some of the faster algorithms for RAM or Turing machines as simulating random access memory or Turing machine tapes by circuits requires substantial overhead. Asharov et al. [2021] asked the question whether one can sort m-bit integers in time $o(nm\log(n))$ when $m = o(\log(n))$. They provide an answer to this question by constructing circuits for sorting m-bit integers of size $\mathcal{O}(nm^2(1+\log^*(n)-\log^*(m))^{2+\varepsilon})$ and super-poly-logarithmic depth, for any $\varepsilon > 0$. We improve their results: We build boolean circuits for sorting m-bit integers of size $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$. As proven by Asharov et al. [2021] pending some unexpected breakthrough this size seems optimal. The depth is provably optimal whenever $m = \mathcal{O}(\log(n)/\log\log(n))$.

Asharov et al. [2021] solve even a more general problem as their circuits partially sort n numbers each of m bits by their first k bits using a circuit of size $\mathcal{O}(nmk(1 + \log^*(n) - \log^*(m))^{2+\varepsilon})$. We improve on this result as well by presenting circuits that sort m-bit integers according to their first k bits of size $\mathcal{O}(nmk(1+\log^*(n)-\log^*(m)))$ and depth $\mathcal{O}(\log^3(n))$. Our small circuits of polylogarithmic depth answer some of the open questions of Asharov et al. [2021]. In a work subsequent to ours, Lin and Shi [2021] get circuits of depth $\mathcal{O}(\log(n)+\log(k))$ and size $\mathcal{O}(nkm \cdot \text{poly}(\log^*(n) - \log^*(m)))$ whenever $n > 2^{4k+7}$. They use substantially different approach. We state our results formally in Chapter 2.

1.5 Turing Machines

We do not show any new results about sorting on Turing machines in this thesis. On the other hand we discuss balls and bins lower bounds in Section 3.4 of this thesis with connection to lower bounds of Oblivious RAM. The beauty of Turing machines is that we have both unconditional lower bounds for one tape Turing machines and a measure of balls and bins permutation complexity of permutations for Turing machines with more tapes. In the classical lower bound for comparison based sorting we observe that the binary logarithm of the number of permutations on n elements is $\Theta(n \log(n))$ and thus most permutations need a lot of comparisons to sort. But that is a result of the type "most objects are hard" which seems to be more common than the type of result we discuss below which is given by Stoß [1973] and Paul [1979]. We assume that the reader is familiar with a definition of Turing machine as in the textbook Hopcroft et al. [2001].

1.5.1 Upper Bounds on Turing Machines

It is an easy exercise to design Turing machines that sort n integers of m-bits each in time $\mathcal{O}(nm^2)$. One may just implement radixsort on a Turing machine with two tapes. Another possible approach is to implement mergesort on a Turing machine with three tapes. This gives us an algorithm running in time $\mathcal{O}(nm \log(n))$. To get a mergesort on two tapes, one can of course use the theorem of Hennie and Stearns [1966] to turn the three tape Turing machine halting in time $T(|x|)$ into a two tape Turing machine halting in time $cT(|x|) \log(T(|x|))$ where c is a constant depending only on the description of the three tape Turing machine. The above gives us a two tape Turing machine running in time $\mathcal{O}(nm \log(n)(\log(n) + \log(m)))$. Both of these algorithms are described in more detail in the book Reischuk [1999].

1.5.2 One Tape Turing Machines

Hennie [1965] shows a general framework of crossing sequences for deriving lower bounds for problems on one head Turing machines. The intuition is that if we have just a single head then at most finitely many bits of information may be carried during single crossing from one side of a boundary to the other. In particular the crossing sequence is the sequence of states of the machine when crossing the boundary between two fixed neighbouring squares of its tape (for a formal

definition see Hennie [1965]). This approach naturally fails when there are two or more heads as a lot of information may be transferred without crossing a boundary many times.

The technique of crossing sequences has been applied successfully many times since. Let us mention that Petersen [2008] uses crossing sequences to show a lower bound of $\Omega(n^2 m)$ if $m \geq 2\log(n)$ for nondeterministic Turing machines with one work-tape and one-way input tape. Petersen [2008] also presents a nondeterministic Turing machine which sorts in time $\mathcal{O}(n^2 m)$ where $m = \mathcal{O}(n)$ implying that this bound is asymptotically tight.

1.5.3 Balls and Bins Model Lower Bounds

Turing machines with two or more working tapes allow more efficient algorithms as discussed in Section 1.5.1. Proving lower bounds also seems to be much harder. However there are lower bounds for the problem of performing a concrete permutation of inputs when we are allowed to only copy and compare the inputs but not to make computations with them. We call this model *balls and bins* as the numbers being sorted behave as atomic balls which can be only moved or copied.

Floyd [1972] is among the first who show a result of the kind that performing a permutation is hard. However the model is more reminiscent of the model considered in cache oblivious analysis (Frigo et al. [1999]) than of Turing machines. A lower bound on the number of the operations is derived assuming that there are pages each holding at most p records and each operation consists of loading two pages into memory and storing a subset of at most p records from their union in a third page. Floyd [1972] presents a concrete permutation that is hard to perform in this model.

Stoß [1973] shows a lower bound for performing a particular permutation on a Turing machine with k heads when the inputs can be just copied but no computation may be done with them. The result states a complexity measure for permutations and shows that when a Turing machine may not do anything with the inputs but to copy those then the complexity measure lower bounds the number of steps of the Turing machine. In particular the complexity measure is $\Omega(n^2 \log(n))$ for the matrix transposition permutation where we are given n^2 inputs $w_{1,1}w_{1,2}w_{1,3}\dots w_{1,n}w_{2,1}\dots w_{n,n}$ and we should output $w_{1,1}w_{2,1}\dots w_{n,1}w_{1,2}\dots w_{n,n}$. This permutation corresponds to transposition of a matrix stored in row-major order.

Paul [1979] partially lifts the assumption of atomicity of the balls and bins model and is still able to show a lower bound of $\Omega(n\log(n))$ steps for sorting (in fact the lower bound also holds for the matrix transposition problem). Still the model is somewhat limited as the Turing machine M cannot "perform magic" which is defined in terms of Kolmogorov complexity (see Kolmogorov [1998] or Li et al. [2008] for the definition of Kolmogorov complexity). The intuition behind the "performing magic" is that when we divide the time of computation into time intervals and the tapes into blocks, then if the machine "performs magic" then there is much more information in visited blocks than in the previous time interval. Paul [1979] states that performing exclusive OR does not fit the "no magic" assumption. For a formal definition and proof see the paper Paul [1979].

1.6 ORAM

In many cryptographic applications there is an interest in oblivious algorithms, algorithms in which the sequence of the operations is independent of the processed data. Oblivious simulation of RAM machines, initially studied in the context of software protection by Goldreich and Ostrovsky [1996], aims at protecting the memory access pattern induced by computation of a RAM from an eavesdropper. In the present day, such oblivious simulation might be needed when performing a computation in the memory of an untrusted server.[1] Despite using encryption for protecting the content of each memory cell, the memory access pattern might still leak sensitive information. Thus, the memory access pattern should be *oblivious* of the data being processed and, ideally, depend only on the size of the input.

1.6.1 Constructions

The strong guarantee of obliviousness of the memory access pattern comes at the cost of additional overhead. A trivial solution which scans the whole memory for each memory access induces linear *bandwidth overhead*, i.e., the multiplicative factor by which the length of a memory access pattern increases in the oblivious simulation of a RAM with n memory cells. Given its many practical applications, an important research direction is to construct an ORAM with as low overhead as possible. The foundational work of Goldreich and Ostrovsky [1996] already gave a construction with bandwidth overhead $O(\log^3(n))$. Subsequent results introduced various improved approaches for building ORAMs (see Ajtai [2010], Damgård et al. [2011], Goodrich and Mitzenmacher [2011], Goodrich et al. [2011], Kushilevitz et al. [2012], Chung and Pass [2013], Gentry et al. [2013], Chung et al. [2014], Ren et al. [2014], Wang et al. [2014, 2015], Patel et al. [2018], Stefanov et al. [2018] and the references therein) leading to the recent construction of Asharov et al. [2018] with bandwidth overhead $O(\log(n))$ for the most natural setting of parameters.

1.6.2 Lower-bounds

It was a folklore belief that an $\Omega(\log(n))$ bandwidth overhead is inherent based on a lower bound presented already in the initial work of Goldreich and Ostrovsky [1996]. However, the Goldreich and Ostrovsky [1996] result was revisited in the work of Boyle and Naor [2016], who pointed out that the lower bound actually holds only in the rather restricted "balls and bins" model where the ORAM is not allowed to read the content of the data cells it processes (it can only copy and compare). In fact, Boyle and Naor [2016] showed that any general lower bound for *offline* ORAM (i.e., where each memory access of the ORAM can depend on the whole sequence of operations it needs to obliviously simulate) implies non-trivial lower bounds on sizes of sorting circuits which seem to be out of reach of the known techniques in computational complexity. The connection between offline ORAM lower bounds and circuit lower bounds was extended to *read-only online* ORAMs (i.e., where only the read operations are processed in an online

[1]Protecting the memory access of a computation is particularly relevant in the light of the Spectre Kocher et al. [2018] and Meltdown Lipp et al. [2018] attacks.

manner) by Weiss and Wichs [2018] who showed that lower bounds on bandwidth overhead for read-only online ORAMs would imply non-trivial lower bounds for sorting circuits or locally decodable codes.

The first general $\Omega(\log(n))$ lower bound for bandwidth overhead in *online* ORAM (i.e., where the ORAM must process sequentially the operations it has to obliviously simulate) was given by Larsen and Nielsen [2018]. The core of their lower bound comprised of adapting the *information transfer* technique of Patrascu and Demaine [2006], originally used for proving lower bounds for data structures in the cell probe model, to the ORAM setting. In fact, the lower bound of Larsen and Nielsen [2018] for ORAM can be cast as a lower bound for the oblivious Array Maintenance problem and it was recently extended to other oblivious data structures by Jacob et al. [2019].

Since our paper Hubáček et al. [2019] there have been papers showing lower bounds for different ranges of parameters Komargodski and Lin [2020], for multi-server Larsen et al. [2020] setting, and many more. Lin et al. [2019] revisit the question of sorting on oblivious RAM, they show that sorting short numbers can be done faster and further investigate sorting in the balls and bins model.

1.7 Connections

In this section we investigate some of the attempts and conditional results connected to proving lower bounds for sorting circuits and similar problems. The ideas mentioned in this section are graph-theoretic in nature but are linked with the study of computational complexity.

1.7.1 Super-concentrators

Super-concentrators are graphs that were introduced by Valiant [1975] in hopes of proving lower bounds in computational complexity. Valiant disproved this conjecture later. For more results and history see the survey of Hoory et al. [2006]. In Chapter 2 we give more overview of super-concentrators and in our construction of sorting circuits we use the result of Pippenger [1996] who constructs super-concentrators where it is possible to compute the actual routing.

1.7.2 Non-adaptive Hellman Attack

Corrigan-Gibbs and Kogan [2019] prove that lower bounds for function inversion would imply circuit lower bounds. This result is using the common bits model introduced by Valiant [1977, 1983, 1992]. Similar connection has also been described by Viola [2019].

1.7.3 Network Coding Conjecture

In network coding model (see the paper of Ahlswede et al. [2000]) we are interested in how much information can be sent through a given network where the nodes can perform computations (each node sends messages based on a predefined function of its incoming messages). The network coding conjecture introduced

by Li and Li [2004] postulates that in undirected graphs network coding provides no benefit over multicommodity flows. Afshani et al. [2019] and Farhadi et al. [2019] use the network coding conjecture to prove conditional lower bounds. Dvořák et al. [2021] use the network coding conjecture to prove conditional lower bounds for polynomial evaluation and link the techniques of Corrigan-Gibbs and Kogan [2019] to problems that are more general than finding an inverse of a given function.

2. Boolean Circuits for Sorting and Counting Short Integers

The result presented in this chapter is based on the paper Koucký and Král [2021].

2.1 Our Results

We provide a family of boolean circuits that sort m-bit strings. Our circuits are smaller than the circuits directly derived from the AKS sorting network and they improve on the result of Asharov et al. [2021]. Our circuits achieve optimal logarithmic depth whenever $m\log(m) \leq \mathcal{O}(\log(n))$. Pending some unexpected breakthrough, their size seems also optimal.

Theorem 2.1.1. *For any integers $n, m \geq 1$ there is a size $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$ circuit that sorts n integers of m bits each.*

For $m \geq \Omega(\log(n))$, the existence of such a circuit directly follows from AKS sorting networks. Our contribution is the construction of such circuits for $m \leq o(\log(n))$. Our construction also uses a sorting network as a building block. We use the AKS sorting network as one of our primitives but in principle, we could use any sorting network or sorting circuit. In particular, we could use any circuit sorting n numbers of $\Theta(\log(n))$ bits each in our construction. Any improvement of asymptotic complexity of sorting of $\log(n)$-bit numbers would give us improved complexity of sorting short numbers.

The main idea behind our construction is to *compress* the input by computing the number of occurrences of each m-bit integer. This gives a vector of 2^m integers, each of size $\mathcal{O}(\log(n))$. Decompressing this vector back gives the sorted input. Combining the counting and decompressing circuit gives us a circuit that sorts. The main technical lemma is our counting circuit which is of independent interest.

Lemma 2.1.2. *For any integers $n, m \geq 1$ where $m \leq \log(n)/10$ there is a circuit*

$$\text{FAST_COUNT}_{n,m} \colon \{0,1\}^{nm} \to \{0,1\}^{2^m\lceil 1+\log(n)\rceil}$$

which given a sequence of n strings of m bits each outputs the number of occurrences of each possible m-bit string among the inputs, that is for input

$$x_1, x_2, \ldots, x_n \in \{0,1\}^m$$

it outputs $n_{0^m}, n_{0^{m-1}1}, \ldots, n_{1^m}$ where for each string $y \in \{0,1\}^m$, the output string $n_y \in \{0,1\}^{\lceil 1+\log(n)\rceil}$ represents $|\{j \in [n] \mid x_j = y\}|$ in binary. The size of the circuit $\text{FAST_COUNT}_{n,m}$ is $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$.

We also provide a family of boolean circuits which sort the input integers by their first k bits only. One can view this as sorting (key, value) pairs, where keys have k bits and values have $m - k$ bits. For the special case of $k = 1$ (that is partially sorting the numbers by a single bit) the problem is equivalent to routing in *super-concentrators* (see Section 2.1.1), and we use super-concentrators of Pippenger [1996] as our building block. We get size improvement over the result of Asharov et al. [2021] while achieving poly-logarithmic depth.

Theorem 2.1.3. *For any integers $n, m, k \geq 1$ where $k \leq m$ and $k \leq \log(n)/11$ there is a circuit*

$$\mathrm{SORT}_{n,m,k} \colon \{0,1\}^{nm} \to \{0,1\}^{nm}$$

which partially sorts n numbers each of m bits by their first k bits. The circuit $\mathrm{SORT}_{n,m,k}$ has size $\mathcal{O}(knm(1 + \log^*(n) - \log^*(m)))$ and depth $\mathcal{O}(\log^3(n))$.

2.1.1 Our Techniques

One can take AKS sorting networks and turn them into a family of boolean circuits of size $\mathcal{O}(nm\log(n))$ and depth $\mathcal{O}(\log(m)\log(n))$. For $m = o(\log(n))$ this is sub-optimal as shown by Asharov et al. [2021] who provide circuits of size $\mathcal{O}(nm^2(1 + \log^*(n) - \log^*(m))^{2+\varepsilon})$ for sorting n integers each of m-bits. Asharov et al. [2021] show how to reduce the problem of sorting m-bit integers according to the first k bits to the problem of sorting m-bit integers according to just a single bit. Sorting according to single bit is essentially equivalent to routing in super-concentrators.

Super-concentrators have been studied originally by Valiant [1975] with the aim of proving circuit lower bounds. A super-concentrator is a graph with two disjoint subsets of vertices $A, B \subseteq V(G)$, called inputs and outputs, with the property that for any set $S \subseteq A$ and $T \subseteq B$ of the same size there is a set of vertex disjoint paths from each vertex of S to some vertex of T. Pippenger [1996] constructs super-concentrators with a linear number of edges and an algorithm that on input describing S and T outputs the list of edges forming the disjoint paths between S and T. This can be turned into a circuit of size $\mathcal{O}(n\log(n))$ and depth $\mathcal{O}(\log^2(n))$.

The result of Pippenger [1996] can be used to build a circuit sorting by one bit, but the resulting circuit is of size $\Theta(nm + n\log(n))$ (see Corollary 2.6.2) which is much more than what we wish for. Thus, Asharov et al. [2021] used the technique of Pippenger [1996] rather than his result to design a circuit sorting by one bit, and iterate it to sort by k bits. Our technique differs substantially from that of Asharov et al. [2021], as we use the circuits from AKS networks and from Pippenger's super-concentrators as black box.

To sort m-bit integers for $2^m \ll n$ our approach is to count the number of occurrences of each number in the input. This compresses the input from nm bits into $2^m\log(n)$ bits. We can then decompress the vector back to get the desired output. So the main challenge is to construct counting (compressing) circuits of size $\mathcal{O}(nm^2)$. Interestingly, we use the sorting circuits derived from AKS networks to do that. But to avoid the size blow-up we don't use them on all of the integers at once but on blocks of integers of size 2^{8m}. Then the $\mathcal{O}(\log(n))$ overhead of the circuits turns into the acceptable $\mathcal{O}(m)$ overhead. Each sorted block is then subdivided into parts of size 2^{2m}. Clearly, most parts in each block will be monochromatic, they will contain copies of the same integer. There will be at most 2^m non-monochromatic parts. We *move* the parts within a block to one side using another application of the AKS sorting circuit. Then we can afford to build a fairly expensive counting circuit for the small fraction of non-monochromatic parts, while cheaply counting the monochromatic parts. Summing the results by linear size circuit gives us the desired compression. Our decompression essentially mirrors the compression.

We also design a circuit to sort according to a single bit improving the parameters of Asharov et al. [2021]. We take the circuit of Pippenger [1996] as basis and apply it iteratively to larger and larger blocks of inputs. Again we start from blocks of size $2^{\mathcal{O}(m)}$, and increase the size of the blocks exponentially at each iteration. We use Pippenger's circuit to sort each block by the bit. When we split the block into parts, only one will be monochromatic. Merging multiple blocks into one gives a mega-block with only a small fraction of non-monochromatic parts. These non-monochromatic parts can be separated from monochromatic ones, re-sorted, and re-partitioned to give only one non-monochromatic part in the mega-block. Each part takes on the role of an "m"-bit integer in the next iteration. Iterating this process leads to the desired result.

To sort according to the first k bits we use the one-bit sorting similarly to Asharov et al. [2021]. Thanks to our efficient sorting circuits for m-bit integers to sort the k-bit keys, we can avoid the use of median finding circuits.

Organization. In the next section we review our notation. We provide basic construction tools including naïve constructions of counting and decompression circuits in Section 2.3. In Section 2.4 we recall basic facts on AKS sorting networks and related sorting circuits. In Section 2.5 we prove our main result by constructing efficient counting and decompression circuits. Finally, we provide a construction of partial sorting circuits for Theorem 2.1.3 in Section 2.6.

2.2 Notation

In this thesis $\mathbb{N}$ denotes the set of natural numbers (without zero), and for $1 \leq a \leq b \in \mathbb{N}$, $[a, b] = \{a, a+1, \ldots, b\}$ and $[a] = \{1, \ldots, a\}$. All logarithms are base two unless stated otherwise. For $m \in \mathbb{N}$, $\{0,1\}^m$ is the set of all binary strings of length m. A string $x \in \{0,1\}^m$, $x = x_1 x_2 \cdots x_m$, represents the number $\sum_{j \in [m]} x_j 2^{m-j}$ in binary, and we often identify the string with that number. As an integer has multiple binary representations differing only in the number of leading zeroes, the number of leading zeroes should be clear from the context. The most significant bit of $x = x_1 x_2 \cdots x_m$ is x_1 and the least significant bit of x is x_m. Symbol $\circ$ denotes the concatenation of two strings. For strings $x, y \in \{0,1\}^m$, $x \oplus y$ denotes the bit-wise XOR of x and y, $x \wedge y$ denotes the bit-wise AND, and $x \vee y$ the bit-wise OR.

We assume the reader is familiar with boolean circuits (see for instance the book of Jukna [2012]). We assume boolean circuits consist of gates computing binary AND and OR, and unary gates computing negation. For us, boolean circuits might have multiple outputs so a circuit can compute a function $f\colon \{0,1\}^n \to \{0,1\}^m$ has n inputs and m outputs. We usually index a circuit family by multiple integral parameters (the number of input and output bits and potentially other integer parameters). Inputs and outputs of boolean circuits are often interpreted as sequences of substrings, e.g., a circuit $C_{n,m}\colon \{0,1\}^{nm} \to \{0,1\}^{nm}$ is viewed as taking n binary strings of length m as its input, and similarly for its output. We say that a circuit family $(C_n)_{n \in \mathbb{N}}$ is uniform, if there is an algorithm that on input 1^n outputs the description of the circuit C_n in time polynomial in n (and similarly if there are more parameters the algorithm gets as an input the unary representation of each parameter of the circuit it should return).

2.3 Preliminaries

Here we review some of the circuits for basic primitives that we will use in our later constructions. Most of them are well known facts.

Lemma 2.3.1 (Addition). *There is a uniform family of boolean circuits*

$$\mathrm{ADD}_m\colon \{0,1\}^{2m} \to \{0,1\}^{m+1}$$

that given $x, y \in \{0,1\}^m$ representing two numbers in binary outputs their sum $x + y \in \{0,1\}^{m+1}$. The circuit ADD_m has size $\Theta(m)$ and depth $\Theta(\log(m))$.

Lemma 2.3.2 (Subtraction). *There is a uniform family of boolean circuits*

$$\mathrm{SUB}_m\colon \{0,1\}^{2m} \to \{0,1\}^{m}$$

that given $x, y \in \{0,1\}^m$ representing two numbers in binary outputs the absolute value of their difference $|x-y| \in \{0,1\}^m$. The circuit SUB_m has size $\Theta(m)$ and depth $\Theta(\log(m))$.

Lemma 2.3.3 (Summation). *There is a uniform family of boolean circuits*

$$\mathrm{SUM}_{n,m}\colon \{0,1\}^{nm} \to \{0,1\}^{\lceil\log(n)\rceil+m}$$

that given $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ interpreted as n numbers, each of m bits, outputs their sum $\sum_{j=1}^n x_j$. The circuit $\mathrm{SUM}_{n,m}$ has size $\Theta(nm)$ and depth $\Theta(\log(n) + \log(m))$.

Proof. We sketch the construction following the technique of Wallace [1964]. Given three numbers $x, y, z \in \{0,1\}^k$ in constant depth and using $\Theta(k)$ gates we can compute $p, q \in \{0,1\}^{k+1}$ such that $x + y + z = p + q$. Here, p is the coordinate-wise addition without carry, i.e., $0 \circ (x \oplus y \oplus z)$, and q is the carry, i.e., $((x \wedge y) \vee (x \wedge z) \vee (y \wedge z)) \circ 0$. Thus as long as there are at least three numbers to sum we can use this to transform x, y, z which are represented by $3k$ bits into p, q which are represented by $2k+2$ bits and continue summing those. Doing this in parallel for disjoint triples of summands after $\mathcal{O}(\log_{3/2}(n)) = \mathcal{O}(\log(n))$ rounds we are left with just two numbers and we sum those using Lemma 2.3.1. ☐

Lemma 2.3.4 (Comparator). *There is a uniform family of boolean circuits*

$$\mathrm{SWITCH}_m\colon \{0,1\}^{2m} \to \{0,1\}^{2m}$$

that given two numbers $x, y \in \{0,1\}^m$ outputs these two numbers sorted as integers, i.e., $\min(x,y) \circ \max(x,y)$. The size of the circuit SWITCH_m is $\Theta(m)$ and depth is $\Theta(\log(m))$.

Technique similar to the proof of the next lemma will be used also later in the proofs of Lemma 2.1.2 and Lemma 2.5.1 in order to achieve smaller circuit size. The main idea is to split inputs into smaller blocks and process the blocks independently by smaller circuits.

Lemma 2.3.5 (Binary to unary)**.** *There is a uniform family of boolean circuits* $\text{ONES}_b \colon \{0,1\}^{b+1} \to \{0,1\}^{2^b}$ *such that for any number* $x \in \{0,1\}^{b+1}$ *represented in binary the output consists of* x *ones followed by* $2^b - x$ *zeroes, provided* $x \leq 2^b$.[1] *The circuit* ONES_b *has size* $\Theta(2^b)$ *and depth* $\Theta(\log(b))$.

Proof. We first show how to construct (ONES'_b) a uniform family of boolean circuits which computes the same function, has the same size but depth $\mathcal{O}(b)$. Then we use $\text{ONES'}_{\log(b)}$ to construct the desired circuit ONES_b.

The main idea of the construction of ONES'_b is to recursively split the number x into two numbers x_L, x_R which describe how many bits set to one there should be in the first and the second half of the output.

Each of the two numbers x_L, x_R will be represented by b bits with the convention that if the most significant bit is equal to one then the number is a power of two (corresponding to all output bits in this part of the output set to one). We recursively split the numbers x_L, x_R in the same fashion until the numbers are represented by a single bit each at which point they will represent the output bits. We set

$$\begin{aligned} x_L &= \min(2^{b-1}, x) \\ x_R &= \min(2^{b-1}, \max(0, x - 2^{b-1})) && \text{(Using Lemma 2.3.2)} \end{aligned}$$

note that if the number x is represented by $(b+1)$ bits ($x \in \{0,1\}^{b+1}$) then the numbers x_L, x_R can be represented by b bits ($x_L, x_R \in \{0,1\}^b$) by their definition. Given $x \in \{0,1\}^{b+1}$ we can compute the maximum and minimum defining x_L, x_R by inspecting the two most significant bits of x:

- If the most significant bit of x is set to one (thus $x = 2^b$) we set $x_L = x_R = x/2$ each a power of two with the most significant bit set to one (and represented by a binary string $1 \circ 0^{b-1}$).
- If the most significant bit of x is set to zero and the second most significant bit is set to one, then x_L will be set to the binary number $1 \circ 0^{b-1}$ and x_R will be $x - x_L$ (a copy of x without the second most significant bit of x).
- If the two most significant bits of x are equal to zero then $x_L = x$ (represented by one less bit than x) and $x_R = 0$.

See Figure 2.1 for an example of splitting of x into x_L, x_R.

Thus we can compute the transformation $x \mapsto (x_L, x_R)$ where $x \in \{0,1\}^{b+1}$ and $x_L, x_R \in \{0,1\}^b$ using a circuit of size $\Theta(b)$ and depth $\Theta(1)$. Then each of the numbers x_L, x_R is again split into two, etc. until we get single bit numbers which represent the final output. The depth of the circuit ONES'_b is $\Theta(b)$ as each splitting can be done in constant depth. If the circuit splitting $b+1$ bits into two b-bit numbers has size $s(b) \leq cb + d$, for some universal constants c and

[1] We allow x to have $b+1$ bits but upper bound it from above by 2^b as we need to represent exactly $2^b + 1$ numbers from $\{0, 1, \ldots 2^b\}$.

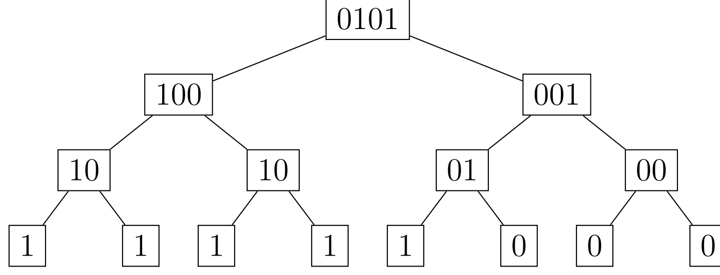

Figure 2.1: An example of splitting numbers where $b = 3$. The input number $x = 5$ is represented as 0101 and is split into $x_L = 100, x_R = 001$ which are themselves split recursively. The bottom nodes form the output.

d, then the circuit ONES'$_b$ has size:

$$\begin{aligned} s(b+1) + 2s(b) + 4s(b-2) + \ldots + 2^b s(1) &= \sum_{j=0}^{b} 2^j s(b-j) \\ &\leq \sum_{j=0}^{b} 2^j c(b-j) + 2^j d \\ &\leq c\left(2^{b+1} - b - 2\right) + 2^{b+1} d \\ &= \mathcal{O}(2^b) \end{aligned}$$

To build the circuit ONES$_b$ of depth $\mathcal{O}(\log(b))$ we proceed as follows. For any $y \geq 2$ we denote the largest power of two that is at most y by $\ell(y) = \max\{2^j \mid j \in \mathbb{N}, 2^j \leq y\}$. We divide the output bits into blocks of $\ell(b)$ bits and for each block $j \in \left[\frac{2^b}{\ell(b)}\right]$ of output bits with positions $[(j-1)\ell(b)+1, j\ell(b)]$ (counting positions from one) we compute if it should be constant (that is either constant zero when $x \leq (j-1)\ell(b)$ or constantly equal to one when $x > j\ell(b)$). This check for constant values can be done in each block by a circuit of size $\Theta(b)$ and depth $\Theta(\log(b))$. We compute ONES'$_{\log(\ell(b))}$ with the input being the $\log(\ell(b))$ least significant bits of x. This circuit is of size $\mathcal{O}(b)$ and depth $\mathcal{O}(\log(b))$. In each block if the block should not be monochromatic then we use the output of that circuit as the output of the block, otherwise we use the appropriate constant one or zero copied $\ell(b)$-times as the output of the block. □

We need a primitive that counts the number of occurrences of each string in the input. A counting similar to Lemma 2.3.6 appears in Appendix A of the paper of Asharov et al. [2021]. The construction of the counting circuit is rather straightforward, we just compare each input string x_j with a given string y getting an indicator bit set to one for equality and to zero for inequality and then sum the indicator bits.

Lemma 2.3.6 (Count)**.** *There is a uniform family of boolean circuits*

$$\text{COUNT}_{n,m} \colon \{0,1\}^{nm} \to \{0,1\}^{2^m \lceil 1+\log(n) \rceil}$$

that given $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ counts the number of occurrences of each $y \in \{0,1\}^m$ among the inputs, i.e., the circuit outputs $n_{0^m}, n_{0^{m-1}1}, \ldots, n_{1^m}$ where for each $y \in \{0,1\}^m$, n_y represents in binary $|\{j \in [n] \mid y = x_j\}|$ using $\lceil 1 + \log(n) \rceil$

bits. The size of the circuit $\mathrm{COUNT}_{n,m}$ *is* $\mathcal{O}(nm2^m)$ *and depth* $\mathcal{O}(\log(n) + \log(m))$*.*

Proof. For each $y \in \{0,1\}^m$ we build a sub-circuit computing the number of times y occurs among the inputs $x_1, \dots, x_n$. This is done by comparing y to each x_i in parallel, $i \in [n]$, to get an indicator bit whether they are equal. We obtain n_y by summing up the indicator bits using the circuit $\mathrm{SUM}_{n,1}$ of size $\Theta(n)$ and depth $\Theta(\log(n))$ from Lemma 2.3.3. Comparing y to x_i can be done by a circuit of size $\Theta(m)$ and depth $\Theta(\log(m))$ and we use n such subcircuits per each y. So we get n_y using a circuit of size $\Theta(nm)$ and depth $\Theta(\log(n) + \log(m))$. Doing this for each $y \in \{0,1\}^m$ in parallel we get a circuit of size $\Theta(nm2^m)$ and depth $\Theta(\log(n) + \log(m))$. □

We need also an inverse operation for the counting. To construct a circuit that decompresses the counts we would like to first compute the interval where a given string x should appear (that is the indexes of the first and last appearance of x) and then get indicator bits for this interval (zeroes for the indexes where x should not appear and ones for the indexes where x should appear). We can compute the interval using prefix sums of the counts. To get the indicator bits for the interval we utilize the circuit from Lemma 2.3.5 which outputs a given number of bits set to one followed by bits set to zero.

Lemma 2.3.7 (Decompress)**.** *There is a uniform family of boolean circuits*

$$\mathrm{DECOMPRESS}_{n,m}\colon \{0,1\}^{\lceil 1+\log(n)\rceil 2^m} \to \{0,1\}^{nm}$$

that decompresses its input *that is on input numbers* $n_{0^m}, n_{0^{m-1}1}, \dots, n_{1^m}$*, each represented in binary by* $\lceil 1 + \log(n)\rceil$ *bits, where* $\sum_{x\in\{0,1\}^m} n_x = s \le n$*, outputs the string*

$$(\underbrace{00\cdots0}_{m})^{n_{00\cdots0}} \circ (\underbrace{00\cdots0}_{m-1}1)^{n_{00\cdots01}} \circ \cdots \circ (\underbrace{11\cdots1}_{m})^{n_{11\cdots1}} \circ (0^m)^{n-s}.$$

When $s > n$ *the output might be arbitrary. The size of* $\mathrm{DECOMPRESS}_{n,m}$ *is* $\mathcal{O}(nm2^m + 2^{2m}\log(n))$ *and depth* $\mathcal{O}(m + \log\log(n))$*.*

Proof. Given $n_{0^m}, n_{0^{m-1}1}, \dots, n_{1^m}$ we can compute the total sum $s = \sum_{x\in\{0,1\}^m} n_x$ and for each $y \in \{0,1\}^m$, the number p_y of binary strings[2] before the first occurrence of y, i.e., $p_y = \sum_{x\in\{0,1\}^m:\, x<y} n_x$. Each of the numbers p_y can be computed using the circuit $\mathrm{SUM}_{y,\lceil 1+\log(n)\rceil}$ from Lemma 2.3.3 of size $\mathcal{O}(2^m\log(n))$ and depth $\mathcal{O}(m + \log\log(n))$. Similarly for s. Thus we can get all numbers p_y in parallel by a circuit of size $\mathcal{O}(2^{2m}\log(n))$. A given string $y \in \{0,1\}^m$, $y \neq 1^m$, should appear at each position $j \in [p_y + 1, p_{y+1}]$. Let $I_y \in \{0,1\}^n$ be the indicator vector of positions where y should appear in the output. We can use $\mathrm{ONES}_{\lceil 1+\log(n)\rceil}(p_y) \oplus \mathrm{ONES}_{\lceil 1+\log(n)\rceil}(p_{y+1})$ to calculate I_y for each $y \neq 1^m$. For $y = 1^m$, $I_y = \mathrm{ONES}_{\lceil 1+\log(n)\rceil}(p_y) \oplus \mathrm{ONES}_{\lceil 1+\log(n)\rceil}(s)$. The size of $\mathrm{ONES}_{\lceil 1+\log(n)\rceil}$ is $\Theta(n)$. As there are 2^m different y's, we need a circuit of size $\Theta(n2^m)$ and depth $\Theta(\log\log(n))$ to calculate all I_y's.

[2]There are no binary strings lexicographically smaller than 0^m so $p_{0^m} = 0$, in other words an empty sum is equal to zero.

If $x_1, x_2, \ldots, x_n$ are the output integers, for each output position $j \in [n]$, we calculate the k-th bit of x_j as

$$\bigvee_{y \in \{0,1\}^m} ((I_y)_j \wedge y_k)$$

To compute all of these ORs we need a circuit of total size $\Theta(nm2^m)$ and depth $\Theta(m)$. □

2.4 Sorting Circuits Based on AKS Network

In this section we recall the construction of circuits for sorting from the Ajtai et al. [1983] sorting networks. They will serve as the basic primitive for our later constructions.

Sorting networks. Sorting networks model parallel algorithms that sort values using only comparisons. A sorting network consists of n wires and s comparators. The wires extend from left to right in parallel. Each wire carries an integer from left to right. Any two wires can be connected by a comparator at any point along their length. The comparator swaps the values carried along the two wires if the higher wire carries a higher value at that point otherwise it has no effect. The sorting network should be such that when we input arbitrary integers to the wires on the left, the integers always exit in sorted order from top to bottom on the right side. The *depth* of a sorting network is the maximum number of comparators a value can encounter on its way. For a formal definition see, e.g., Ajtai et al. [1983]. Observe that if the depth of a sorting network is d and the number of inputs is n then there are at most $s \leq nd$ comparators. Ajtai et al. [1983] established the existence of sorting networks of logarithmic depth. An example of a small sorting network is given in Figure 2.2.

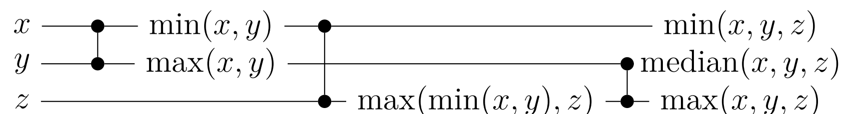

Figure 2.2: An example of a sorting network with three inputs (the horizontal lines), three comparators (the vertical lines), and depth three. The inputs on the left are numbers x, y, z and after each comparator we noted what is on the horizontal line. Note that the bottom most output is $\max(\max(x, y), \max(\min(x, y), z)) = \max(x, y, z)$ and the middle one is $\min(\max(x, y), \max(\min(x, y), z))$ which is the median.

Theorem 2.4.1 (Ajtai et al. [1983]). *For any integer $n \geq 1$, there is a sorting network for n integers of depth $\mathcal{O}(\log(n))$.*

Sorting circuits. Here we give a precise definition of sorting by a circuit. First we consider a circuit sorting n integers, each of them m bits long.

Definition 2.4.2 (Sort). *Let $n, m \in \mathbb{N}$, and $(C_{n,m})$ be a family of boolean circuits. We say that the* circuit $C_{n,m} \colon \{0,1\}^{nm} \to \{0,1\}^{nm}$ sorts its input *interpreted as n integers $x_1, x_2, \ldots, x_n$ each represented by m bits if it outputs $y_1, y_2, \ldots, y_n \in \{0,1\}^m$ such that:*

1. *The outputs are sorted: For any $i < j \in [n]$, $y_i \leq y_j$.*
2. *The inputs and outputs form the same multiset:*

$$\text{For each } z \in \{0,1\}^m \text{ we have } |\{i \in [n] \mid y_i = z\}| = |\{i \in [n] \mid x_i = z\}| .$$

An immediate consequence of the existence of AKS sorting networks is the existence of shallow sorting circuits, since by Lemma 2.3.4, each comparator can be replaced by a small circuit:

Corollary 2.4.3. *There is a family of boolean circuits*

$$\mathrm{AKS}_{n,m} \colon \{0,1\}^{nm} \to \{0,1\}^{nm}$$

that on an input $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ sorts these numbers. The size of the circuit $\mathrm{AKS}_{n,m}$ is $\mathcal{O}(nm\log(n))$ and depth $\mathcal{O}(\log(n)\log(m))$.

We also need circuits that sort the n input integers, each of m bits, by the k most significant bits where $k < m$. Such sorting can be thought of as sorting (key, value) pairs, where keys are k-bit long and values $(m-k)$-bit long. Formally it can be defined as follows:

Definition 2.4.4 (Partial Sort). *Let $n, m, k \in \mathbb{N}$, be such that $k < m$, and let $(C_{n,m,k})$ be a family of boolean circuits. We say that the* circuit $C_{n,m,k} \colon \{0,1\}^{nm} \to \{0,1\}^{nm}$ partially sorts by the first k bits its input *interpreted as n integers $x_1, x_2, \ldots, x_n$ each represented by m bits if it outputs $y_1, y_2, \ldots, y_n \in \{0,1\}^m$ such that:*

1. *The outputs are partially sorted: For any $i < j \in [n]$, $(y_i)_1(y_i)_2 \cdots (y_i)_k \leq (y_j)_1(y_j)_2 \cdots (y_j)_k$. That is if y_i, y_j are two output numbers where $i < j$ then we have $\lfloor y_i/2^{m-k} \rfloor \leq \lfloor y_j/2^{m-k} \rfloor$.*
2. *The inputs and outputs form the same multiset:*

$$\text{For each } z \in \{0,1\}^m \text{ we have } |\{i \in [n] \mid y_i = z\}| = |\{i \in [n] \mid x_i = z\}| .$$

Using a circuit of size $\mathcal{O}(m)$ and depth $\mathcal{O}(\log(k))$ implementing a comparator which swaps two m-bit integers based only on the first k bits we get the following variant of the previous corollary.

Corollary 2.4.5. *There is a family of boolean circuits*

$$\mathrm{PARTIAL_AKS}_{n,m,k} \colon \{0,1\}^{nm} \to \{0,1\}^{nm},$$

for $k \leq m$ and $k \leq \log(n)$, that on input $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ partially sorts these numbers according to their k most significant bits. The size of the circuit $\mathrm{PARTIAL_AKS}_{n,m,k}$ is $\mathcal{O}(nm\log(n))$ and depth $\mathcal{O}(\log(n)\log(k))$.

2.5 Sorting n Binary Strings of Length m

Here we present a sorting circuit for short numbers. The construction consists of two circuits. The first circuit counts the number of occurrences of various strings (as stated in Lemma 2.1.2) and the second circuit decompresses these counts. Both of these constructions use heavily the following technique: we divide the problem into blocks which can be efficiently sorted using the AKS-based circuit. These blocks will be of size between $2^{\mathcal{O}(m)}$ and $n/2^{\mathcal{O}(m)}$ where m is the binary length of the input integers.

We sort the numbers inside each block and subdivide the block into parts, in such a way that by the pigeon-hole principle most of the parts will be monochromatic (containing copies of a single string only). We can then separately count the strings in monochromatic parts (count the first string and then multiply that by the length of the part) and in the non-monochromatic parts (there are not that many strings in total in non-monochromatic parts). However a priori we do not know which parts turns out to be monochromatic and which does not. To save on circuitry we use sorting (on whole parts) to move the non-monochromatic parts aside. We build the (expensive) counting circuits only for non-monochromatic parts.

Lemma 2.1.2. (Restated) *For any integers $n, m \geq 1$ where $m \leq \log(n)/10$ there is a circuit*

$$\text{FAST_COUNT}_{n,m}\colon \{0,1\}^{nm} \to \{0,1\}^{2^m\lceil 1+\log(n)\rceil}$$

which given a sequence of n strings of m bits each outputs the number of occurrences of each possible m-bit string among the inputs, that is for input

$$x_1, x_2, \dots, x_n \in \{0,1\}^m$$

it outputs $n_{0^m}, n_{0^{m-1}1}, \dots, n_{1^m}$ where for each string $y \in \{0,1\}^m$, the output string $n_y \in \{0,1\}^{\lceil 1+\log(n)\rceil}$ represents $|\{j \in [n] \mid x_j = y\}|$ in binary. The size of the circuit $\text{FAST_COUNT}_{n,m}$ is $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$.

Proof. For the sake of simplicity let us assume that n is a power of two, so it is divisible by 2^{8m}. (By our assumption $n \geq 2^{10m}$, thus if n is not a power of two take the circuit for the closest power of two larger than n and feed ones for the extra input bits.) We partition the input into $n/2^{8m}$ blocks each consisting of 2^{8m} numbers. We sort each block in parallel by the circuit $\text{AKS}_{2^{8m},m}$ of size $\mathcal{O}(2^{8m}m\log(2^{8m})) = \mathcal{O}(2^{8m}m^2)$ and depth $\mathcal{O}(m\log(m))$ as given in Corollary 2.4.3. Thus for this phase we need a circuit of total size $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(m\log(m))$.

Then we subdivide each block into 2^{6m} parts each consisting of 2^{2m} numbers. Observe that most of these parts are monochromatic: a part is monochromatic if it contains 2^{2m} copies of a single m-bit number. We can upper bound the number of non-monochromatic parts by 2^m. We can add a single indicator bit to each part indicating whether this part is monochromatic. As the parts are sorted it is enough to compare the first and last number in each part and set the bit to 1 if the numbers are equal and to 0 otherwise. Inside each block we sort the parts prefixed by their indicator bit using the circuit $\text{PARTIAL_AKS}_{2^{6m},1+m2^{2m},1}$ from

Corollary 2.4.5 to move all non-monochromatic parts to the front of each block. Thus the total size of the circuit sorting parts inside each block is

$$\mathcal{O}\left(\frac{n}{2^{8m}}(2^{6m})(1+m2^{2m})6m\right) = \mathcal{O}(nm^2)$$

and depth $\mathcal{O}(m)$. We call the first 2^m parts of each block *potentially non-monochromatic*. The other parts are *definitely monochromatic*.

From each definitely monochromatic part we take the first m-bit number and we count them. This can be done by the circuit $\mathrm{COUNT}_{\frac{n}{2^{8m}}(2^{6m}-2^m),m}$ from Lemma 2.3.6 of size $\mathcal{O}\left(\left(\frac{n}{2^{2m}} - \frac{n}{2^{7m}}\right)m2^m\right) \leq \mathcal{O}(nm)$ and depth $\mathcal{O}(\log(n) + \log(m))$. By multiplying each count by 2^{2m} (that is by appending $2m$ zeroes) we get the number of occurrences of each number in the definitely monochromatic parts.

As there are relatively few (exactly $\frac{n}{2^{8m}}2^m2^{2m}$) numbers overall in potentially non-monochromatic parts we can use the circuit $\mathrm{COUNT}_{n/2^{5m},m}$ from Lemma 2.3.6 to count those numbers by a circuit of size $\mathcal{O}\left(\frac{n}{2^{5m}}m2^m\right) \leq \mathcal{O}(nm)$ and depth $\mathcal{O}(\log(n) + \log(m))$.

We got two vectors of counts for numbers in potentially non-monochromatic and definitely monochromatic blocks. Finally, we add the two vectors of 2^m numbers each consisting of at most $\lceil 1 + \log(n) \rceil$ bits to get the resulting counts. This uses a circuit of size $\mathcal{O}(m2^m) = \mathcal{O}(n)$ and depth $\mathcal{O}(\log\log(n))$. Thus, the overall size of the circuit is $\mathcal{O}(nm^2)$ and depth $\mathcal{O}(\log(n) + m\log(m))$. □

Lemma 2.5.1. *For integers $n, m \geq 1$ such that $m \leq \log(n)/11$, there is a family of boolean circuits*

$$\mathrm{FAST_DECOMPRESS}_{n,m} \colon \{0,1\}^{2^m\lceil 1+\log(n)\rceil} \to \{0,1\}^{nm}$$

that decompresses its input as in Lemma 2.3.7.

The size of the circuit $\mathrm{FAST_DECOMPRESS}_{n,m}$ *is $\mathcal{O}(nm^2)$ and its depth is $\mathcal{O}(m\log(m) + \log\log(n))$.*

The construction of the decompression circuit mirrors the counting circuit albeit it is somewhat simpler with a different choice of parameters. We separately decompress monochromatic blocks (by decompressing just a single string from each block and then creating the right number of copies) and the strings from non-monochromatic blocks (as there are not many of those). We then use partial sorting to rearrange the blocks in the proper order to construct a sorted sequence.

Proof. For the sake of simplicity let us assume that n is a power of two and let us set $k = n/2^{8m}$. (Thus k is an integer.) We think of the output as partitioned into 2^{8m} blocks of size k. As in the proof of Lemma 2.3.7 we compute the prefix sums

$$p_x = \sum_{y \in \{0,1\}^m \colon y < x} n_y \qquad \text{for each } x \in \{0,1\}^m$$

and we set $p_{2^m} = n$. (Here, we identify m-bit strings x and y with integers they represent.) We can compute each p_x using the circuit $\mathrm{SUM}_{2^m,1+\log(n)}$, thus computing all of them using a circuit of size $\mathcal{O}(\log(n)2^{2m}) \leq \mathcal{O}(n)$ (by the assumption

$m \leq \log(n)/11$) and depth $\mathcal{O}(m + \log\log(n))$. Thus the string $x \in \{0,1\}^m$ should appear at output positions $[p_x + 1, p_{x+1}]$. For any $x \in \{0,1\}^m$ we set:

$$r_x = ((k - (p_x \mod k)) \mod k) + (p_{x+1} \mod k)$$
$$q_x = \frac{n_x - r_x}{k}$$

The meaning is that if we partition the output into blocks of k consecutive numbers, then for any $x \in \{0,1\}^m$ the number r_x tells the number of times the string x appears in non-monochromatic blocks. (These occurrences are located in at most two non-monochromatic blocks.) The number q_x tells us in how many monochromatic blocks the string $x \in \{0,1\}^m$ appears. Observe that q_x is an integer. Since n is a power of two, so is k, furthermore, k is fixed for given n and m, and thus computing mod k and division by k corresponds to selecting appropriate bits from the binary representation of numbers. All numbers p_x, q_x and r_x are integers represented by $1 + \log(n)$ bits. Hence, each q_x and r_x can be computed from n_x and p_x by one circuit $\mathrm{ADD}_{1+\log(n)}$ and two $\mathrm{SUB}_{1+\log(n)}$. The circuit computing values q_x and r_x for all x has total size $\mathcal{O}(2^m \log(n))$ and depth $\mathcal{O}(\log\log(n))$.

The following holds:

$$n_x = kq_x + r_x$$
$$\sum_{x \in \{0,1\}^m} q_x = \sum_{x \in \{0,1\}^m} \frac{n_x - r_x}{k} \leq n/k = 2^{8m}$$
$$\sum_{x \in \{0,1\}^m} r_x \leq 2k2^m = 2n/2^{7m}$$

We use circuit $\mathrm{DECOMPRESS}_{2^{8m},m}(q_{0^m}, q_{0^{m-1}1}, \ldots, q_{1^m})$ from Lemma 2.3.7 of size $\mathcal{O}\left(m2^{9m}\right)$ and depth $\mathcal{O}\left(m\right)$ to decompress monochromatic blocks. We then just copy each resulting number k times to create sorted monochromatic blocks. Last $2^{8m} - \sum_{x \in \{0,1\}^m} q_x$ blocks contain zero padding corresponding to the numbers in non-monochromatic blocks. They will be merged with the non-monochromatic blocks obtained next.

In order to properly match the non-monochromatic blocks to the padded zeroes we adjust the count r_{0^m}:

$$r'_{0^m} = \left(2n/2^{7m}\right) - \sum_{x \in \{0,1\}^m :\, x \neq 0^m} r_x$$

using circuit $\mathrm{SUM}_{2^m,1+\log(n)}$ and $\mathrm{SUB}_{1+\log(n)}$ of size $\mathcal{O}(n)$ and depth $\mathcal{O}(m + \log\log(n))$. We use the circuit $\mathrm{DECOMPRESS}_{2n/2^{7m},m}(r'_{0^m}, r_{0^{m-1}1}, \ldots, r_{1^m})$ from Lemma 2.3.7 to decompress the non-monochromatic blocks. The circuit is of size $\mathcal{O}\left(\left(2n/2^{7m}\right)m2^m + 2^{2m}\log\left(2n/2^{7m}\right)\right) \leq \mathcal{O}\left(nm/2^{6m}\right)$ and of depth $\mathcal{O}(m + \log\log(n))$. (Here, we used our assumption $m \leq \log(n)/11$, to bound $n \geq 2^{11m}$ and $2^{2m} \leq n^{3/4}/2^{6m}$.)

Finally, we compute the bit-wise OR of the last 2^{m+1} blocks of the output from the previous step (monochromatic decompression) with the current output (non-monochromatic decompression). This way we get a sequence of n numbers

partitioned into blocks where each block corresponds to one of the blocks in the desired output. However, we still need to rearrange the blocks in the proper order. We will use partial sorting of the whole blocks to do that.

For a given block let x be the first number in that block. We prefix the block by a number $2x$ (represented by $m+1$ bits) if the block is monochromatic or the number $2x+1$ if the block is non-monochromatic. To determine whether the block is monochromatic we compare for equality the first and last number inside the block. We do this for each block. Thus each block of k numbers is prefixed by an $m+1$ bit number. Computing these prefixes requires a circuit of total size $\mathcal{O}(2^{8m}m) \leq \mathcal{O}(n)$ and depth $\mathcal{O}(\log(m))$. We then use the $\mathrm{PARTIAL_AKS}_{2^{8m},(m+1)+km,m+1}$ circuit of size $\mathcal{O}(2^{8m}((m+1)+km)8m) \leq \mathcal{O}(nm^2)$ and depth $\mathcal{O}(m\log(m))$ to sort the blocks (see Corollary 2.4.5). Finally, we ignore the $m+1$ bit prefixes of each block to get the desired output. □

Proof of Theorem 2.1.1. It is enough to combine Lemma 2.1.2 and Lemma 2.5.1. □

Observe that the proofs of Lemma 2.1.2 and Lemma 2.5.1 do not depend on using specifically the AKS sorting. In particular for the case of Lemma 2.1.2 if there is a circuit that sorts input numbers that is linear in the number of input bits then there is a linear size circuit that counts these numbers.

2.6 Partial Sorting by the First k Bits in Polylogarithmic Depth

Here we design a family of boolean circuits that partially sorts by the first k bits out of m bits which is asymptotically smaller than $\mathrm{PARTIAL_AKS}_{n,m,k}$. Another natural way how to think about partial sorting by the first k bits is as sorting keys with payload data ($w-k$ bits) for each k-bit key. We will need super-concentrators for our construction.

A directed acyclic graph $G=(V,E,A,B)$, where V is the set of vertices, E is the set of directed edges, and A and B are disjoint subsets of vertices of the same size, is a *super-concentrator* if the following hold: The vertices in A (*inputs*) have in-degree zero, vertices in B (*outputs*) have out-degree zero, and for any $S \subseteq A$ and for any $T \subseteq B$: $|S|=|T|$ there is a set of pairwise vertex disjoint paths connecting each vertex from S to some vertex in T.

We parametrize the super-concentrator by the number of input vertices n, and we measure its size by the number of edges. We want the graph to have as few edges as possible. The depth of the super-concentrator is the number of edges on the longest directed path.

Pippenger [1996] shows a construction of super-concentrators of linear size and logarithmic depth. More specifically, Pippenger [1996] constructs a family of super-concentrators S_n for n being the number of inputs, where the in-degree and out-degree of each vertex is bounded by some universal constant, the number of edges is linear in n, and the depth is $\mathcal{O}(\log(n))$. Moreover he shows that there are finite automatons which for any $S \subset A, T \subset B$: $|S|=|T|$ when put on the vertices of the super-concentrator find the set of vertex disjoint paths from S to T in $\mathcal{O}(\log(n))$ iterations, each taking $\mathcal{O}(\log(n))$ steps, for the total number of

$\mathcal{O}(n)$ steps of the automatons. We describe this construction using the language of circuits. The circuit on input of characteristic vectors of S and T computes the set of $|T|$ vertex disjoint paths connecting S and T. The circuit outputs the characteristic vector of the set of edges participating in the paths.

Theorem 2.6.1 (Pippenger [1996])**.** *There is a family of super-concentrators S_n as described above and boolean circuits*

$$\mathrm{ROUTE}_n \colon \{0,1\}^{2n} \to \{0,1\}^{|S_n|}$$

of size $\mathcal{O}(n \log(n))$ and depth $\mathcal{O}(\log^2(n))$ that on input characteristic vector of any set $T \subseteq [n]$ and characteristic vector of any $S \subseteq [n]$ where $|T| = |S|$, outputs the characteristic vector of edges that form $|T|$ vertex disjoint paths between S and T.

By routing m bits along each path in the super-concentrator we can use the above circuit to build a circuit that partially sorts m-bit integers by their most significant bit.

Corollary 2.6.2. *There is a family of boolean circuits*

$$\mathrm{PIPPENGER_SORT}_{n,m,1} \colon \{0,1\}^{nm} \to \{0,1\}^{nm}$$

that on input $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ partially sort these numbers according to their first most significant bit. The size of the circuit $\mathrm{PIPPENGER_SORT}_{n,m,1}$ is $\mathcal{O}(nm + n\log(n))$ and depth $\mathcal{O}(\log^2(n))$.

Proof. We give a sketch of the proof. First, we will use the graph S_n to get all inputs starting with one to the proper place. Then, using the same construction we will move all inputs starting by 0 to the proper place. We transform the graph S_n into a circuit by replacing each vertex of in-degree d by a *routing gadget* (circuit) which takes d m-bit inputs together with d control bits, one bit for each of the m-bit inputs, and outputs the bit-wise OR of inputs for which their control bit is set to 1. Such a routing gadget of size $\mathcal{O}(dm)$ and depth $\mathcal{O}(\log(d))$ can be easily constructed. If (u, v) is the j-th incoming edge of v in S_n, we connect the j-th block of m input bits of the routing gadget corresponding to v to the output of the routing gadget of u. The routing gadgets of input vertices of S_n are connected directly to the appropriate inputs of the sorting circuit. The routing gadget will be used with at most single control bit set to one, thus it will route the corresponding input.

It remains to calculate paths that will route the integers starting with 1 in the above circuit in the desired way. For that, we calculate the sum s of the most significant bits by which we are sorting using $\mathrm{SUM}_{n,1}$ from Lemma 2.3.3, we expand it back using $\mathrm{ONES}_{\lceil \log(n) \rceil + 1}(s)$, and reverse it to get the characteristic vector of a set T, where we want to route to. Together with the most significant bits of each input integer (which form the characteristic vector of S from which we route) we feed this as an input to ROUTE_n. The output bits of ROUTE_n are connected to the appropriate control bits of our routing gadgets. The sorted output is obtained as the output of the n routing gadgets corresponding to the output vertices of S_n.

The size of the ROUTE_n is $\mathcal{O}(n\log(n))$ and the total size of the circuits implementing the routing gadgets is $\mathcal{O}(mn)$. These two terms dominate the overall size of the circuit. The depth of the circuit is dominated by the depth of the ROUTE_n which is $\mathcal{O}(\log^2(n))$. □

We can use the above circuit in an iterative fashion to build a smaller circuit for the same primitive. Again we use blocks and parts where most parts are monochromatic. But in the following lemma we use larger and larger blocks (and parts) and monochromatic parts are sorted as one entity (the sorting works with whole monochromatic parts instead of individual inputs). Thus the route-counting part from Corollary 2.6.2 is relatively small as it is computing routing for a limited number of strings (inputs and later monochromatic parts) at a time.

Lemma 2.6.3. *There is a family of boolean circuits*

$$\mathrm{ITERATIVE_SORT}_{n,m,1}\colon \{0,1\}^{nm} \to \{0,1\}^{nm}$$

that on input $x_1, x_2, \ldots, x_n \in \{0,1\}^m$ partially sort these numbers according to their first most significant bit. The size of the circuit $\mathrm{ITERATIVE_SORT}_{n,m,1}$ is $\mathcal{O}(nm(1+\log^*(n)-\log^*(m)))$ and its depth is $\mathcal{O}(\log^2(n))$.

Proof. Assume $m \le \log(n)/11$ otherwise use Corollary 2.6.2. We will build the circuit iteratively using the circuit from Corollary 2.6.2 for blocks of various sizes. We will start with small blocks of items and we will iteratively sort larger and larger number of items organized into mostly monochromatic blocks. Without loss of generality we assume that m is a power of two, and we will ignore the rounding issues. We will have two parameters m_i and $n_i = 2^{3m_i}$, where $m_0 = m$ and $m_{i+1} = 2^{m_i}$ for $i \ge 0$. At iteration i, all the items will be partitioned into *parts* of consecutive numbers, each part will be either *monochromatic* containing all zeros, all ones, or it will be *mixed*. (Here we refer to the most significant bits of the numbers in the part.) For each part we will maintain two indicator bits which of the three possibilities occurs: an indicator which is one if the block is mixed, and another *color* indicator which specifies the highest order bit of the integers if the block is monochromatic. (For the latter we could use the first bit of the first integer in the part.) At each iteration $i > 0$, m_i will denote the number of items in each part. n_i/m_i consecutive parts form a *block*, so each block contains n_i items. The blocks partition the input. We will maintain an invariant that the fraction of mixed parts in each block is at most $2/m_i^3$ at the beginning of i-th iteration.

At iteration 0 we apply $\mathrm{PIPPENGER_SORT}_{n_0,m,1}$ to consecutive blocks of n_0 input integers. Afterwards, the block is partitioned into parts of size m_1 and for each part we determine its status by comparing the most significant bits of the first and last integer in the part. It is clear that each block of size n_0 contains at most one mixed part. As the number of parts in the block is m_1^3, the fraction of mixed parts in each block is at most $2/m_1^3$, and this is also true for blocks of size n_1.

At iteration $i > 0$, we divide the current sequence of parts of size m_i into blocks containing n_i/m_i parts, and we proceed in three steps:

Step 1. Sort the parts in each block using $\mathrm{PIPPENGER_SORT}_{n_i/m_i,2+m_i\cdot m,1}$ according to the mixed indicator. Hence, all the mixed parts will move to

the end of the block. There are at most $2n_i/m_i^3$ mixed parts in each block, the remaining parts must be monochromatic.

Step 2. In each block, sort all the m-bit integers in the last $2n_i/m_i^3$ parts according to their most significant bit using $\mathrm{PIPPENGER_SORT}_{2n_i/m_i^2,m,1}$. This sorts together all the integers in the mixed parts (and perhaps few other parts). Repartition them into parts of m_i consecutive numbers and determine their indicator bits. Only one of the parts should be mixed at this point. Swap it with the last part in the block. (We provide details of the swap later.)

Step 3. In each block, sort all the parts except for the last one according to their color indicator using $\mathrm{PIPPENGER_SORT}_{(n_i/m_i)-1,2+m_i\cdot m,1}$. This moves all the parts of color 0 to the front. Repartition all the numbers in the block into parts of m_{i+1} consecutive integers and determine their indicator bits, where the last part is marked as mixed. At most two of the new parts should be mixed at this point. Notice, that out of m_{i+1}^3 parts in each block, at most two are marked as mixed so the invariant applies. We can move to the next iteration.

We iterate the algorithm until $m_i \geq \log(n)/4$. Once $m_i \geq \log(n)/4$, the number of integers in mixed parts is at most $2n/m_i^2 \leq \mathcal{O}(n/\log^2(n))$, remaining items are in monochromatic parts. At this point we cannot form a block of size n_i, but we can still perform the same type of actions as in Steps 1-3: We can bring the monochromatic parts forward as in Step 1, sort the last $32n/\log^2(n)$ integers belonging to the mixed parts, move the remaining mixed part to the end, sort the monochromatic parts and swap the mixed part with the first monochromatic part of color 1.

To swap a single mixed part with the last part we can copy the mixed part into a buffer by AND-ing every part bit-wise with the indicator whether that is the mixed part, and OR-ing all the results together. This copies the mixed part into a buffer. In a similar fashion we can copy the last part into the now unused part by letting each part bit-wise copy to its place either its original content or the content of the last part, again conditioning on an appropriate indicator bit. Hence, the swap can be implemented by a circuit of size proportional to the total size of the parts and depth logarithmic in the number of parts.

Now we will bound the total size of the circuit we constructed. Step 1 requires n/n_i circuits of size $\mathcal{O}(n_i m + n_i/m_i \log(n_i/m_i)) = \mathcal{O}(n_i m)$, as $\log(n_i) = \mathcal{O}(m_i)$, and of depth at most $\mathcal{O}(\log^2(n_i))$. Step 2 requires n/n_i sorting circuits of size $\mathcal{O}(mn_i/m_i^2 + 2n_i/m_i^2 \log(2n_i/m_i^2)) = \mathcal{O}(n_i)$ and of depth at most $\mathcal{O}(\log^2(n_i))$, together with a circuit of total linear size $\mathcal{O}(n)$ to recalculate the parts and do the swaps. The last step requires the same amount of circuitry as the first step.

Hence, each step requires circuits of total size $\mathcal{O}(nm)$. The same goes for the initial sort at iteration 0, and the final sorts at the end. As there are at most $\log^*(n) - \log^*(m)$ iterations, the resulting size is $\mathcal{O}(nm(\log^*(n) - \log^*(m)))$. Each step requires a circuit of depth $\mathcal{O}(\log^2(n_i))$, recall that by our choice $n_i = 2^{3m_i}$, thus $\log(n_i) = 3m_i$. Since $m_{i+1} = 2^{m_i}$ and for each i we have $m_i \leq \log(n)/4$, thus the total depth is dominated by the last iteration where we use a circuit of depth $\mathcal{O}(\log^2(n))$. □

Theorem 2.1.3. (Restated) *For any integers $n, m, k \geq 1$ where $k \leq m$ and $k \leq \log(n)/11$ there is a circuit*

$$\text{SORT}_{n,m,k}\colon \{0,1\}^{nm} \to \{0,1\}^{nm}$$

which partially sorts n numbers each of m bits by their first k bits. The circuit $\text{SORT}_{n,m,k}$ has size $\mathcal{O}(knm(1+\log^*(n)-\log^*(m)))$ and depth $\mathcal{O}(\log^3(n))$.

As we have shown in Lemma 2.6.3 we can sort the integers by one bit. Unfortunately this sort is not stable thus we cannot simply use it to sort by k bits. The main idea is to sort only the keys (i.e., take only first k bits of each integer and sort them) and use this information for splitting the array into two halves – less than or equal and larger or equal to the median key. We observe that in one of the halves all elements have the same first bit (same as the most significant bit of the median) and thus need to be sorted by $k-1$ bits only. Thus we get two subproblems: sorting $n/2$ elements by k bits and sorting $n/2$ integers by $k-1$ bits. The recursion gives us a circuit of the desired size and depth.

Proof of Theorem 2.1.3. We assume that $k \leq \log(n)/11$ otherwise we can use Corollary 2.4.5 to sort the elements. Without loss of generality we assume n is a power of two. We think of the input as organized into an array. We extract the first k bits (*key*) from each input element and we sort the keys using the circuit from Theorem 2.1.1 of size $\mathcal{O}(nk^2)$ and depth $\mathcal{O}(\log(n) + k\log(k))$.

We build recursively a circuit that sorts the input array of n elements according to the first k bits when the input is augmented with the array of sorted keys. Now our goal is to split the input array into two equal sized parts L and R where all elements in L are less or equal to elements in R when comparing only the keys.

To do that we take the *median*, the $(n/2)$-th element among the keys, and we partition the array according to it. We split the input array into three arrays L, M, and R of length n with elements less than, equal to, and greater than the median, resp., and we mark the unused elements as *dummy* using an extra bit associated to each element (see Figure 2.3 for an illustration of the following process). In the case of the L array the dummy elements are there for each element with a key at least as large as the median key. In the case of the M array the dummy elements are there for each element with a key not equal to the median key. In the case of the R array the dummy elements are there for each element with a key at most as large as the median key. We sort L and M so that all non-dummy elements are to the left and R so that all non-dummy elements are to the right. We use three circuits $\text{ITERATIVE_SORT}_{n,m+1,1}$ to do that. Now, we flip the first half of elements in M, i.e., swap the i-th element with the element in position $(n/2) - i + 1$, and we replace the dummy elements in the first half of L by the corresponding elements in M. By one application of $\text{ITERATIVE_SORT}_{n,m+1,1}$ we move all the remaining non-dummy elements in M to the left, and we merge those elements with the second half of R. We discard the second and first half of L and R, respectively. (They contain only dummy elements.)

If the highest order bit of the median is set to 0 then all the elements in L have the highest order bit set to 0, otherwise all the elements in R have the highest order bit set to 1. In either case we reduced the problem to one problem

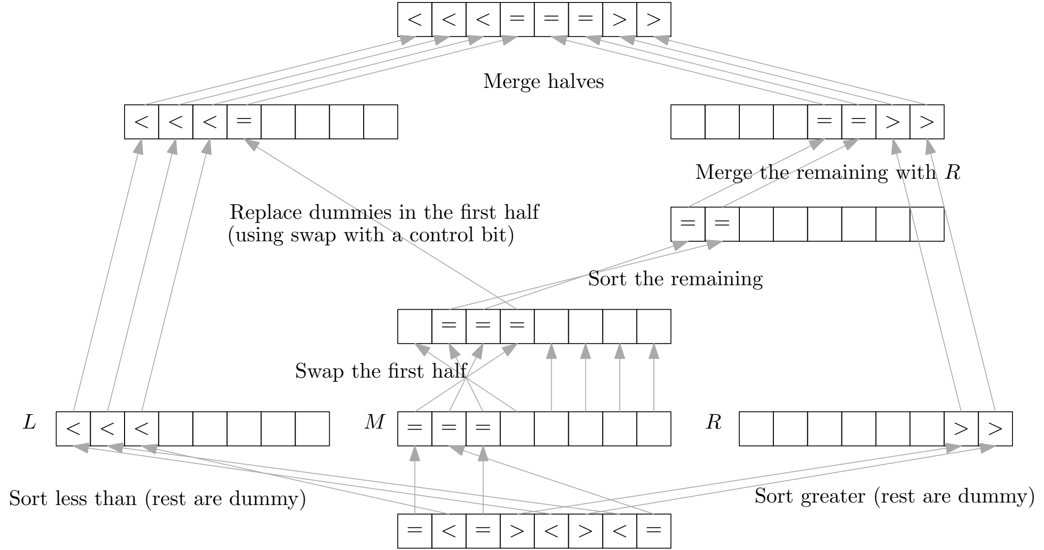

Figure 2.3: Sorting by the median key using unstable sorting by a single bit. Inputs are at the bottom and outputs are at the top. The symbol $<$ means an element with a key less than the median key, $=$ stands for an element with a key equal to the median key, and $>$ stands for an element with a key greater than the median key. Blanks are "dummy" elements and arrows are either use of sorting by a single bit or fixed permutations.

of sorting half of the elements according to $k-1$ bits and the other half according to k-bits. We recursively build a circuit to sort $\text{SORT}_{n/2,m,k-1}$ and $\text{SORT}_{n/2,m,k}$ when the input is augmented with the sorted array of keys. We pass to each of the sorting sub-circuits the appropriate sub-problem and we re-route the results from them to form the final output.

Not counting the two sub-circuits $\text{SORT}_{n/2,m,k-1}$ and $\text{SORT}_{n/2,m,k}$, this step requires four copies of the circuit $\text{ITERATIVE_SORT}_{n,m+1,1}$ and additional $\mathcal{O}(nm)$ gates to do the moves and element comparison with the median. Denote the size of this part of the circuit by

$$L_m(n) = \mathcal{O}(nm(1 + \log^*(n) - \log^*(m))).$$

The depth of the resulting circuit to perform all those operations is $\mathcal{O}(\log^2(n))$ as the move operations are done in parallel (again, not counting the depth of $\text{SORT}_{n/2,m,k-1}$ and $\text{SORT}_{n/2,m,k}$). If we denote by $S_{m,k}(n)$ the size of the circuit $\text{SORT}_{n,m,k}$ we get the following recurrence:

$$\begin{aligned}
S_{m,k}(1) &= \mathcal{O}(m) \\
S_{m,1}(n) &= \mathcal{O}(nm(1 + \log^*(n) - \log^*(m))) \\
S_{m,k}(n) &\leq L_m(n) + S_{m,k-1}\left(\frac{n}{2}\right) + S_{m,k}\left(\frac{n}{2}\right)
\end{aligned}$$

when we iterate the recurrence:

$$
\begin{aligned}
S_{m,k}(n) &= L_m(n) + S_{m,k-1}(n/2) + S_{m,k}(n/2) \\
&= L_m(n) + S_{m,k-1}(n/2) + L_m(n/2) + S_{m,k-1}(n/4) + S_{m,k}(n/4) \\
&= L_m(n) + S_{m,k-1}(n/2) + L_m(n/2) \\
&\quad + S_{m,k-1}(n/4) + L_m(n/4) + S_{m,k-1}(n/8) + S_{m,k}(n/8) \\
&= \ldots \\
&= (L_m(n) + L_m(n/2) + \ldots + L_m(1)) \\
&\quad + (S_{m,k-1}(n/2) + S_{m,k-1}(n/4) + \ldots + S_{m,k-1}(1)) + S_{m,k}(1) \\
&\leq L_m(2n) + S_{m,k-1}(n) + S_{m,k}(1)
\end{aligned}
$$

which gives us

$$
\begin{aligned}
S_{m,k}(n) &= kL_m(2n) + S_{m,1}(n) + (k-1)S_{m,k}(1) \\
&\leq \mathcal{O}(knm(1 + \log^*(n) - \log^*(m)))
\end{aligned}
$$

To bound the depth $D_{m,k}(n)$ we use the following recurrence:

$$
\begin{aligned}
D_{m,k}(1) &= \mathcal{O}(1) \\
D_{m,k}(n) &\geq D_{m,k-1}(n) \\
D_{m,1}(n) &= \mathcal{O}(\log^2(n)) \\
D_{m,k}(n) &= \mathcal{O}(\log^2(n)) + \max\left(D_{m,k}(n/2) + D_{m,k-1}(n/2)\right) \\
&\leq \mathcal{O}(\log^2(n)) + D_{m,k}(n/2) \\
&\leq \mathcal{O}(\log^3(n))
\end{aligned}
$$

□

2.7 Conclusion

We have provided improved sorting circuits. Our technique used in the proof of Theorem 2.1.1 can be viewed as information compression and decompression. This technique might prove useful for other related problems. We list some open problems:

- Most of our circuits are uniform. The non-uniform part is due to the use of the AKS circuits and Pippenger's super-concentrators. Can one make uniform circuits of the same size?

- Kospanov [1994] shows that there is a family of sorting circuits with depth $\mathcal{O}(\log(n)+\log(m))$ and size $\mathcal{O}(mn^2)$ that sorts n numbers each of m bits. Is there a circuit family for sorting with circuits of depth $\mathcal{O}(\log(n)+\log(m))$ and size $\mathcal{O}(nm^2)$? In other words can we get rid of the $m\log(m)$ factor in the circuit depth from Theorem 2.1.1 while keeping the $\mathcal{O}(nm^2)$ size?

- Is it possible to partially sort n numbers of m bits each by their first bit using a circuit of size $\mathcal{O}(nm)$ and depth $\mathcal{O}(\log(n))$?

3. Oblivious RAM

The results presented in this chapter are based on the paper Hubáček et al. [2019].

3.1 Our Results

In this work, we further develop the information transfer technique of Patrascu and Demaine [2006] when applied in the context of online ORAMs. We revisit the lower bound of Larsen and Nielsen [2018] which was proved under the assumption that the adversarial server knows exactly which server accesses correspond to each input operation. Specifically, we prove a stronger matching lower bound in a relaxed model without any restriction on the format of the access sequence to server memory.

Note that the Larsen and Nielsen [2018] lower bound does apply to the known constructions of ORAMs where it is possible to implicitly separate the accesses corresponding to individual input operations – since each input operation generates an access sequence of roughly the same length. However, the Larsen and Nielsen [2018] result does not rule out the possibility of achieving sub-logarithmic overhead in an ORAM which obfuscates the boundaries in the access pattern (e.g., by translating input operations into variable-length memory accesses). We show that obfuscating the boundaries between the input operations does not help in building a more efficient ORAM. In other words, our lower bound justifies the design choice of constructing ORAMs where each input operation is translated to roughly the same number of probes to server memory (common to the known constructions of ORAMs).

Besides online ORAM (i.e., the oblivious Array Maintenance problem), our techniques naturally extend to other oblivious data structures and allow to generalize also the recent lower bounds of Jacob et al. [2019] for oblivious stacks, queues, deques, priority queues and search trees.

For online ORAMs with statistical security, our results are stated in the following informal theorem.

Theorem 3.1.1 (Informal). *Any statistically secure online ORAM with internal memory of size m has expected bandwidth overhead $\Omega(\log(n))$, where $n \geq m^2$ is the length of the sequence of input operations. This result holds even when the adversarial server has no information about boundaries between probes corresponding to different input operations.*

In the computational setting, we consider two definitions of computational security. Our notion of *weak computational security* requires that no polynomial time algorithm can distinguish access sequences corresponding to any two input sequences of the same length – this is closer in spirit to computational security for ORAMs previously considered in the literature. The notion of *strong computational security* requires computational indistinguishability even when the distinguisher is given the two input sequences together with an access sequence corresponding to one of them. The distinguisher should not be able to tell which one of the two input sequences produced the access sequence. Interestingly, our technique (as well as the proof technique of Larsen and Nielsen [2018] in the

model with structured access pattern) yields different lower bounds with respect to the two definitions stated in the following informal theorem.

Theorem 3.1.2 (Informal)**.** *Any weakly computationally secure online ORAM with internal memory of size m must have expected bandwidth overhead $\omega(1)$. Any strongly computationally secure online ORAM with internal memory of size m must have expected bandwidth overhead $\Omega(\log(n))$, where $n \geq m^2$ is the length of the sequence of input operations. This result holds even when the adversarial server has no information about boundaries between probes corresponding to different input operations.*

Note that even the $\omega(1)$ lower bound for online ORAMs satisfying weak computational security is an interesting result in the light of the work of Boyle and Naor [2016]. It follows from Boyle and Naor [2016] that any super-constant lower bound for *offline* ORAM would imply super-linear lower bounds on size of sorting circuits – which would constitute a major breakthrough in computational complexity (for additional discussion, see Section 3.5). Our techniques clearly do not provide lower bounds for offline ORAMs. On the other hand, we believe that proving the $\omega(1)$ lower bound in any meaningful weaker model would amount to proving lower bounds for offline ORAM or read-only online ORAM (see Weiss and Wichs [2018]) which would have important implications in computational complexity.

Alternative Definitions of ORAM. Previous works considered various alternative definitions of ORAM. We clarify the ORAM model in which our techniques yield a lower bound in Section 3.2.1 and discuss its relation to other models in Section 3.5. As an additional contribution, we demonstrate an issue with the definition of ORAM appearing in Goldreich and Ostrovsky [1996]. Specifically, we show that the definition can be satisfied by a RAM with constant overhead and no meaningful security. The definition of ORAM in Goldreich and Ostrovsky [1996] differs from the original definition in Goldreich [1987] and Ostrovsky [1990], which do not share the issue we observed in the definition from Goldreich and Ostrovsky [1996]. Given that the work of Goldreich and Ostrovsky [1996] might serve as a primary reference for our community, we explain the issue in Section 3.5 to help preventing the use of the problematic definition in future works.

Persiano and Yeo [2019] adapted the chronogram technique Fredman and Saks [1989] from the literature on data structure lower bounds to prove a lower bound for *differentially private RAMs* (a relaxation of ORAMs in the spirit of differential privacy Dwork et al. [2006] which ensures indistinguishability only for input sequences that differ in a single operation). Similarly to the work of Larsen and Nielsen [2018], the proof in Persiano and Yeo [2019] exploits the fact that the distinguisher knows exactly which server accesses correspond to each input operation. However, as the chronogram technique significantly differs from the information transfer approach, we do not think that our techniques would directly allow to strengthen the Persiano and Yeo [2019] lower bound for differentially private RAMs and prove it in the model with an unstructured access pattern.

3.1.1 Our Techniques

The structure of our proof follows a similar blueprint as the work of Larsen and Nielsen [2018]. However, we must handle new issues introduced by the more general adversarial model. Most significantly, our proof cannot rely on any formatting of the access pattern, whereas Larsen and Nielsen leveraged the fact that the access pattern is split into blocks corresponding to each read/write operation. To handle the lack of structure in the access pattern, we study the properties of the *access graph* induced naturally by the access pattern of an ORAM computation. We identify a particular graph property that can be efficiently tested and that all access graphs of ORAM computation must satisfy with high probability. This property is reminiscent of the Larsen-Nielsen property but it is substantially less structured; that is, it is more generic.

The access graph is defined as follows: the vertices are timestamps of server probes and there is an edge connecting two vertices if and only if they correspond to two subsequent accesses to the same memory cell. We define a graph property called *ℓ-dense k-partition*. Roughly speaking, graphs with ℓ-dense k-partitions are graphs which may be partitioned into k disjoint subgraphs, each subgraph having at least ℓ edges. We show that this property has to be satisfied (with high probability) by access graphs induced by an ORAM for any k and an appropriate ℓ. To leverage this inherent structure of access graph towards a lower bound on bandwidth overhead, we prove that if a graph has $\frac{\ell}{k}$-dense k-partition for some ℓ and K different values of k then the graph must have at least $\Omega(\ell \log(K))$ edges. In Section 3.3, we provide the formal definition of access graph and ℓ-dense k-partitions and prove a lower bound on the expected number of edges for a graph that has many ℓ-dense k-partitions.

In Section 3.4, we prove that access graphs of ORAMs have many dense partitions. Specifically, using a communication-type argument we show that for $\Omega(n)$ values of k, there exist input sequences for which the corresponding graph has $\Omega(\frac{n}{k})$-dense k-partition with high probability. Applying the indistinguishability of sequences of probes made by ORAM, we get one sequence for which its access graph satisfies $\frac{n}{k}$-dense k-partition for $\Omega(n)$ values of k with high probability. Combining the above results from Section 3.4 with the results from Section 3.3, we get that the graph of such a sequence has $\Omega(n \log(n))$ edges, and thus by definition, $\Omega(n \log(n))$ vertices in expectation. This implies that the expected number of probes made by the ORAM on any input sequence of length n is $\Omega(n \log(n))$.

3.2 Preliminaries

In this section, we introduce some basic notation and recall some standard definitions and results. Throughout the rest of the thesis, we let $[n]$ for $n \in \mathbb{N}$ to denote the set $\{1, 2, \ldots, n\}$. A function $\mathsf{negl}(n) \colon \mathbb{N} \to \mathbb{R}$ is *negligible* if it approaches zero faster than any inverse polynomial, formally:

Definition 3.2.1 (Negligible function)**.** *A function $\mu \colon \mathbb{N} \to \mathbb{R}$ is* negligible *if for every $c \in \mathbb{N}$ there is $N_c \in \mathbb{N}$ such that for all $x \geq N_c$ we have*

$$|\mu(x)| \leq 1/x^c.$$

Definition 3.2.2 (Statistical Distance)**.** *For two probability distributions X and Y on a finite set S, we define* statistical distance of X and Y *as*

$$\mathrm{SD}(X,Y) = \frac{1}{2}\sum_{s\in S} |\Pr[X=s] - \Pr[Y=s]| \enspace .$$

We use the following observation, which characterizes statistical distance as the difference of areas under the curve (see Fact 3.1.9 in Vadhan [1999]).

Proposition 3.2.3. *Let X and Y be probability distributions on a finite set S, let $S_X = \{s \in S \colon \Pr[X=s] > \Pr[Y=s]\}$, and define S_Y analogously. Then*

$$\mathrm{SD}(X,Y) = \Pr[X \in S_X] - \Pr[Y \in S_X] = \Pr[Y \in S_Y] - \Pr[X \in S_Y] \enspace .$$

We also use the following data-processing-type inequality.

Proposition 3.2.4. *Let X and Y be probability distributions on a finite set S. Then for any function $f\colon S \to \{0,1\}$, it holds that*

$$|\Pr[f(X)=1] - \Pr[f(Y)=1]| \le \mathrm{SD}(X,Y) \,.$$

A *probability ensemble* is a family of distributions of random variables. We say that two probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable if no polynomial distinguishing algorithm D is able to tell whether it is running on $1^n, X_n$ or on $1^n, Y_n$, formally:

Definition 3.2.5 (Computational indistinguishability)**.** *Two probability ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, are* computationally indistinguishable *if for every polynomial-time algorithm D there exists a negligible function $\mathsf{negl}(\cdot)$ such that*

$$|\Pr[D(1^n, X_n)=1] - \Pr[D(1^n, Y_n)=1]| \le \mathsf{negl}(n) \enspace .$$

The security parameters 1^n are redundant in the case of strong computational security (see Definition 3.2.7) and thus not written down explicitly in the rest of the chapter as we consider only ensembles with $|X_n| \ge n$. In the case of weak computational security (see Definition 3.2.7) we explicitly write them in the ensambles.

3.2.1 Online ORAM

In this section, we present the formal definition for online oblivious RAM model (ORAM) we consider in our work – we build on the oblivious cell-probe model of Larsen and Nielsen [2018].

Definition 3.2.6 (Array Maintenance Problem Larsen and Nielsen [2018])**.** *The* Array Maintenance problem with parameters (ℓ, w) *is to maintain an array B of ℓ w-bit entries under the following two operations:*

Write operation (W, a, d)**:** *Set the content of $B[a]$ to d, where the address a is from $[\ell]$ and data $d \in \{0,1\}^w$.*

Read operation (R, a, d)**:** *Return the content of $B[a]$, where the address $a \in [\ell]$ and data $d \in \{0,1\}^w$ (note that d is ignored).*

We say that a machine $\mathcal{M}$ implements the Array Maintenance problem with parameters (ℓ, w) and probability p, *if for every* $n \in \mathbb{N}$ *and every input sequence of* n *operations*

$$y = (o_1, a_1, d_1), \ldots, (o_n, a_n, d_n), \text{ where each } o_i \in \{R, W\}, a_i \in [\ell], d_i \in \{0,1\}^w,$$

and for every read operation in the sequence y, $\mathcal{M}$ *returns the correct answer (that is the data* d *which were last written in the address location* a*) with probability at least* p.

Definition 3.2.7 (Online Oblivious RAM)**.** *For* $m, w \in \mathbb{N}$, *let RAM**(m, w) *denote a probabilistic random access machine* $\mathcal{M}$ *with* m *cells of internal memory, each of size* w *bits, which has access to a data structure, called* server*, implementing the Array Maintenance problem with parameters* $(2^w, w)$ *and probability 1. In other words, in each step of computation* $\mathcal{M}$ *may* probe *the server on a triple* $(o, a, d) \in \{R, W\} \times [2^w] \times \{0,1\}^w$ *and on every input* (R, a, d) *the server returns to* $\mathcal{M}$ *the data last written in* $B[a]$*. We say that RAM** probes *the server whenever it makes an Array Maintenance operation to the server.*

Let m, M, w *be any natural numbers such that* $M \leq 2^w$*. An* online Oblivious RAM $\mathcal{M}$ with address range M, cell size w bits and m cells of internal memory *is a RAM**(m, w) *satisfying online access sequence, correctness, and statistical (resp. computational) security as defined below.*

Online Access Sequence: *For any input sequence* $y = y_1, \ldots, y_n$ *the RAM* machine* $\mathcal{M}$ *gets* y_i *one by one, where each* $y_i \in \{R, W\} \times [M] \times \{0,1\}^w$*. Upon the receipt of each operation* y_i*, the machine* $\mathcal{M}$ *generates a possibly empty sequence of* server probes $(o_1, a_1, d_1), \ldots, (o_{\ell_i}, a_{\ell_i}, d_{\ell_i})$*, where each* $(o_i, a_i, d_i) \in \{R, W\} \times [2^w] \times \{0,1\}^w$*, and updates its internal memory state in order to correctly implement the request* y_i*. We define the* access sequence corresponding to y_i *as* $A(\mathcal{M}, y_i) = a_1, a_2, \ldots, a_{\ell_i}$*. For the input sequence* y*, the* access sequence $A(\mathcal{M}, y)$ *is defined as*

$$A(\mathcal{M}, y) = A(\mathcal{M}, y_1), A(\mathcal{M}, y_2), A(\mathcal{M}, y_3), \ldots, A(\mathcal{M}, y_n).$$

Note that the definition of the machine $\mathcal{M}$ *is* online*, and thus for each input sequence* $y = y_1, \ldots, y_n$ *and each* $i \in [n-1]$*, the access sequence* $A(\mathcal{M}, y_i)$ *does not depend on* $y_{i+1}, \ldots, y_n$*.*

Correctness: $\mathcal{M}$ *implements the Array Maintenance problem with parameters* (M, w) *with probability at least* $1 - p_{\text{fail}}$ *(for some fixed number* $p_{\text{fail}} \in [0, 1)$*).*

Statistical Security: *For any two input sequences* y, y' *of the same length, the statistical distance of the distributions of access sequences* $A(\mathcal{M}, y)$ *and* $A(\mathcal{M}, y')$ *is at most* $\frac{1}{4}$*.*

Computational Security: *We consider infinite families of ORAM where we allow* m, M, w *to be functions of the length* n *of the input sequence. We distinguish between the following two notions:*

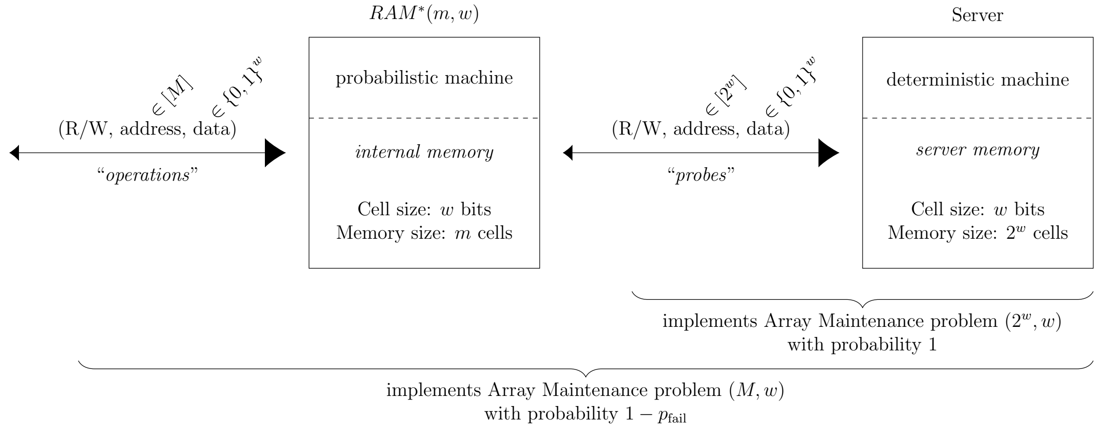

Figure 3.1: Schema of online ORAM from Definition 3.2.7.

Weak Computational Security: *For any infinite families of input sequences $\{y_n\}_{n\in\mathbb{N}}$ and $\{y'_n\}_{n\in\mathbb{N}}$ such that $|y_n| = |y'_n| \geq n$ for all $n \in \mathbb{N}$, the probability ensembles (together with 1^n as security parameters) $\{(1^n, A(\mathcal{M}, y_n))\}_{n\in\mathbb{N}}$ and $\{(1^n, A(\mathcal{M}, y'_n))\}_{n\in\mathbb{N}}$ are computationally indistinguishable.*

Strong Computational Security: *For any infinite families of input sequences $\{y_n\}_{n\in\mathbb{N}}$ and $\{y'_n\}_{n\in\mathbb{N}}$ such that $|y_n| = |y'_n| \geq n$ for all $n \in \mathbb{N}$, the probability ensembles*

$$\{(y_n, y'_n, A(\mathcal{M}, y_n))\}_{n\in\mathbb{N}}$$

and

$$\{(y_n, y'_n, A(\mathcal{M}, y'_n))\}_{n\in\mathbb{N}}$$

are computationally indistinguishable.

The parameters of our ORAM model from Definition 3.2.7 are depicted in Figure 3.1. We use different sizes of arrows on server and RAM side to denote the asymmetry of the communication (the RAM sends type of operation, address, and data and the server returns requested data in case of a read operation and a dummy value in case of a write operation). Note that the input sequence y of ORAM consists of a sequence of all operations, whereas the access sequence $A(\mathcal{M}, y)$ consists of a sequence of addresses of all probes.

Arguably, a user of an ORAM might want the stronger notion of computational security whereas the weaker notion is closer to the past considerations. Note that in the case of weak computational security, the adversarial distinguisher does not have access to the input sequences. Thus, it is restricted to contain only constant amount of information about the whole families of input sequences $\{y_n\}_n$ and $\{y'_n\}_n$. In contrast, in the case of strong computational security, the adversarial distinguisher is given also the input sequences. Thus, it is able to compute any polynomial time computable information about the input sequences. This distinction is crucial for our results, as we are able to prove only an $\omega(1)$ lower bound for the weak security as opposed to the $\Omega(\log(n))$ lower bound for strong security (see Theorem 3.4.10 and Theorem 3.4.9). Nevertheless, we believe that

the known constructions of ORAM satisfy the notion of strong computational security.

For ease of exposition, in the rest of the chapter we assume perfect correctness of the ORAM (i.e., $p_{\text{fail}} = 0$). However, our lower bounds can be extended also to ORAMs with imperfect correctness (see Remark 3.4.3). Finally, our lower bounds hold also for *semi-offline* ORAMs where the ORAM machine $\mathcal{M}$ receives the type and address of each operation in advance and it has to process in online manner only the data to be written during each write operation (see Remark 3.4.4).

3.3 Dense Graphs

In this section, we define an efficiently testable property of graphs that we show to be satisfied by graphs induced by the access pattern of any statistically secure ORAM. This property implies that the expected overhead of such ORAM must be logarithmic.

We say that a directed graph $G = (V, E)$ is *ordered* if V is a subset of integers and for each edge $(u, v) \in E$: $u < v$. For a graph $G = (V, E)$ and $S, T \subseteq V$, we let $E(S, T) \subseteq E$ be the set of edges that start in S and end in T, and for integers $a < m < b \in V$ we let $E(a, m, b) = E(\{a, a+1, \ldots, m-1\}, \{m, m+1, \ldots, b-1\})$.

Definition 3.3.1. *A* k-partition *of an ordered graph* $G = (V = \{0, 1, 2, \ldots, N-1\}, E)$ *is a sequence* $0 = b_0 < m_0 < b_1 < m_1 < \cdots < b_k = N$*. We say that the* k*-partition is* ℓ*-dense if for each* $i \in \{0, \ldots, k-1\}$*,* $E(b_i, m_i, b_{i+1})$ *is of size at least* ℓ*.*

There is a simple greedy algorithm running in time $\mathcal{O}(|V|^2 \cdot |E|)$ which tests for given integers k, ℓ whether a given ordered graph $G = (V, E)$ has an ℓ-dense k-partition. (The algorithm looks for the k parts one by one greedily from left to right.)

Lemma 3.3.2. *Let* $K \subseteq \mathbb{N}$ *be a subset of powers of 4. Let* $\ell \in \mathbb{N}$ *be given. Let* $G = (\{0, \ldots, N-1\}, E)$ *be an ordered graph which for each* $k \in K$ *has an* (ℓ/k)*-dense* k*-partition. Then* G *has at least* $\frac{\ell}{2} \cdot |K|$ *edges.*

Proof. We use the following claim to bound the number of edges.

Claim 3.3.3. *Let* $k > k' > 0$ *be integers. Let* $0 = b_0 < m_0 < b_1 < m_1 < \cdots < b_k = N$ *be a* k*-partition of* G*, and* $0 = b'_0 < m'_0 < b'_1 < m'_1 < \cdots < b'_{k'} = N$ *be a* k'*-partition of* G*. Then for at least* $k - k'$ *distinct* $i \in \{0, \ldots, k-1\}$

$$E(b_i, m_i, b_{i+1}) \cap \bigcup_{j \in \{0, \ldots, k'-1\}} E(b'_j, m'_j, b'_{j+1}) = \emptyset. \tag{3.3.1}$$

Proof. For any $j \in \{0, \ldots, k'-1\}$ and $(u, v) \in E(b'_j, m'_j, b'_{j+1})$, if $(u, v) \in E(b_i, m_i, b_{i+1})$ for some i then $b_i < m'_j < b_{i+1}$ (as $b_i \leq u < m'_j \leq v \leq b_{i+1}$.) Thus, i is uniquely determined by j. Hence, $E(b_i, m_i, b_{i+1})$ may intersect $\bigcup_{j \in \{0, \ldots, k'-1\}} E(b'_j, m'_j, b'_{j+1})$ only if $b_i \leq m'_j < b_{i+1}$, for some $j \in \{0, \ldots, k'-1\}$. Thus, such an intersection occurs only for at most k' different i. The claim follows. □

Now we are ready to prove Lemma 3.3.2. For each $k \in K$, pick an (ℓ/k)-dense k-partition $0 = b_0 < m_0 < b_1 < m_1 < \cdots < b_k = N$ of G and define the set of edges E_k:

$$E_k = \bigcup_{i \in \{0,\ldots,k-1\}} E(b_i, m_i, b_{i+1}).$$

For each $k \in K$, we lower-bound $\left| E_k \setminus \bigcup_{k' \in K, k' < k} E_{k'} \right|$ by $\ell/2$. Since K contains powers of 4, $\sum_{k' \in K, k' < k} k' \le k/2$. By the above claim, for at least $k - \sum_{k' \in K, k' < k} k' \ge k/2$ different $i \in \{0, \ldots, k-1\}$, $E(b_i, m_i, b_{i+1}) \cap \bigcup_{k' \in K, k' < k} E_{k'} = \emptyset$. By density, $|E(b_i, m_i, b_{i+1})| \ge \ell/k$, so $\left| E_k \setminus \bigcup_{k' \in K, k' < k} E_{k'} \right| \ge \frac{\ell}{k} \cdot \frac{k}{2} = \ell/2$. Hence, $|\bigcup_{k \in K} E_k| = \sum_{k \in K} \left| E_k \setminus \bigcup_{k' \in K, k' < k} E_{k'} \right| \ge |K| \cdot \frac{\ell}{2}$. □

In the following corollary, we show that the property of having many dense partitions with some probability implies proportionally many edges. (Note that the $\lfloor \log_4(t) \rfloor - \lceil \log_4(s) \rceil$ term corresponds exactly to the number of powers of four between s and t.)

Corollary 3.3.4. *Let ℓ, s, t be natural numbers, where $s \le t$. Let $p \in [0, 1]$ be a real number. Let G be an ordered graph picked at random from a distribution such that for each integer k, $s \le k \le t$, the randomly chosen ordered graph G has (ℓ/k)-dense k-partition with probability at least p. Then the expected number of edges in G is at least $\frac{p\ell}{2} \cdot (\lfloor \log_4(t) \rfloor - \lceil \log_4(s) \rceil)$.*

Proof. Let K be the set of integers such that $k \in K$ if and only if k is a power of 4 and G has an (ℓ/k)-dense k-partition. K is a random variable. The expected size of K is at least $p(\lfloor \log_4(t) \rfloor - \lceil \log_4(s) \rceil)$. By Lemma 3.3.2, the expected number of edges in G is at least $\frac{\ell}{2} \cdot p \cdot (\lfloor \log_4(t) \rfloor - \lceil \log_4(s) \rceil)$. □

3.4 ORAM Lower Bound

In this section, we fix integers $n, m, M, w \ge 1$ such that $m \le \sqrt{n}$, $n \le M \le 2^w$, and an ORAM $\mathcal{M}$ with address range M, cell size w and m cells of internal memory (see Definition 3.2.7). We argue that any statistically secure ORAM $\mathcal{M}$ must make $\Omega(n \log(n))$ server probes in expectation in order to implement a sequence of n input operations. We prove the same result for ORAM $\mathcal{M}$ satisfying Strong Computational Security. We also show that any ORAM $\mathcal{M}$ satisfying Weak Computational Security must make $\omega(n)$ server probes in expectation on any input sequence of length n.

Definition 3.4.1. *Let $A(\mathcal{M}, y) = a_0, \ldots, a_{N-1}$ be an access sequence of $\mathcal{M}$ for some input sequence y. We define a directed graph $G(A(\mathcal{M}, y)) = (V, E)$ called* access graph *as follows: $V = \{0, \ldots, N-1\}$ and $(i, j) \in E$ iff $i < j$, $a_i = a_j$, and for each $k \in \{i+1, \ldots, j-1\}$: $a_k \ne a_i$.*

Notice that every vertex of an access graph has outdegree as well as indegree at most one.

In the following, we consider input sequences of even length $n \in \mathbb{N}$. First, we define a sequence of alternating writes and reads at address $a = 1$ with data

$d = 0^w$ as $Y_{n,0} = [(W, 1, 0^w), (R, 1, 0^w)]^{n/2}$. Second, for each $k \in \left\{1, 2, \ldots, \frac{n}{2}\right\}$, let $\ell = \left\lfloor \frac{n}{2k} \right\rfloor$, we define a distribution $Y_{n,k}$ of input sequences as

$$
\begin{aligned}
Y_{n,k} =& (W, 1, b_{1,1}), (W, 2, b_{1,2}), \ldots, (W, \ell, b_{1,\ell}), (R, 1, 0^w), (R, 2, 0^w), \ldots, (R, \ell, 0^w), \\
& (W, 1, b_{2,1}), (W, 2, b_{2,2}), \ldots, (W, \ell, b_{2,\ell}), (R, 1, 0^w), (R, 2, 0^w), \ldots, (R, \ell, 0^w), \\
& \ldots, \\
& (W, 1, b_{k,1}), (W, 2, b_{k,2}), \ldots, (W, \ell, b_{k,\ell}), (R, 1, 0^w), (R, 2, 0^w), \ldots, (R, \ell, 0^w), \\
& (W, 1, 0^w), (R, 1, 0^w), (W, 1, 0^w), \ldots, (R, 1, 0^w) \ ,
\end{aligned}
$$

where each $b_{i,j} \in \{0, 1\}^w$ is an independently uniformly chosen bit string. We define the i-th block of writes

$$W_i = (W, 1, b_{i,1}), (W, 2, b_{i,2}), \ldots, (W, \ell, b_{i,\ell})$$

and the i-th block of reads R_i to be the sequence of operations

$$R_i = (R, 1, 0^w), (R, 2, 0^w), \ldots, (R, \ell, 0^w)$$

following right after W_i. Note that after the k-th block of reads the sequence is padded to length n by a sequence of alternating writes and reads. For an ORAM $\mathcal{M}$, we use the notation $G_{n,k} = G(A(\mathcal{M}, Y_{n,k}))$ and $G_{n,0} = G(A(\mathcal{M}, Y_{n,0}))$ when $\mathcal{M}$ is clear from the context.

The following lemma uses only correctness of ORAM and does not depend on its security. The proof of the lemma uses the information transfer technique similarly to Lemma 2 in Larsen and Nielsen [2018].

Lemma 3.4.2. *Let $n, m, M, w, \mathcal{M}$ be as in the beginning of this section, moreover suppose $n \geq 10$ is an even integer. Let $k \geq 1$ be an integer such that $k \leq \frac{n}{10(m+2\log(n)+11)}$. Let $A(\mathcal{M}, Y_{n,k})$ be the access sequence of $\mathcal{M}$ and $G_{n,k}$ be the corresponding access graph. ($G_{n,k}$ is a random variable that depends on $Y_{n,k}$ and the internal randomness of $\mathcal{M}$.) With probability at least $1 - \frac{1}{n}$, $G_{n,k}$ has $(n/5k)$-dense k-partition.*

Proof. By our assumption from the beginning of this section, $n \leq M$, and thus for any $k \in \{1, 2, \ldots, \frac{n}{2}\}$ all sequences $Y_{n,k}$ have all addresses in the correct range. Fix any k satisfying the assumptions of this lemma and set $\ell = \left\lfloor \frac{n}{2k} \right\rfloor$. As defined before let W_i and R_i be the i-th block of writes and reads in $Y_{n,k}$, respectively. Let U_i be the vertices of $G_{n,k}$ corresponding to W_i, and V_i be the vertices corresponding to R_i. It suffices to prove that for each $i \in \{1, \ldots, k\}$, the probability that there are fewer than $n/5k$ edges between U_i and V_i is less than $1/n^2$. If this holds then by the union bound the lemma follows.

For contradiction, assume there exists $i \in \{1, \ldots, k\}$ such that the probability that there are fewer than $n/5k$ edges between U_i and V_i is at least $1/n^2$. Here, the randomness is taken over the choice of an input sequence $y \leftarrow Y_{n,k}$ and the internal randomness of $\mathcal{M}$. Fix such an i. Fix all the randomness except for the choice of $b_{i,1}, \ldots, b_{i,\ell}$ in $Y_{n,k}$ so that $G_{n,k}$ obtained from this restricted distribution has fewer than $n/5k$ edges between U_i and V_i with probability $\geq 1/n^2$ over the choice of $b_{i,1}, \ldots, b_{i,\ell}$. (This is possible by an averaging argument.) Let $B \subseteq \{0, 1\}^{w \times \ell}$

be the set of choices for $b_{i,1}, \ldots, b_{i,\ell}$ which give fewer than $n/5k$ edges between U_i and V_i in $G_{n,k}$. Clearly, $|B| \geq 2^{w\ell}/n^2$.

We use $\mathcal{M}$ to construct a deterministic protocol that transmits any string from B from Alice to Bob, two communicating parties, using at most $\log(|B|) - 10$ bits. That gives a contradiction as such an efficient transmission violates the pigeon-hole principle.

On input $b \in B$ to Alice, Alice sends a single message to Bob who can determine b from the message. They proceed as follows. Both Alice and Bob simulate $\mathcal{M}$ on $Y_{n,k}$ up until reaching W_i. All the randomness used before the i-th block of writes W_i is fixed and known both to Alice and Bob. Then Alice continues with the simulation of $\mathcal{M}$ on W_i with data $b_{i,1}, b_{i,2}, \ldots, b_{i,\ell}$ set to b. Once she finishes it, she sends the content of the internal memory of $\mathcal{M}$ to Bob using wm bits. Then Alice continues with the simulation of $\mathcal{M}$ on R_i and whenever $\mathcal{M}$ makes a server probe to read from a location that was written last time during the simulation of W_i, Alice sends over the address and the content of that cell to Bob. Overall, Alice sends to Bob at most $mw + 2wn/5k$ bits of communication that can be concatenated into a single message of this size.

On receiving side, Bob uses the internal state of $\mathcal{M}$ communicated by Alice to continue with the computation on R_i, while he uses the state of the server he obtained initially before reaching W_i. He simulates all server probes by himself, except for read operations that match the list sent by Alice, where he initially uses the content provided by Alice. Clearly, Bob can determine b from the simulation.

As $k \leq \frac{n}{10(m+2\log(n)+11)}$ hence the number of communicated bits is at most

$$\begin{aligned}
mw + 2wn/5k &\leq \frac{5mw\frac{n}{10(m+2\log(n)+11)} + 2wn}{5k} \\
&= \frac{mwn + 4wn(m + 2\log(n) + 11)}{10k(m + 2\log(n) + 11)} \\
&= \frac{wn}{2k} - w(2\log(n) + 11)\frac{n}{10k(m + 2\log(n) + 11)} \\
&\leq w\left(\frac{n}{2k} - (2\log(n) + 11)\right) \\
&\leq w\left(\ell - 2\log(n) - 10\right) \\
&\leq \log(|B|) - (2w - 2)\log(n) - 10w \qquad \text{(by } |B| \geq 2^{w\ell}/n^2\text{)}
\end{aligned}$$

which is a contradiction. □

Remark 3.4.3. *Using good error-correcting codes (see for instance MacWilliams and Sloane [1977]), this lemma could be generalized to the case when $\mathcal{M}$ implements Array Maintenance problem with probability $1-p_{\text{fail}} < 1$, i.e., $\mathcal{M}$ is allowed to return a wrong value for each of its input read operations with a small constant probability p_{fail}. The graph $G_{n,k}$ would still have $(\epsilon n/k)$-dense k-partition with $1 - 1/n$ probability for some $\epsilon > 0$ which depends only on the allowed failure probability p_{fail}.*

Remark 3.4.4. *Note that the randomness of input sequence $Y_{n,k}$ is used only for the data to be written. Moreover, the proof relies only on incompressibility of a random string stored during the write block and it does not rely on the addresses used to store this data. Thus, the same proof goes through even for* semi-offline

ORAMs, *i.e., if we allow the ORAM to know the type and address of each input operation in y in advance. On the other hand, as our proof uses interleaved sequences of write blocks and read blocks, it is unlikely that it would be possible to extend it to the* read-only online ORAM *model of Weiss and Wichs [2018].*

Note that using an averaging argument we can assume that the probability in Lemma 3.4.2 is only over the randomness of $\mathcal{M}$. Thus we get the following corollary proving for every k the existence of a single input sequence whose corresponding access graph has $\frac{n}{5k}$-dense k-partition with high probability.

Corollary 3.4.5. *For any even integer $n \geq 10$ and an integer $k \geq 1$ such that $k \leq \frac{n}{10(m+2\log(n)+11)}$ there is an input sequence $y_{n,k}$ of length n such that $G(A(\mathcal{M}, y_{n,k}))$ has a $(n/5k)$-dense k-partition with probability at least $1 - \frac{1}{n}$.*

We show that by statistical security of $\mathcal{M}$, this property holds for a single input sequence and many different values of k.

Lemma 3.4.6. *Let $n, m, M, w, \mathcal{M}$ be as in the beginning of this section, and assume n is even and $n \geq 10$. Let y be an input sequence to $\mathcal{M}$ of length n. If $\mathcal{M}$ is a statistically secure online ORAM then for every $k \in \left\{1, 2, \ldots, \left\lfloor \frac{n}{10(m+2\log(n)+11)} \right\rfloor\right\}$*

$$\Pr\left[G(A(\mathcal{M}, y)) \textit{ has an } (n/5k)\textit{-dense } k\textit{-partition}\right] \geq \frac{3}{5}.$$

Proof. For contradiction, suppose that for some k the probability is less than 3/5. From the statistical security of $\mathcal{M}$ we know that the statistical distance $\mathrm{SD}\left(A(\mathcal{M}, y), A(\mathcal{M}, y_{n,k})\right) \leq \frac{1}{4}$ where $y_{n,k}$ is given by Corollary 3.4.5. By Corollary 3.4.5 the sequence $y_{n,k}$ gives us a graph $G(A(\mathcal{M}, y_{n,k}))$ which has an $(n/5k)$-dense k-partition with probability at least $1 - 1/n \geq 9/10$. Define a function $f_{\ell,k}$ on ordered graphs that is an indicator of having an ℓ-dense k-partition. Applying Proposition 3.2.4 with $X \leftarrow G(A(\mathcal{M}, y))$, $Y \leftarrow G(A(\mathcal{M}, y_{n,k}))$, and $f = f_{n/5k,k}$, we can conclude that $G(A(\mathcal{M}, y))$ has an $(n/5k)$-dense k-partition with probability at least $3/4 - 1/10 \geq 3/5$. □

We are ready to prove our main theorem for statistically secure ORAM.

Theorem 3.4.7. *There are constants $c_0, c_1 > 0$ such that for any integers $m, w \geq 1$ and $M \geq n \geq c_0$ where $m \leq \sqrt{n}$ and $M \leq 2^w$, any statistically secure online ORAM $\mathcal{M}$ with address range M, cell size w bits and m cells of internal memory must perform at least $c_1 n \log(n)$ server probes in expectation (the expectation is over the randomness of $\mathcal{M}$) on any input sequence of length n.*

Proof. Fix an ORAM machine $\mathcal{M}$. Consider any input sequence y to $\mathcal{M}$ of length n. By Lemma 3.4.6 for every k, such that $1 \leq k \leq \left\lfloor \frac{n}{10(m+2\log(n)+11)} \right\rfloor$, we get that

$$\Pr\left[G(A(\mathcal{M}, y)) \text{ has an } (n/5k)\text{-dense } k\text{-partition}\right] \geq \frac{3}{5}.$$

Applying Corollary 3.3.4 with $s = 1$, $t = \left\lfloor \frac{n}{10(m+2\log(n)+11)} \right\rfloor$, $\ell = \left\lfloor \frac{n}{5} \right\rfloor$, and $p = 3/5$, we can lower bound the expected number of edges in $G(A(\mathcal{M}, y))$ by

$$\frac{3n}{50} \left\lfloor \log_4 \left\lfloor \frac{n}{10(m + 2\log(n) + 11)} \right\rfloor \right\rfloor.$$

For $n \geq 1000$, $\left\lfloor \frac{n}{10(m+2\log(n)+11)} \right\rfloor \geq \frac{\sqrt{n}}{40}$. Hence, the expected number of edges in $G(A(\mathcal{M}, y))$ is at least $\frac{3}{100} \cdot n \log\left(\frac{\sqrt{n}}{40}\right) \geq \frac{1}{100} \cdot n \log(n)$, provided c_0 is large enough. Since the indegree of each vertex of an access graph is at most one, the expected number of vertices in $G(A(\mathcal{M}, y))$, which is the same as the expected number of probes in $A(\mathcal{M}, y)$, is at least $\frac{1}{100} \cdot n \log(n)$. □

Next, we prove $\Omega(\log(n))$ lower bound for ORAMs satisfying strong computational security from Definition 3.2.7.

Lemma 3.4.8. *Let $m, M, w \colon \mathbb{N} \to \mathbb{N}$ be non-decreasing functions such that for all n large enough: $m(n) \leq \sqrt{n}$ and $n \leq M(n) \leq 2^{w(n)}$. Let $\{\mathcal{M}_n\}_{n \in \mathbb{N}}$ be a sequence of online ORAMs with address range $M(n)$, cell size $w(n)$ bits and $m(n)$ cells of internal memory which satisfy strong computational security. Let $\{y_n\}_{n \in \mathbb{N}}$ be an infinite family of input sequences where $|y_n| = n$, for each $n \in \mathbb{N}$.*

Then there exists n_0 such that for every $n \geq n_0$ and for every

$$k \in \left\{1, 2, \ldots, \left\lfloor \frac{n}{10(m(n) + 2\log(n) + 11)} \right\rfloor \right\}$$

it holds that

$$\Pr\left[G(A(\mathcal{M}_n, y_n)) \text{ has an } (n/5k)\text{-dense } k\text{-partition}\right] \geq \frac{3}{5}.$$

Proof. For contradiction, assume there are infinitely many pairs of integers (n, k), s.t. $k \leq \left\lfloor \frac{n}{10(m(n)+2\log(n)+11)} \right\rfloor$ and that the probability that y_n has an $(n/5k)$-dense k-partition is less than $3/5$.

Let $\mathcal{D}$ be an algorithm which given two input sequences y and y' of length n and an access sequence $A(\mathcal{M}_n, z)$, where $z \in \{y, y'\}$, does the following:

1. Compute n.
2. Compute k' to be the number of blocks of consecutive reads of length $\lfloor n/k' \rfloor$ in the input sequence y'.
3. If $A(\mathcal{M}_n, z)$ does not have $(n/5k')$-dense k'-partition $\mathcal{D}$ returns "1" (i.e. D guesses that $z = y$).
4. Otherwise $\mathcal{D}$ returns "1" with probability $1/2$ and "2" with probability $1/2$ (i.e. D guesses at random).

There is a polynomial time greedy algorithm determining whether the graph $G(A(\mathcal{M}_n, z))$ contains an ℓ-dense k-partition. Thus algorithm $\mathcal{D}$ runs in time polynomial in the length of the access sequence $A(\mathcal{M}_n, z)$.

Let $y_{n,k}$ be a sequence from Corollary 3.4.5. So, $G(A(\mathcal{M}_n, y_{n,k}))$ has an $(n/5k)$-dense k-partition with probability at least $1 - 1/n \geq 9/10$. Observe that if $y = y_n$ and $y' = y_{n,k}$ then:

$$\begin{aligned} &|\Pr[\mathcal{D}(y_n, y_{n,k}, A(\mathcal{M}_n, y_n)) = 1] - \Pr[\mathcal{D}(y_n, y_{n,k}, A(\mathcal{M}_n, y_{n,k})) = 1]| \\ &\qquad \geq \left(\frac{2}{5} + \frac{3}{5} \cdot \frac{1}{2}\right) - \left(\frac{1}{10} + \frac{9}{10} \cdot \frac{1}{2}\right) = \frac{3}{20}. \end{aligned}$$

By the assumption $\mathcal{D}$ returns “1” in step 3 on $A(\mathcal{M}_n, y_n)$ with probability at least 2/5. By Corollary 3.4.5 $\mathcal{D}$ answers “1” on $A(\mathcal{M}_n, y_{n,k})$ with probability at most 1/10.

This contradicts the strong computational security of $\mathcal{M}_n$ as $\mathcal{D}$ should not distinguish between y and y' with non-negligible probability. □

Theorem 3.4.9. *Let $m, M, w\colon \mathbb{N} \to \mathbb{N}$ be non-decreasing functions such that for all n large enough: $m(n) \leq \sqrt{n}$ and $n \leq M(n) \leq 2^{w(n)}$. Let $\{\mathcal{M}_n\}_{n\in\mathbb{N}}$ be a sequence of online ORAMs with address range $M(n)$, cell size $w(n)$ bits and $m(n)$ cells of internal memory which satisfy strong computational security. Let $\{y_n\}_{n\in\mathbb{N}}$ be an infinite family of input sequences where $|y_n| = n$, for each $n \in \mathbb{N}$.*

There are constants $c_0, c_1 > 0$, such that for any $n \geq c_0$, $\mathcal{M}_n$ must perform in expectation at least $c_1 n \log(n)$ server probes on the input sequence y_n.

Proof. The proof of this theorem is identical to the proof of Theorem 3.4.7 but we use Lemma 3.4.8 instead of Lemma 3.4.6. Note that the different order of quantifiers is caused by different order of quantifiers in Lemma 3.4.6 and in Lemma 3.4.8. □

In the rest of this section, we prove an $\omega(1)$ lower bound for ORAMs satisfying weak computational security from Definition 3.2.7. Note that in the case of weak computational security it is unclear which k should the adversary use to distinguish y and y'. Thus, we cannot directly conclude that y has $\frac{n}{5k}$-dense k-partition for every n and $k \leq \left\lfloor \frac{n}{10(m(n)+2\log(n)+11)} \right\rfloor$. On the other hand, for every k there could be only finitely many values n such that there is an input sequence of length n which has no $\frac{n}{5k}$-dense k-partition. This fact allows us to prove the $\omega(1)$ lower bound for weak computational security.

Theorem 3.4.10. *Let $m, M, w\colon \mathbb{N} \to \mathbb{N}$ be non-decreasing functions such that for all n large enough: $m(n) \leq \sqrt{n}$ and $n \leq M(n) \leq 2^{w(n)}$. Let $\{\mathcal{M}_n\}_{n\in\mathbb{N}}$ be a sequence of online ORAMs with address range $M(n)$, cell size $w(n)$ bits and $m(n)$ cells of internal memory which satisfy weak computational security. Let $\{y_n\}_{n\in\mathbb{N}}$ be a sequence of input sequences where $|y_n| = n$, for each $n \in \mathbb{N}$.*

For any constant $c_1 > 0$ there is a constant $c_0 > 0$, such that for any $n \geq c_0$, $\mathcal{M}_n$ must perform in expectation at least $c_1 n$ server probes on the input sequence y_n.

In particular there is no computationally secure online ORAM with constant bandwidth overhead.

Proof. For each $n \in \mathbb{N}$, define $k(n)$ to be the smallest k such that

$$\Pr[G(A(\mathcal{M}_n, y_n)) \text{ has } (n/5k)\text{-dense } k\text{-partition}] < 1/2.$$

Using Corollary 3.3.4 we get for each n large enough that the expected number of edges in $G(A(\mathcal{M}_n, y_n))$ is at least $c \cdot n \log(k(n))$, for some absolute constant $c > 0$. It suffices to show that $k(n) \to \infty$ as $n \to \infty$. There cannot exist a constant k such that Y_n has $(n/5k)$-dense k-partition with probability less than $\frac{1}{2}$ for infinitely many n. Otherwise $\{y_n\}_n$ would be computationally distinguishable from $\{Y_{n,k}\}_n$ (by the greedy algorithm which has k hard-wired). So, $k(n) \to \infty$ as $n \to \infty$. □

3.5 Alternative Definitions for Oblivious RAM

In this section, we recall some alternative definitions for ORAM which appeared in the literature and explain the relation of our lower bound to those models.

The definition of Larsen and Nielsen. Larsen and Nielsen (see Definition 4 in Larsen and Nielsen [2018]) required that for any two input sequences of equal length, the corresponding distributions of access sequences cannot be distinguished with probability greater than 1/4 by any algorithm running in polynomial time in the sum of the following terms: the length of the input sequence, logarithm of the number of memory cells (i.e., $\log(n)$), and the size of a memory cell (i.e., $\log(n)$ for the most natural parameters). We show that their definition implies statistical closeness as considered in our work (see the statistical security property in Definition 3.2.7). Therefore, any lower bound on the bandwidth overhead of ORAM satisfying our definition implies a matching lower bound w.r.t. the definition of Larsen and Nielsen [2018].

To this end, let us show that if two distributions of access sequences are not statistically close, then they are distinguishable in the sense of Larsen and Nielsen. Assume there exist two input sequences y and y' of equal lengths, for which the access sequences $A(\mathcal{M}, y)$ and $A(\mathcal{M}, y')$ have statistical distance greater than 1/4. We define a distinguisher algorithm D that on access sequence x outputs 1 whenever $\Pr[A(\mathcal{M}, y) = x] > \Pr[A(\mathcal{M}, y') = x]$, outputs 0 whenever $\Pr[A(\mathcal{M}, y) = x] < \Pr[A(\mathcal{M}, y') = x]$, and outputs a uniformly random bit whenever $\Pr[A(\mathcal{M}, y) = x] = \Pr[A(\mathcal{M}, y') = x]$. It follows from definition of D, basic properties of statistical distance (see Proposition 3.2.3), and our assumption about the statistical distance of $A(\mathcal{M}, y)$ and $A(\mathcal{M}, y')$ that

$$|\Pr[D(A(\mathcal{M}, y)) = 1] - \Pr[D(A(\mathcal{M}, y')) = 1]| = \mathrm{SD}\left(A(\mathcal{M}, y), A(\mathcal{M}, y')\right) > \frac{1}{4} .$$

Note that D can be specific for the pair of the two input sequences y and y' and it can have all the significant information about the distributions $A(\mathcal{M}, y)$ and $A(\mathcal{M}, y')$ hardwired. For example, it is sufficient to store a string describing for each access sequence x whether it is more, less, or equally likely under $A(\mathcal{M}, y)$ or $A(\mathcal{M}, y')$. Even though such string is of exponential size w.r.t. the length of the access pattern, D needs to simply access the position corresponding to the observed access pattern to output its decision as described above. Thus, D can run in linear time in the length of the access sequence (which is polynomial in the length of the input sequence) and distinguishes the two access sequences with probability greater than 1/4.

The definition of Goldreich and Ostrovsky. Unlike the original definition of ORAM from Goldreich [1987] and Ostrovsky [1990], the definition of ORAM presented in Goldreich and Ostrovsky [1996] contains an alternative security requirement. However, the alternative definition suffers from an issue which is not present in the original definition and which, to the best of our knowledge, was not pointed out in the literature. In particular, the definition in Goldreich and Ostrovsky [1996] can be satisfied by a dummy ORAM construction with only a constant overhead and without achieving any indistinguishability of the access

sequences. Given that Goldreich and Ostrovsky [1996] might serve as a primary reference for our community, we explain the issue in the following paragraph to help preventing the use of the problematic definition in future works.

Recall the definition of ORAM with perfect security from Goldreich and Ostrovsky (Definition 2.3.1.3 in Goldreich and Ostrovsky [1996]):

Goldreich-Ostrovsky security: *For any two input sequences y and y', if the length distributions $|A(\mathcal{M}, y)|$ and $|A(\mathcal{M}, y')|$ are identical, then $A(\mathcal{M}, y)$ and $A(\mathcal{M}, y')$ are identical.*

As we show, this requirement can be satisfied by creating an ORAM that makes sure that on any two distinct sequences y, y', the length distributions $|A(\mathcal{M}, y)|$ and $|A(\mathcal{M}, y')|$ differ. Note that no indistinguishability is required in that case and the ORAM can then reveal the access pattern of the input sequence.

To this end, we describe an ORAM with a constant overhead so that

$$|A(\mathcal{M}, y)| \in \{2|y|, 2|y| + 1\}$$

and the distribution $|A(\mathcal{M}, y)|$ encodes the sequence y. The ORAM proceeds by performing every operation y_i directly on the server followed by a read operation from address 1. After the last instruction in y, the ORAM selects a random sequence of operations r of length $|y|$ and if r is lexicographically smaller than y then the ORAM performs an extra read from address 1 before terminating. Note that this ORAM can be efficiently implemented using constant amount of internal memory by comparing the input sequence to the randomly selected one online. Also, the machine does not need to know the length of the sequence in advance. Finally, the length distribution $|A(\mathcal{M}, y)|$ is clearly different for each input sequence y. Given that the above definition of ORAM of Goldreich and Ostrovsky allows the dummy construction with a constant overhead, we do not hope to extend our lower bound towards this definition.

One could object that the above dummy ORAM exploits the fact that indistinguishability of access sequences must hold only if the length distributions are identical. However, it is possible to construct a similar dummy ORAM with low overhead satisfying even the following relaxation of the definition requiring indistinguishability of access sequences corresponding to any pair of y and y' for which $|A(M, y)|$ and $|A(M, y')|$ are statistically close (i.e., the indistinguishability is required for a potentially larger set of access patterns):

Relaxation of Goldreich-Ostrovsky security: *For any two input sequences y and y', if the length distributions $|A(\mathcal{M}, y)|$ and $|A(\mathcal{M}, y')|$ are statistically close, then $A(\mathcal{M}, y)$ and $A(\mathcal{M}, y')$ are statistically close.*

We show there is a dummy ORAM $\mathcal{M}$ with a constant overhead such that for any two input sequences y and y' which differ in their accessed memory locations, the statistical distance $\mathrm{SD}\left(|A(\mathcal{M}, y)|, |A(\mathcal{M}, y')|\right)$ is at least $\frac{1}{nM}$ (where $n = |y| = |y'|$ and M is the size of address range).

The ORAM $\mathcal{M}$ works as follows. At the beginning, the ORAM picks $i \in [n]$ and $r \in [M]$ uniformly at random. Then for $j = 1, \dots n$, it executes each of the input operations (o_j, a_j, d_j) directly on the server. For each $j < i$, it performs two additional reads from address 1 after executing the j-th input operation. For $j = i$, after the i-th input operation it performs two additional reads from address

1 if $r \leq a_i$, and it performs one additional read from address 1 if $r > a_i$. For $j > i$, it performs each of the input operations without any additional read.

It is straightforward to verify that the distribution of $|A(\mathcal{M}, y)|$ satisfies: for each $i \in [n]$, $\Pr[|A(\mathcal{M}, y)| = n + 2i] = \frac{a_i}{nM}$. Hence, for any pair y and y' of two input sequences of length n, if the sequences of addresses accessed by them differ then the statistical distance between the distributions of $|A(\mathcal{M}, y)|$ and $|A(\mathcal{M}, y')|$ is at least $1/nM$. If M is polynomial in n this means that their distance is at least $\frac{1}{\text{poly}(n)}$. Thus, $\mathcal{M}$ satisfies even the stronger variant of the definition from Goldreich and Ostrovsky [1996] even though its access sequence leaks the addresses from the input sequence.

It was previously shown by Haider et al. [2017] that there exists an ORAM construction which reveals all memory accesses from the input sequence while satisfying the definition of Goldreich and Ostrovsky from Goldreich and Ostrovsky [1996]. However, their construction has an *exponential* bandwidth overhead which makes it insufficient to demonstrate any issue with the definition of Goldreich and Ostrovsky. Clearly, any definition of ORAM can disregard constructions with super-linear overhead as a perfectly secure ORAM (with linear overhead) can be constructed by simply passing over the whole server memory for each input operation. Unlike the construction of Haider et al. [2017], our constructions of the dummy ORAMs with constant bandwidth overhead exemplify that the definition of Goldreich and Ostrovsky [1996] is problematic in the interesting regime of parameters.

Simulation-based definitions. The recent work of Asharov et al. [2018] employs a simulation-based definition parameterized by a functionality which implements an oblivious data structure. Our lower bounds directly extend to their stronger definition when the functionality implements Array Maintenance problem. Moreover, our techniques can be adapted to give lower bounds for functionalities implementing stacks, queues and others considered in Jacob et al. [2019].

Weak vs. strong computational security. In this work, we distinguish between weak and strong computational security (see Definition 3.2.7). Our techniques do not allow to prove matching bounds for ORAMs satisfying the two notions and we show $\Omega(\log(n))$ lower bound only w.r.t. strong computational security. Though, as we noted in Section 3.1, even the $\omega(1)$ lower bound for online ORAMs satisfying weak computational security is an interesting result in the light of the work of Boyle and Naor [2016]. It follows from Boyle and Naor [2016] that any super-constant lower bound for *offline* ORAM would imply super-linear lower bounds on size of sorting circuits – which would constitute a major breakthrough in computational complexity. The main result from Boyle and Naor [2016] can be rephrased using our notation as follows.

Theorem 3.5.1 (Theorem 3.1 Boyle and Naor [2016])**.** *Suppose there exists a Boolean circuit ensemble $C = \{C(n, w)\}_{n,w}$ of size $s(n, w)$, such that each $C(n, w)$ takes as input n words each of size w bits, and outputs the words in sorted order. Then for word size $w \in \Omega(\log(n)) \cap n^{o(1)}$ and constant internal memory $m \in \mathcal{O}(1)$, there exists a secure offline ORAM (as per Definition 2.8 Boyle and Naor [2016]) with total bandwidth and computation $\mathcal{O}(n \log(w) + s(2n/w, w))$.*

Moreover, the additive factor of $\mathcal{O}(n \log(w))$ follows from the transpose part of the algorithm of Boyle and Naor [2016] (see Figures 1 and 2 in Boyle and Naor [2016]). As Boyle and Naor showed in their appendix (Remark B.3 Boyle and Naor [2016]) this additive factor in total bandwidth may be reduced to $\mathcal{O}(n)$ if the size of internal memory is $m \geq w$. Thus, sorting circuit of size $\mathcal{O}(nw)$ implies offline ORAM with total bandwidth $\mathcal{O}(n + 2\frac{n}{w}w) = \mathcal{O}(n)$. Or the other way around, lower bound $\omega(n)$ for total bandwidth of offline ORAM implies $\omega(nw)$ lower bound for circuits sorting n words of size w bits, each.

We leave it as an intriguing open problem whether it is possible to prove an $\Omega(\log(n))$ lower bound for online ORAMs satisfying weak computational security.

4. Sorting on RAM

The work presented in this chapter is based on unpublished joint work with Michal Koucký, Michael Saks, and Veronika Slívová. We present an algorithm which works in expected time $\mathcal{O}(n \log\log(n))$ and additional linear space and is simple to analyse. To analyse the running time we need to know the coupon collector problem and basic algorithms and data-structures techniques such as hash-maps, binary search, amortized analysis of resizeable array[1], and any $\mathcal{O}(n \log(n))$ sorting algorithm. Our algorithm has larger expected running time than the expected $\mathcal{O}(n\sqrt{\log\log(n)})$ algorithm of Han and Thorup [2002]. The main advantage of our algorithm is that it is both easy to implement and easy to analyse.

4.1 Overview of Our $\mathcal{O}(n \log\log(n))$ Algorithm

We first present an overview of our algorithm (formally described in Algorithm 4). Then we provide more details and analysis in following sections.

- Pick uniformly at random a set S of samples from the inputs and sort the samples using any standard $\mathcal{O}(n \log(n))$ algorithm, e.g., mergesort or quicksort, (see Procedure 1).
- For each sample store all prefixes of its binary representation in a hash-map together with the index of the smallest and the index of the largest sample containing this prefix (see Procedure 2 which stores single prefix length).
- Place a bucket between every two consecutive samples, before the smallest and after the largest sample and fill them with numbers which belong between the samples as follows: For each input number, which is not equal to any sample, use binary search to find the length of the largest prefix such that there is a sample with the same prefix. Use constant number of comparisons to put this number into the corresponding bucket.
- With a good probability each block of poly-logarithmically many outputs contains at least one sample (by a coupon collector analysis). Thus with large probability each bucket contains at most poly-logarithmically many inputs.
- Sort each bucket using any standard $\mathcal{O}(n \log(n))$ algorithm, concatenate the results, and merge those with all occurrences of samples (see Procedure 3 for the code that extracts all samples).

4.2 Coupon Collector

In the *coupon collector problem* we are interested in how many independent uniformly random samples from a finite universe we need before seeing all elements of the universe. Another possible view of the problem in the balls and bins setting

[1] Such as `std::vector` in C++ (see Stroustrup [2013]).

is to consider throwing balls into n bins, each ball uniformly independently at random, and waiting until all bins have at least one ball. In the following text we call the finite universe the *coupon types* and the random samples *bought coupons* as bin could be mistaken for a bucket which is a term used in Algorithm 4. These problems are very well studied and have many important applications. We need just a simple folklore observation present for instance in the textbook of Motwani and Raghavan [1995].

Formally, for a given $N \in \mathbb{N}$ we sample from the set of coupon types $[N] = \{1, 2, 3, \ldots, N\}$. Each random sample is taken uniformly and independently on other samples from the set $[N]$ with repetition. The problem asks what is the probability of seeing each coupon type at least once among R samples.

Lemma 4.2.1 (Motwani and Raghavan [1995] (Coupon collector))**.** *Let there be N types of coupons. For any $\beta > 1$ the probability of not receiving all coupon types when buying $R = \beta N \ln(N)$ coupons uniformly independently at random can be bounded as:*

$$\Pr[\textit{missing a coupon type in } \beta N \ln(N) \textit{ coupons}] \leq N^{-(\beta-1)}$$

We assume that most courses on data structures and algorithms contain the used techniques (an $\mathcal{O}(n \log(n))$ sorting algorithm, hashing, binary search, and amortized analysis of resizeable array). But the coupon collector model is perhaps less known among undergraduates. Since our claim is that the algorithm is easy to analyse we also provide the proof of this lemma for the sake of completeness.

Proof as in Motwani and Raghavan [1995]. Let X_i^R be the indicator variable for the event that the i-th coupon type does not appear in the chosen R coupons. We may bound

$$\begin{aligned} \Pr[X_i^R] &= \left(1 - \frac{1}{N}\right)^R \\ &\leq e^{-R/N} && \text{(by the inequality } 1 - x \leq e^{-x}\text{)} \end{aligned}$$

Using union bound we get

$$\begin{aligned} \Pr[\text{missing a coupon type}] &\leq \sum_{i=1}^{N} \Pr[X_i^R] \\ &\leq N e^{-R/N} \\ &\leq N e^{-(\beta N \ln N)/N} && \text{(for } R = \beta N \ln N\text{)} \\ &\leq N^{-(\beta-1)} \end{aligned}$$

□

4.3 The Algorithm

We state our algorithm as taking n inputs $x_0, x_1, \ldots, x_{n-1} \in \{0, 1\}^w$ where $\log(n) \leq w \leq \log^3(n)$. This range of parameters works great since $\log(w) \leq$

$\log(\log^3(n)) = \mathcal{O}(\log\log(n))$. At the same time it is wide enough range of parameters, since when $w \leq \log(n)$ one may sort in linear time using bucket sort (see Cormen et al. [2009]) provided additional linear sized memory. On the other end of the range of parameters considered here one may sort in expected linear time when $w \geq \Omega(\log^2(n)\log\log(n))$ using the result of Belazzougui et al. [2014].

We present the algorithm with parts written as procedures (see Procedures 1, 2, and 3). These roughly correspond to the algorithm overview and are quite simple. The word-length w which is the number of bits in an integer is considered to be implicitly known in all procedures as well as in the main algorithm (this is often the case in modern programming languages).

There is a single division which could be substituted with a bit-shift to roughly approximate division by w without changing the asymptotic behaviour of the algorithm. The algorithm otherwise does not make expensive arithmetic operations (such as multiplication or division) explicitly. However many parts of the algorithm use hash-maps which may or may not use multiplication internally. The hash-maps also use randomness which again may or may not use multiplication during generation of (pseudo-)random bits. Another explicit use of randomness is in Procedure 1.

Arrays are numbered starting with zero. We use the notation $x_{0\dots b}$ to denote the $b+1$ most significant bits of an integer x. One could either use bit-shift or bit-wise AND with a bit-mask, both of these are present in many programming languages and are efficiently implementable in boolean circuits. We consider $x_{0\dots b}$ as being represented by an integer even if it is not the whole word (i.e., $b+1 < w$). This causes no problem since we do not mix prefixes of different lengths (each prefix length is kept in a separate hash-map). We measure size in the number of w-bit words unless stated otherwise.

Procedure 1: RandomSamples

Data: Array of integers $X = [x_0, x_1, \dots, x_{n-1}], x_j \in \{0,1\}^w$,
Integer k – the number of samples to take

Result: Sorted array of k samples from X (taken independently uniformly at random with repetition)

1 Let S be an empty array of size k
2 **for** $i \in \{0, 1, \dots, k-1\}$ **do**
3   Let $0 \leq r \leq n-1$ be a uniformly random integer
4     $S[i] = X[r]$
5 **end**
6 Sort S using mergesort
7 **return** S

Procedure 2: PreparePrefixes

Data: Sorted array of integers $S = [s_0, s_1, \ldots, s_{k-1}], s_j \in \{0,1\}^w$,
Integer b – the prefix length to hash (where $1 \leq b \leq w$)

Result: Hash-map mapping the prefix of length b (of each element of S) to the index of the largest and smallest element of S containing this prefix

1 Let *sampled_prefixes* be an empty hash-map
2 **for** $i \in \{0, 1, \ldots, |S| - 1\}$ **do**
3 Let $p = S[i]_{0 \ldots b-1}$ be the prefix of length b of the i-th sample
4 **if** *p is in sampled_prefixes* **then**
5   $sampled_prefixes[p].min = \min(i, sampled_prefixes[p].min)$
6   $sampled_prefixes[p].max = \max(i, sampled_prefixes[p].max)$
7 **end**
8 **else**
9   $sampled_prefixes[p].min = i$
10   $sampled_prefixes[p].max = i$
11 **end**
12 **end**
13 **return** *sampled_prefixes*

Procedure 3: ExtractSamples

```
  Data: Array of integers X = [x_0, x_1, ..., x_{n-1}], x_j ∈ {0,1}^w,
        Sorted array of integers S = [s_0, s_1, ..., s_{k-1}], s_j ∈ {0,1}^w
  Result: Sorted array of all elements from X (with all repetitions) which
          are equal to some element of S
  /* Count all samples:                                              */
1 Let count be an empty hash-map (integer to integer)
2 for s ∈ S do
3 |  count[s] = 0
4 end
5 for x ∈ X do
6 |  if x is present in count then
7 |  |  count[x] = count[x] + 1
8 |  end
9 end
  /* Concatenate the right number of samples:                        */
10 Let extracted_samples be an empty resizeable array of integers
11 for i ∈ {0, 1, ..., |S| - 1} do
12 |  if i > 0 and S[i] = S[i-1] then
   |  |  /* We have already extracted these samples.                 */
13 |  |  Continue
14 |  end
15 |  for j ∈ {0, 1, ..., count[S[i]] - 1} do
16 |  |  Append S[i] to extracted_samples
17 |  end
18 end
19 return extracted_samples
```

Algorithm 4: Sort pseudo-code

```
Data: Array of integers X = [x_0, x_1, ..., x_{n-1}], x_j ∈ {0,1}^w where
      log(n) ≤ w ≤ log^3(n)
Result: Array Y of sorted inputs (that is Y = [y_0, y_1, ..., y_{n-1}] where Y
        is a permutation of X and y_{j-1} ≤ y_j for each j ∈ [n − 1])
   /* Take ⌈n/w⌉ sorted random samples using Procedure 1:      */
 1 S = RandomSamples(X, ⌈n/w⌉)
   /* Prepare a hash-map for each prefix length b using
      Procedure 2:                                             */
 2 Let sampled_prefixes be an array of w empty hash-maps
 3 for b ∈ [w] do
 4 |   sampled_prefixes[b] = PreparePrefixes(S, b)
 5 end
   /* For each input x determine the corresponding bucket based
      on the longest prefix shared with a sample               */
 6 Let buckets be an array of |S| + 1 empty resizeable arrays
 7 for x ∈ [x_0, x_1, ..., x_{n-1}] do
 8 |   Use binary search to find the length b of the longest prefix of x that is
 |       in sampled_prefixes
 9 |   if b ≠ w then
10 |   |   Let ℓ = sampled_prefixes[b][x_{0...b-1}].min
11 |   |   Let u = sampled_prefixes[b][x_{0...b-1}].max
12 |   |   if x < S[ℓ] then
13 |   |   |   Append x to buckets[ℓ]
14 |   |   end
15 |   |   else
16 |   |   |   Append x to buckets[u + 1]
17 |   |   end
18 |   end
19 end
   /* Sort and concatenate the buckets                         */
20 for bucket ∈ buckets do
21 |   Sort bucket using mergesort
22 end
23 Concatenate buckets into not_sampled
24 sampled = ExtractSamples(X, S)                  // Procedure 3
25 Set result to merged arrays not_sampled and sampled
26 return result
```

Theorem 4.3.1. *Given n inputs $x_0, x_1, \ldots, x_{n-1} \in \{0,1\}^w$ where $\log(n) \leq w \leq \log^3(n)$ Algorithm 4 sorts its inputs using $\mathcal{O}(n)$ additional memory (measured in the number of w-bit words). The expected running time of Algorithm 4 is $\Theta(n \log\log(n))$ where the expectation is taken over random bits of the algorithm (namely the choice of samples and random bits of the internal hash-map).*

Proof. First we argue that the algorithm sorts its inputs. The algorithm puts each input number into a single bucket. Since b is the length of the longest prefix of our input x such that there is a sample with the same prefix (see Algorithm 4,

Lines 4 and 8) in the hash-map

$$sampled_prefixes[b][x_{0\ldots b-1}].min, \text{ resp. } sampled_prefixes[b][x_{0\ldots b-1}].max,$$

we have the index of the smallest, resp. largest, sample which starts with this prefix (see Procedure 2). If x is equal to one of the samples we know that $b = w$ and that x gets counted (see Procedure 3 called in Algorithm 4, Line 24). Otherwise $b < w$ and moreover $x_{0\ldots b}$ is not present in the hash-map $sampled_prefixes[b+1]$ and thus all samples that agree with x on the b most significant bits but differ from x on the $(b+1)$-th bit:

$$\begin{aligned}
\ell &= sampled_prefixes[b][x_{0\ldots b-1}].min \\
u &= sampled_prefixes[b][x_{0\ldots b-1}].max \\
S[\ell]_{0\ldots b-1} &= x_{0\ldots b-1} && \text{(first } b \text{ bits of } x \text{ are present in the hash-map)} \\
S[\ell]_{0\ldots b} &\neq x_{0\ldots b} && \text{(first } b+1 \text{ bits of } x \text{ are not present in the hash-map)} \\
S[\ell]_{0\ldots b} &= S[u]_{0\ldots b}
\end{aligned}$$

thus either

$$x < S[sampled_prefixes[b][x_{0\ldots b-1}].min]$$

or

$$x > S[sampled_prefixes[b][x_{0\ldots b-1}].max]$$

in either case x gets into the corresponding bucket (see Algorithm 4, Line 12). Since each bucket is sorted (see Algorithm 4, Line 21) and the buckets are also sorted their concatenation merged with all occurrences of samples contains exactly all the inputs sorted.

We bound the expected runtime of each part of Algorithm 4:

Procedure 1: We call the procedure with $k = \lceil n/w \rceil$. Sampling is trivially $\mathcal{O}(k) \leq \mathcal{O}(n)$. Mergesort takes time $\mathcal{O}(n \log(n))$ (see for instance the textbook of Knuth [1997]). We use it on an array of length $k = \lceil n/w \rceil$ thus it takes time

$$\begin{aligned}
\mathcal{O}(k \log(k)) &= \mathcal{O}\left(\left\lceil \frac{n}{w} \right\rceil \log\left(\left\lceil \frac{n}{w} \right\rceil\right)\right) \\
&= \mathcal{O}\left(\frac{n}{w} \log\left(\frac{n}{w}\right)\right) \\
&\leq \mathcal{O}\left(\frac{n}{w} \log\left(n\right)\right) \\
&\leq \mathcal{O}(n). && \text{(as } w \geq \log(n)\text{)}
\end{aligned}$$

Procedure 2: The procedure is called once for each prefix length. There are w prefixes in each sample and $k = \lceil n/w \rceil$ samples, thus in total there are $\Theta(n)$ prefixes to be inserted. To add into or to retrieve a key-value pair from a hash-map it takes expected time $\mathcal{O}(1)$ (see for instance the textbook of Cormen et al. [2009]).

Line 8: For any $b \in [w]$ we may query the corresponding hash-map to find out if the prefix $x_{0\ldots b-1}$ is present (in expected time $\mathcal{O}(1)$). If the prefix is not present for some $b \in [w-1]$ then it is not present for any $b' > b$. Similarly

if the prefix is present for some $b \in [w]$ then it is present for all $b' \leq b$. Thus we may use binary search (see for instance the textbook of Cormen et al. [2009]) which does $\log(w) \leq \log(\log^3(n)) = 3\log\log(n)$ queries to the hash-map to determine the maximum common prefix with some sample. This can be done in expected time $\mathcal{O}(\log\log(n))$ for each of our inputs.

Line 13: If the algorithm remembers the size and number of elements in each array, it may double the size whenever the array is full. Thus getting amortized time $\mathcal{O}(1)$ to add a single input x (see for instance the textbook of Cormen et al. [2009], Chapter on Amortized Analysis).

Line 21: The size of a bucket is the number of non-sampled inputs between the samples delimiting the bucket. To analyse the size of each bucket let us consider the samples as being sampled uniformly at random from the already sorted sequence. The idea that is formalized in the following paragraph is to think of the samples as coupons bought and of blocks of sorted outputs as of coupon types. Thus when we sample a number that should be present inside a block of outputs we say that we have bought the corresponding coupon. The process of the algorithm is identical to the coupon collector model only when the inputs are distinct. If there is an input that appears in more output blocks then by sampling this input we collect many coupons. But we may think of sampling output indices $j \in [n]$ rather than the outputs y_j.

Without loss of generality we may assume that n is divisible by $k/\lceil 10\ln(n)\rceil$ (otherwise we could add several copies of the minimal element where we would count the logarithm just once during the algorithm). We divide the sorted outputs into $k/\lceil 10\ln(n)\rceil$ blocks[2] each consisting of

$$\begin{aligned}\frac{n}{k/\lceil 10\ln(n)\rceil} &= \frac{n}{\lceil n/w\rceil/\lceil 10\ln(n)\rceil} \\ &= \Theta(w\log(n)) \\ &\leq \mathcal{O}(\log^4(n))\end{aligned}$$

consecutive outputs. Then we invoke Lemma 4.2.1 with:

$$\begin{aligned}R &= k = \lceil n/w\rceil && \text{(coupons bought)} \\ N &= k/\lceil 10\ln(n)\rceil && \text{(the number of coupon types)} \\ \beta &= 10\end{aligned}$$

Thus

$$\begin{aligned}R &\geq 10\ln(n)N \\ &\geq 10N\ln(N) && (\text{as } N = \Theta(n/(w\ln(n))) \leq n)\end{aligned}$$

Observe that $N^{-9} \leq n^{-8}$ for large enough n. Thus by Lemma 4.2.1 with probability at least $1 - N^{-9} \geq 1 - n^{-8}$ each block of outputs will contain at least one sample therefore no bucket will be of size larger than $\mathcal{O}(\log^4(n))$ (no bucket spans over more than two blocks). With probability at most

[2]Buckets appear in the algorithm, blocks are used only in the analysis.

n^{-8} there will be a large bucket and we may upper bound the time to sort all buckets by $\mathcal{O}(n \log(n))$ – the time of mergesorting all inputs. We may upper bound $\mathbb{E}[T]$, the expected time of mergesorting all buckets, by

$$\begin{aligned}
\mathbb{E}[T] &\leq \left(1 - n^{-8}\right) \Theta \left(\sum_{bucket \in buckets} |bucket| \log(|bucket|) \right) + n^{-8}\Theta(n \log(n)) \\
&\leq \left(1 - n^{-8}\right) \Theta \left(\log\log(n) \sum_{bucket \in buckets} |bucket| \right) + n^{-8}\Theta(n \log(n)) \\
&= \Theta(n \log\log(n)) \qquad\qquad (w \leq \log^3(n))
\end{aligned}$$

Line 23: We allocate an array of size n and copy the sorted buckets. Since each bucket itself is sorted and all buckets are also sorted this can be done in time $\mathcal{O}(n)$. Merging two arrays of size $\mathcal{O}(n)$ can also be done in time $\mathcal{O}(n)$.

Now we argue that the additional space used is linear in n (we count the number of words, so both the input and output consist of n words or nw bits). The number of samples is sub-linear in n and the number of their prefixes is $\mathcal{O}(n)$. All together the hash-maps inside $sampled_prefixes$ have size linear in the number of prefixes of samples, with our choice of parameters there are $\mathcal{O}(n)$ entries in the hash-map thus it has size linear in n (see for instance the textbook of Cormen et al. [2009]). Each of the buckets has size at most twice the number of elements present in it (when we just add elements this is sufficient for amortized analysis) and each input is placed in a single bucket, thus all the buckets together have size $\mathcal{O}(n)$. □

4.4 Practical Implementation

A naive version of Algorithm 4 can be implemented rather easily using the constructs present in the programming language of the readers choice. We note that for instance one may use data-structures and algorithms present in the C++ standard library (see Stroustrup [2013]):

`std::unordered_map<T, size_t>`, which implements a hash table

`std::vector<T>`, which implements a resizeable array

`std::sort`

We have not determined the constants hidden in the $\mathcal{O}$ notation, but we claim that these are not astronomical. However just the use of a hash-map means that our algorithm is not competitive when compared with a well tuned implementation of sorting present in many programming languages. Another practical disadvantage is that our algorithm allocates additional memory. The additional memory is linear in the input size but the multiplicative constant is not small.

4.5 Short Word Lengths

We can modify our algorithm to get expected running time $\mathcal{O}(n \log(w/\log(n)))$. When the word length w is small, for instance $w = \log(n) \log\log(n)$, this results in a faster algorithm. The price of this modification is that the algorithm is more complex and uses a non-trivial sorting algorithm of Belazzougui et al. [2014]. Similar modifications have been also made to the algorithms Han and Thorup [2002] and Andersson et al. [1998]. Thus we just briefly outline the modification in the following paragraph.

When we say that we radixsort elements according to some key we mean that we create a resizeable array for each possible value of the key, add our elements there, and concatenate the arrays in the correct order. Specially elements with the same key retain their order (the sort is stable). The main idea of the modification is to limit the precision of the binary search to find the length b of the longest prefix of an input x that is shared with some sample. We rather determine an interval of length $\log(n)$ in which b lies and use radixsort to subdivide the inputs into small buckets. That is, we find an integer $k \in \mathbb{N}$ such that $b \in [k \log(n), (k+1)\log(n) - 1]$. We augment each input number with its k (instead of x we store a pair (x, k)) and radixsort all inputs by their k. For each input (x, k) we know that x shares at least $k \log(n)$ most significant bits with some sample. For each value of k we further sort all inputs with this k (that is all (x, k)) by their $k \log(n)$ most significant bits (this is done similarly to our original algorithm by dividing the inputs by their prefix in common with some sample). We radixsort all inputs where for the key we use the respective $[k \log(n), (k+1)\log(n) - 1]$ bits of the individual input number. Thus we have grouped inputs with equal $(k+1)\log(n)$ most significant bits together into buckets (note that each bucket here is a subset of some bucket in our original algorithm). We sort each bucket using the algorithm of Belazzougui et al. [2014]. For each k we concatenate all buckets containing inputs (x, k) into a sorted array. We then merge the $\lceil w/\log(n) \rceil$ arrays, always merging the shortest two arrays together.

4.6 Open Questions

The main question is what modifications to this algorithm can be made?

- Can this algorithm be made parallel? Two places that seem to be not trivial to do in parallel are using hash-maps and later placing the numbers into correct buckets. The last merge step is rather easy as the buckets interleave with copies of samples (first the first bucket, then all copies of the smallest sample, second bucket, etc.).

- If we substitute sorting of the samples and buckets with a faster algorithm the main bottleneck is the binary search. Single hash-map query could be sufficient if inputs are assumed to be distributed uniformly independently at random. But in this case one could use radixsort on $\mathcal{O}(\log(n))$ most significant bits of each number and with a good probability get small buckets. Are there any distributions of inputs which are not trivial to sort by themselves but we can significantly reduce expected complexity of prefix search (binary search part of Algorithm 4)?

Conclusion

Sorting has been one of the central themes of computer science for more than a century (as discussed in Chapter 1). Still many problems remain wide open. We provide conclusions and reflection connected to each of our results at the end of the respective chapters.

4.6.1 Acknowledgements

We are grateful for insightful discussions with Mike Saks on sorting. We also thanks Igor Sergeev for pointing us to the paper of Kospanov [1994] and anonymous reviewers of ICALP 2021 for their comments which improved the presentation of Chapter 2.

We wish to thank Oded Goldreich for clarifications regarding the ORAM definitions in Goldreich [1987], Ostrovsky [1990], Goldreich and Ostrovsky [1996] and Jesper Buus Nielsen for clarifying the details of the lower bound for computationally secure ORAMs from Larsen and Nielsen [2018]. We are also thankful to the anonymous TCC 2019 reviewers for insightful comments that helped us improve the presentation of Chapter 3.

Bibliography

Peyman Afshani, Casper Benjamin Freksen, Lior Kamma, and Kasper Green Larsen. Lower Bounds for Multiplication via Network Coding. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:12, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-109-2. doi: 10.4230/LIPIcs.ICALP.2019.10. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10586`.

Rudolf Ahlswede, Ning Cai, S-YR Li, and Raymond W. Yeung. Network information flow. *IEEE Transactions on information theory*, 46(4):1204–1216, 2000.

Miklós Ajtai. Oblivious RAMs without cryptographic assumptions. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 181–190, 2010.

Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c\log(n)$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. *IACR Cryptology ePrint Archive*, 2018:892, 2018.

Gilad Asharov, Wei-Kai Lin, and Elaine Shi. Sorting short keys in circuits of size $o(n\log n)$. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2249–2268. SIAM, 2021.

Kenneth Edward Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected linear time sorting for word size $\Omega(\log^2(n)\log\log(n))$. In *Scandinavian Workshop on Algorithm Theory*, pages 26–37. Springer, 2014.

Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

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, Cambridge, MA, USA, January 14-16, 2016*, pages 357–368, 2016.

Martin Campbell-Kelly and William Aspray. Computer: A history of the information machine.

Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 62–81, 2014.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Henry Corrigan-Gibbs and Dmitry Kogan. The function-inversion problem: Barriers and opportunities. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography*, pages 393–421, Cham, 2019. Springer International Publishing. ISBN 978-3-030-36030-6.

Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 144–163, 2011.

Pavel Dvořák, Michal Koucký, Karel Král, and Veronika Slívová. Data structures lower bounds and popular conjectures. *arXiv preprint arXiv:2102.09294*, 2021.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 265–284, 2006.

Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, page 997–1008, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367059. doi: 10.1145/3313276.3316337. URL `https://doi.org/10.1145/3313276.3316337`.

Robert W. Floyd. Permuting information in idealized two-level storage. In *Complexity of computer computations*, pages 105–109. Springer, 1972.

Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 345–354, 1989.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 285–297. IEEE, 1999.

Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International*

Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings, pages 1–18, 2013.

Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194, 1987.

Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $\mathcal{O}(n \log n)$ time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 684–693, 2014.

Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 576–587, 2011.

Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 95–100, 2011.

Syed Kamran Haider, Omer Khan, and Marten van Dijk. Revisiting definitional foundations of oblivious RAM for secure processor implementations. *CoRR*, abs/1706.03852, 2017. URL `http://arxiv.org/abs/1706.03852`.

Yijie Han and Mikkel Thorup. Sorting integers in $O(n\sqrt{\log \log n})$ expected time and linear space. In *IEEE Symposium on Foundations of Computer Science (FOCS'02)*, 2002.

Fred C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, 1965.

Fred C. Hennie and Richard Edwin Stearns. Two-tape simulation of multitape turing machines. *Journal of the ACM (JACM)*, 13(4):533–546, 1966.

Charles Antony Richard Hoare. Algorithm 64: quicksort. *Communications of the ACM*, 4(7):321, 1961.

Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

John Edward Hopcroft, Rajeev Motwani, and Jeffrey David Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

Pavel Hubáček, Michal Koucký, Karel Král, and Veronika Slívová. Stronger lower bounds for online ORAM. In *Theory of Cryptography Conference*, pages 264–284. Springer, 2019.

Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2439–2447, 2019.

Stasys Jukna. *Boolean function complexity: advances and frontiers*, volume 27. Springer Science & Business Media, 2012.

Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In *Italian Conference on Algorithms and Complexity*, pages 217–228. Springer, 1997.

David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983.

Donald Ervin Knuth. The art of computer programming, volume 3: Sorting and searching. 1997.

Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018. URL http://arxiv.org/abs/1801.01203.

Andrei N. Kolmogorov. On tables of random numbers. *Theoretical Computer Science*, 207(2):387–395, 1998.

Ilan Komargodski and Wei-Kai Lin. Lower Bound for Oblivious RAM with Large Cells. 2020.

E. S. Kospanov. Scheme realization of the sorting problem. *Diskretnyi Analiz i Issledovanie Operatsii*, 1(1):13–19, 1994.

Michal Koucký and Karel Král. Sorting short integers. *arXiv preprint arXiv:2102.10027*, 2021. Accepted to ICALP 2021.

Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156, 2012.

Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 523–542, 2018.

Kasper Green Larsen, Mark Simkin, and Kevin Yeo. Lower bounds for multi-server oblivious rams. In *Theory of Cryptography Conference*, pages 486–503. Springer, 2020.

Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.

Zongpeng Li and Baochun Li. Network coding: The case of multiple unicast sessions. *Proceedings of the 42nd Allerton Annual Conference on Communication, Control, and Computing*, 10 2004.

Wei-Kai Lin and Elaine Shi. Optimal sorting circuits for short keys. *arXiv preprint arXiv:2102.11489*, 2021.

Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2419–2438. SIAM, 2019.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 973–990, 2018.

F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, Amsterdam, 1977.

Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge university press, 1995.

Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523, 1990.

John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 871–882, 2018.

Michael S. Paterson. Improved sorting networks with $\mathcal{O}(\log n)$ depth. *Algorithmica*, 5(1):75–92, 1990.

Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

Wolfgang J. Paul. Kolmogorov complexity and lower bounds. In *FCT*, pages 325–334, 1979.

Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private RAMs. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, pages 404–434, 2019.

Holger Petersen. Sorting and element distinctness on one-way turing machines. In *International Conference on Language and Automata Theory and Applications*, pages 433–439. Springer, 2008.

Nicholas Pippenger. Self-routing superconcentrators. *Journal of Computer and System Sciences*, 52(1):53–60, 1996.

K. Rüdiger Reischuk. *Einführung in die Komplexitätstheorie: Band 1: Grundlagen Maschinenmodelle, Zeit-und Platzkomplexität, Nichtdeterminismus*, volume 1. Springer-Verlag, 1999.

Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: closing the gap between small and large client storage oblivious RAM. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

Robert Sedgewick. *Quicksort*. PhD thesis, Stanford University, 1975. URL `https://www.cs.princeton.edu/~rs/papers/Quicksort.pdf`.

Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

Hanns-Jörg Stoß. Rangierkomplexität von permutationen. *Acta Informatica*, 2 (1):80–96, 1973.

Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

Salil Pravin Vadhan. *A Study of Statistical-Zero Knowledge Proofs*. PhD thesis, Massachusetts Institute of Technology, 9 1999.

Leslie G. Valiant. On non-linear lower bounds in computational complexity. In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 45–53, 1975.

Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *International Symposium on Mathematical Foundations of Computer Science*, pages 162–176. Springer, 1977.

Leslie G. Valiant. Exponential lower bounds for restricted monotone circuits. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 110–117, 1983.

Leslie G. Valiant. Why is boolean complexity theory difficult. *Boolean Function Complexity*, 169(84-94):4, 1992.

Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84. IEEE, 1975.

Emanuele Viola. Lower bounds for data structures with space close to maximum imply circuit lower bounds. *Theory of Computing*, 15(18):1–9, 2019. doi: 10.4086/toc.2019.v015a018. URL `http://www.theoryofcomputing.org/articles/v015a018`.

Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, (1):14–17, 1964.

Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 850–861, 2015.

Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202, 2014.

Mor Weiss and Daniel Wichs. Is there an oblivious RAM lower bound for online reads? In *Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II*, pages 603–635, 2018.

Sebastian Wild, Markus Nebel, Raphael Reitzig, and Ulrich Laube. Engineering java 7's dual pivot quicksort using malijan*: Adventures with just-in-time compilation. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 55–69. SIAM, 2013.

Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

List of Figures

2.1 An example of splitting numbers where $b = 3$. The input number $x = 5$ is represented as 0101 and is split into $x_L = 100, x_R = 001$ which are themselves split recursively. The bottom nodes form the output. 16

2.2 An example of a sorting network with three inputs (the horizontal lines), three comparators (the vertical lines), and depth three. The inputs on the left are numbers x, y, z and after each comparator we noted what is on the horizontal line. Note that the bottom most output is $\max(\max(x, y), \max(\min(x, y), z)) = \max(x, y, z)$ and the middle one is $\min(\max(x, y), \max(\min(x, y), z))$ which is the median. 18

2.3 Sorting by the median key using unstable sorting by a single bit. Inputs are at the bottom and outputs are at the top. The symbol $<$ means an element with a key less than the median key, $=$ stands for an element with a key equal to the median key, and $>$ stands for an element with a key greater than the median key. Blanks are "dummy" elements and arrows are either use of sorting by a single bit or fixed permutations. 28

3.1 Schema of online ORAM from Definition 3.2.7. 35

List of Publications

1. Martin Balko, Josef Cibulka, Karel Král, and Jan Kynčl. Ramsey numbers of ordered graphs. *Electronic Notes in Discrete Mathematics*, 49: 419–424, 2015. ISSN 1571-0653. `https://www.sciencedirect.com/science/article/pii/S1571065315001055` EuroComb 2015

2. Bernd Gärtner, Thomas Dueholm Hansen, Pavel Hubáček, Karel Král, Hagar Mosaad, and Veronika Slívová. ARRIVAL: Next Stop in CLS. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107, pages 60:1–60:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018

3. Pavel Hubáček, Michal Koucký, Karel Král, and Veronika Slívová. Stronger Lower Bounds for Online ORAM. In *Theory of Cryptography Conference*, pages 264–284. Springer, 2019

4. Pavel Hubáček, Chethan Kamath, Karel Král, and Veronika Slívová. On Average-Case Hardness in TFNP from One-Way Functions. In *Theory of Cryptography Conference*, pages 614–638. Springer, 2020

5. Pavel Dvořák, Michal Koucký, Karel Král, and Veronika Slívová. Data Structures Lower Bounds and Popular Conjectures. *arXiv:2102.09294, 2021*, *Under submission*

6. Michal Koucký and Karel Král. Sorting Short Integers. *arXiv:2102.10027*, *Accepted to ICALP 2021*