Towards Simpler Sorting Networks and Monotone Circuits for Majority

Natalia Dobrokhotova-Maikova   Alexander Kozachinskiy
Vladimir Podolskii

Abstract

In this paper we study the problem of computing majority function by low depth monotone circuits and a related problem of constructing low-depth sorting networks. We consider both the classical setting with elementary operations of arity 2 and the generalized setting with operations of arity k, where k is a parameter. For both problems and both settings there are various constructions known, the minimal known depth being logarithmic. However, there is currently no known construction that simultaneously achieves sub-log-squared depth, effective constructability, simplicity, and has a potential to be used in practice. In this paper we make progress towards resolution of this problem.

For computing majority by standard monotone circuits (gates of arity 2) we provide an explicit monotone circuit of depth $O(\log_2^{5/3} n)$. The construction is a combination of several known and not too complicated ideas.

For arbitrary arity of gates k we provide a new sorting network architecture inspired by representation of inputs as a high-dimensional cube. As a result we provide a simple construction that improves previous upper bound of $4\log_k^2 n$ to $2\log_k^2 n$. We prove the similar bound for the depth of the circuit computing majority of n bits consisting of gates computing majority of k bits. Note, that for both problems there is an explicit construction of depth $O(\log_k n)$ known, but the construction is complicated and the constant hidden in O-notation is huge.

1   Introduction

A sorting network receives an array of numbers and outputs the same numbers in the non-decreasing order. It consists of *comparators*, each of which is given some fixed pair of array entries as an input and it swaps them if they are not in the non-decreasing order. The main parameters of a sorting

1

network are the size, that is, the number of comparators, and the depth, that is, the number of layers in the network, where each layer consists of several comparators applied to disjoint pairs of variables. Sorting networks are a classical model in theoretical computer science with vast literature devoted to them, see, for example [4, 1, 22, 26, 24, 17, 28, 7], see also the Knuth's book [18] and the Baddar's and Batcher's book [3]. Despite considerable efforts, still there are many open problems related to sorting networks. In this paper, our main interest is the depth of sorting networks.

There is a related setting of computing majority function by monotone Boolean circuits. Majority function receives as input a sequence of n bits and outputs 1 if and only if more than a half of the inputs are 1's. Monotone Boolean circuits consist of AND and OR gates of fan-in 2. Constructing a monotone Boolean circuit for the majority function can only be easier than constructing a sorting network. This is because a sorting network can be transformed into a monotone Boolean circuit which computes majority and has the same depth. Indeed, if we restrict inputs to $\{0, 1\}^n$, then each comparator can be simulated by a pair of AND and OR gates (AND computes the minimum of two Boolean inputs and OR computes the maximum). And the majority is just the median bit of the sorted array.

For the depth of sorting networks, there are several simple and practical constructions of depth $\Theta(\log^2 n)$ [18, 4, 24]. A construction with $O(\log n)$ depth was given by Ajtai, Komlós and Szemerédy [1] and is usually referred to as the AKS sorting network. Although their bound on the depth is asymptotically optimal, the construction is very complicated and impractical due to a large constant hidden in the O-notation. There are some simplifications and improvements of this construction [26, 28], but the construction is still elaborate and is not practical. As for the lower bounds, there is a folklore $(2 - o(1)) \log_2 n$ depth lower bound for networks sorting n numbers. It was improved by Yao [34] and later by Kahale et al. [17] with the current record about $3.27 \log_2 n$.

As we discussed above, any construction for a sorting network translates to a monotone circuit for majority of the same depth. In particular, we get an $O(\log n)$-depth monotone circuit for majority from the AKS sorting network. Yet again, the resulting circuit has the same disadvantages as the AKS construction. But in contrast to sorting networks, there is an alternative construction of a monotone depth-$O(\log n)$ Boolean circuit for majority due to Valiant [32]. His construction is simple and has a reasonable constant hidden in the O-notation, but it is randomized. It was partially derandomized and made closer to practice by Hoory, Magen and Pitassi [14]. But still all known fully deterministic constructions that are simple and

practical are of depth $\Theta(\log^2 n)$. Thus, there is an open problem for both sorting networks and monotone circuits for majority to come up with simple and deterministic construction of sub-log-squared depth.

One potential approach to this is to consider sorting networks with comparators that have $k > 2$ inputs. We will call them k-sorting networks. They appear in the literature since 70s, the setting is mentioned already in the Knuth's book [18, Problem 5.3.4.54], followed by numerous works [30, 25, 5, 23, 10, 21, 29, 13, 35]. They are usually studied to better understand the structure of ordinary sorting networks (for example, a version of AKS sorting network with improved constant relies on k-sorting network in intermediate constructions [8]). In particular, k-sorting networks are closely related to recursive constructions of sorting networks. Having a good construction of a k-sorting network, one can apply it to its own comparators, getting a construction with smaller k, until eventually k becomes 2, and we get an ordinary sorting network.

Parker and Parbery [25] constructed a simple and potentially practical k-sorting network of depth $\leqslant 4\log_k^2 n$ (in case when n is an integral power of k). At the same time, as Chvátal shows in his lecture notes [8], the AKS sorting network also generalizes to this setting, giving a construction of depth $O(\log_k n)$. However, as with the AKS sorting network itself, this construction is complicated and impractical. So the search for simple constructions continues. As for the lower bounds, any k-sorting network with n inputs must have depth at least $\log_k n$ because otherwise outputs cannot be connected to all n inputs. Dobokhotova-Maikova et al. [11] improved this bound to roughly $2\log_k n$. They also found optimal values of k for small values of depth d. More specifically, for sorting networks of depth $d = 1, 2$ they show that k cannot be smaller than n, for $d = 3$ the optimal value is $k = \left\lceil \frac{n}{2} \right\rceil$ and for $d = 4$ the optimal value is $k = \Theta(n^{2/3})$. These results indicate that small depth k-sorting networks are not enough for iterative approach to sub-log-squared sorting network and we need either good k-sorting network constructions of depth greater than 4, or additional ideas.

Just as with sorting networks, we can consider circuits for majority function that are constructed from *threshold* gates of fan-it at most k. A threshold gate is a Boolean function that first sorts its input bits in the non-decreasing order, and then outputs the ith one from the beginning, for some fixed $1 \leq i \leq k$. For $k = 2$, AND and OR are the only two threshold functions. In general, there are k threshold functions of fan-in k. By taking one copy of each, we get a comparator of arity k. Thus, as in the case $k = 2$, a k-sorting network can be transformed into a circuit of the same

3

depth which computes majority and consists of threshold gates of fan-in k. In other words, constructing a k-sorting network can only be harder than constructing a circuit for majority with threshold gates of fan-in k.

There is a line of work, initiated by Kulikov and Podolskii [20], which addresses the following question: given d and n, what is the minimal k for which there exists a circuit with threshold gates of fan-in k, which has depth d and computes majority on n bits? The paper [20] shows that, up to a polylogarithmic factor, $k \geq n^{14/(7d+6)}$. In subsequent works, special attention was given to the case $d = 2$. In this case, the lower bound of [20] is $k \geq n^{14/20}$. It was improved to $k \geq n^{4/5}$ by Engels et al. [12]. Then a linear lower bound $k \geq n/2 - o(n)$ was obtained by Hrubes et al. [15]. An upper bound $k \leq 2n/3 + O(1)$ was given in [27]. Let us also mention an upper bound $k \leq n - 2$ for circuits that only use majority gates [19, 2].

Now, for $d \geq 3$ the situation is less clear. For $d = 3$, the paper [20] gave an upper bound $k = O(n^{2/3})$. In turn, their lower bound in this case is of order $n^{14/27}$. We are not aware of any non-trivial upper bound for $d \geq 4$.

Our results.   In this paper we make progress towards better constructions of monotone circuits for majority and sorting networks.

First, we give an explicit and reasonably simple construction of a monotone circuit for majority of depth $O(\log^{5/3} n)$.

Theorem 1. *There is a polynomial time constructable monotone circuit for majority of polynomial size and depth $O(\log^{5/3} n)$.*

Our proof combines several relatively simple steps. We start with partial derandomization of Valiant's construction using some ideas from the paper by Cohen et al. [9]. Next we apply to the resulting randomized circuit two operations several times. The first of them is a brute-force derandomization, that searches through all possible random bits of the randomized circuit. The second one is a composition with a k-sorting network of depth $O(\log_k^2 n)$. For such a network we can use either the construction of Parker and Parbery [25], or, for the better constant, our next result.

In our second result we come up with the new architecture for k-sorting networks. As an application of this architecture we construct a k-sorting network of depth $2 \log_k^2 n$, improving the constant compared to the result of [25]. More precisely, we prove the following theorem.

Theorem 2. *For any n and for any k such that $\log k = \omega(\log \log n)$ (or, to put it differently, k is growing faster than any $\mathsf{polylog}(n)$), there exists a k-sorting network of depth at most $(2 + o(1)) \log_k^2 n$.*

The key idea behind this construction is to represent the input array as a hypercube of high dimension and sort various sections of this cube. We note that the idea of representing an array as a multidimensional structure is not new, for example, Leighton [22] in his ColumnSort represented the array as a two dimensional table. However, in our construction it is important that we use the dimension greater than 2, since we use the fact that the sections of the cube have non-trivial intersection. On the conceptual level, the main novelty in our construction is the notion of s-sorting. We call the array s-sorted if the whole array is sorted correctly apart from some interval of length at most s. Most (if not all) log-squared-depth sorting network constructions adopt the divide and conquer strategy. The $O(\log_k^2 n)$-depth construction in [25] is not an exception, to sort an array of size n, they split it into subarrays of size n/k, sort them recursively and merge them afterward. However, merging k subarrays using k-sorting network is relatively expensive. To improve over previous construction, we work with s-sorted subarrays instead. We show how to merge them effectively (using the hypercube idea) and then show how we can build a recursive construction based on them.

To additionally illustrate applications of our construction, we consider constant depth sorting networks and circuits for majority. We show that there is a MAJ_k-circuit for MAJ_n for $k = O(n^{3/5})$. For a second application we address the question of k-sorting networks for $k = O(n^{1/2})$. In [18] Knuth posed a problem of constructing a minimal depth k-sorting network for the input of size k^2. Parker and Parbery [25] gave a construction of depth 9. We improve this to depth 8 at the cost of using comparators of size $O(k)$ for k^2 input size.

The rest of the paper is organized as follows. In Section 2 we provide necessary preliminary information. In Section 3 we construct a monotone circuit for majority of depth $O(\log^{5/3} n)$. In Section 4 we provide a new construction of k-sorting networks and deduce the corollaries. In Section 5 we discuss some open problems.

2 Preliminaries

We use the standard notation $[n] = \{1, \ldots, n\}$. We sometimes omit the base of the logarithms, by default we assume that the base is 2.

2.1 Sorting Networks

A depth-d k-sorting network with n inputs consists of $d + 1$ arrays $A_1, \ldots, A_{d+1}$, each of length n. Between any two arrays A_i and A_{i+1} there

is a *layer of comparators* (the first layer is between A_1 and A_2, the second layer is between A_2 and A_3, and so on). A layer of comparators is a partition of the set $\{1, 2, \ldots, n\}$ into subsets of size at most k called *comparators*.

The input is given in an array A_1 and all other arrays are computed by the network one by one in the following way. If $S \subseteq [n]$ is a comparator from the ith layer, then it is applied to the entries $\{A_i[j] \mid j \in S\}$. It sorts their values in the non-decreasing order and puts the results into the entries $\{A_{i+1}[j] \in A_{i+1} \mid i \in S\}$. We say that a network is *sorting* if for any input A_1 the array A_{d+1} is sorted.

We reserve the name *sorting network* for 2-sorting networks.

It is well known that to check that the sorting network sorts all possible inputs, it is enough to check that it sorts just 0/1-inputs.

Lemma 3 (Zero-one principle [18]). *A network with n inputs sorts all integer sequences in the non-decreasing order if and only if it sorts all sequences from $\{0, 1\}^n$ in the non-decreasing order.*

By this principle, when constructing sorting networks, we can assume that each input cell receives either 0 or 1.

The following simple observation will be useful to us.

Lemma 4. *If t largest or t smallest entries in the array are positioned correctly (i.e., in the last t cells and in the first t cells, respectively), then after the application of several comparators they are still positioned correctly.*

Proof. We can show by induction on i that the smallest and the largest entries do not move if they are already positioned correctly. The key observation is that if some of these entries are inputted into one of the comparators S, they will not be moved. $\square$

2.2 From Sorting Networks to Majority Circuits

We use the standard notion of Boolean circuits (see, e.g. [16]). As inputs, we allow Boolean variables and Boolean constants 0 and 1. The size of the circuit is the number of gates in it.

Given a k-sorting network we can get a circuit computing majority from it. More specifically, restrict the inputs to the network to $\{0, 1\}^n$ and consider one k-comparator S. Note that its kth output is equal to 1 if and only if there is at least one 1 in the input. In other words, the kth output is equal to OR of input bits. Its $(k-1)$th output is equal to 1 if and only if there

are at least two 1s in the input. More generally, it is easy to see that the $(k-i)$th output of the k-comparator outputs a threshold function

$$\mathrm{THR}_k^i(x) = \begin{cases} 1 & \text{if } |x| > i, \\ 0 & \text{otherwise,} \end{cases}$$

where $|x|$ denotes the weight of the vector $x \in \{0,1\}^k$, that is, the number of 1s in it. We reserve the notation $\mathrm{MAJ}_k(x)$ for the function $\mathrm{THR}_k^{k/2}(x)$.

We can substitute each comparator in the network by k majority functions. Note that by adding several constants 0 or 1 as inputs to the gate we can convert any THR_k^i function into $\mathrm{MAJ}_{k'}$ with $k' \leqslant 2k$.

Now, it remains to observe that the median bit in the output array computes exactly MAJ_n. Thus, as a result, we get the following lemma.

Lemma 5. *Any k-sorting network of depth d and size s can be effectively converted into a circuit of depth d and size ks consisting of MAJ_{2k} gates and computing majority. In the case $k = 2$, we get just a monotone circuit consisting of* AND *and* OR.

2.3 Approximate Majority

By ε-approximate majority function $\mathrm{MAJ}_n^\varepsilon$ we denote the partial function that outputs MAJ_n of its input but is defined only on the inputs where the fraction of ones in it is bounded away by ε from $1/2$.

We need the following known result.

Theorem 6 ([33])**.** *For any constant $\varepsilon > 0$, one can compute $\mathrm{MAJ}_n^\varepsilon$ explicitly by a monotone circuit of size $poly(n)$ and depth $O(\log n)$.*

2.4 t-Wise Independent Hash Functions

We need the notion of t-wise independent hash functions.

Definition 7. For integers N and t such that $t \leqslant N$, a family of function $\mathscr{H} = \{h\colon [N] \to [N]\}$ is t-wise independent if for all distinct $x_1, \ldots, x_t \in [N]$ the random variables $h(x_1), \ldots, h(x_t)$ are independent and uniformly distributed in $[N]$, when $h \in \mathscr{H}$ is drawn uniformly.

Theorem 8 ([31])**.** *For every integer n and t such that $t \leqslant 2^n$ there is a family of k-wise independent functions $\mathscr{H} = \{h\colon \{0,1\}^n \to \{0,1\}^n\}$ such that choosing a random function from $\mathscr{H}$ takes nt random bits and evaluating a function from $\mathscr{H}$ takes time $poly(n,t)$.*

Theorem 9 ([6]). *Let X be the average of N t-wise independent random variables $X_1, \ldots, X_N \in [0, 1]$ for even t. Then for any $\varepsilon > 0$ we have*

$$\Pr\left[|X - \mathrm{E}[X]| \geqslant \varepsilon\right] \leqslant 1.1 \left(\frac{t}{N\varepsilon^2}\right)^{t/2}.$$

3 Sub-log-squared Circuit for Majority

In this section, we provide a proof of Theorem 1.

Our goal is to compute MAJ_n by an explicit circuit of polynomial size and $o(\log^2 n)$ depth. We assume for convenience that n is odd (for even n we can consider a circuit for $n + 1$ and substitute one variable by a constant). We start with some inferior circuit and perform several operations that allow us to gradually improve the parameters. However, on our way, we need to consider randomized circuits as well, and apart from size and depth, we will also be interested in the number of random bits and the error probability. More specifically, a circuit is an (s, d, r, err)-circuit for majority if its size is at most 2^s, depth is at most d, we can construct a circuit using at most r random bits and the error probability on each input is at most $2^{-\mathsf{err}}$. Here all parameters are functions in the number of inputs n (we write $\mathsf{err} = \infty$ when the circuit is correct with probability 1). All circuits we are going to consider are effectively constructible: there is an algorithm that given the values of random bits constructs a circuit in polynomial time in the size of the circuit.

Given a circuit with some parameters, we will use two operations to obtain new circuits. We are introducing these operations in the next two lemmas. Their effect on the circuit is summarized in the table below.

Initial circuit	Brute-force derandomization	Downward self-reduction
$s(n)$	$O(s(n) + r(n))$	$O(\log n) + s(2k)$
$d(n)$	$d(n) + O(r(n))$	$O\left(\left(\frac{\log_2 n}{\log_2 k}\right)^2 d(2k)\right)$
$r(n)$	0	$r(2k)$
$\mathsf{err}(n)$	∞	$\mathsf{err}(2k) - O(\log n)$

Lemma 10 (Brute-force derandomization). *If there is an $(s, d, r, 2)$-circuit C, then there is an $(O(s + r), d + O(r), 0, \infty)$-circuit.*

8

This lemma allows us to get rid of randomness but increases the depth and the size of the circuit if r is large.

Proof. Consider a randomized circuit $C_y(x)$, where $x \in \{0,1\}^n$ is an input and $y \in \{0,1\}^r$ is the sequence of random bits. Assume $C_y(x)$ has the parameters, as in the statement of the lemma. Consider circuits $C_y(x)$ for all possible values of y and observe that for any x the fraction of circuits that output $\text{MAJ}_n(x)$ is at least $1 - 1/4 = 3/4$. Thus, if we feed $C_y(x)$ for all y into a circuit from Theorem 6 computing $\text{MAJ}_{2^r}^\varepsilon$, the output is exactly $\text{MAJ}_n(x)$.

The size of the resulting circuit is at most $2^r \cdot 2^s + \text{poly}(2^r)$, where the first term corresponds to computing $C_y(x)$ for all y and the second term corresponds to computing $\text{MAJ}_{2^r}^\varepsilon$. Thus, the size is $2^{O(s+r)}$. Since all $C_y(x)$ can be computed in parallel, the depth of the circuit is at most $d + O(r)$. The resulting circuit does not use random bits and is always correct. $\square$

Lemma 11 (Downward self-reduction)**.** *If there is an $(s(n), d(n), r(n), \mathsf{err}(n))$-circuit C, then for any $k < n$ there is an $(O(\log n) + s(2k), O(\log_k^2 n \cdot d(2k)), r(2k), \mathsf{err}(2k) - O(\log n))$-circuit.*

This operation increases the depth (if $d(n)$ is sub-log-squared), but allows to reduce other parameters.

Proof. Consider a k-sorting network of depth $O(\log_k^2 n)$, given by [25] or by our Theorem 2 (the latter allows only for limited values of k, but the values we will actually use in the construction below are within the limits). By Lemma 5 this network gives us a monotone circuit with the same parameters consisting of MAJ_{2k} gates computing MAJ_n, denote this circuit by $C(x)$, where $x \in \{0,1\}^n$.

Consider a $(s(2k), d(2k), r(2k), \mathsf{err}(2k))$-circuit C_y on k inputs, where $y \in \{0,1\}^{r(2k)}$. Fix y and substitute each MAJ_{2k} gate in C by C_y. Denote the resulting circuit by $D_y(x)$. This is a standard monotone Boolean circuit, its size is $\text{poly}(n) \cdot 2^{s(2k)}$, its depth is $O(\log_k^2 n \cdot d(2k))$ and the number of random bits is $r(2k)$.

It remains to show that the error probability is not too large. For this fix some input $x \in \{0,1\}^n$. Consider all MAJ_{2k} gates in $C(x)$ and denote their inputs when x is fed to C by $z^1, z^2, \ldots, z^t$. Here t is the size of C and is polynomial in n.

For each z^i the probability over random y that $C_y(z^i)$ computes $\text{MAJ}_{2k}(z^i)$ incorrectly is at most $2^{-\mathsf{err}(2k)}$. By union bound, with probability at least $1 - t2^{-\mathsf{err}(2k)}$ we have $C_y(z^i) = \text{MAJ}_k(z^i)$ for all i and thus

$D_y(x)$ computes $\mathrm{MAJ}_n(x)$ correctly. Thus, the probability of error of the resulting circuit is at most

$$t \cdot 2^{-\mathsf{err}(2k)} = 2^{-\mathsf{err}(2k)+O(\log n)}$$

□

Now we describe our starting circuit. Interestingly, it is constructed as a partial derandomization of Valiant's construction.

Lemma 12. *There is an explicit circuit for majority with parameters* $(O(\log n), O(\log n), O(\log^3 n), \Omega(\log^2 n))$.

We provide the proof of Lemma 12 in Section 3.1 below, but before that, we explain how to finish the construction of the desired circuit for MAJ_n.

Starting with the circuit provided by Lemma 12, we first apply downward self-reduction with the parameter k satisfying $\log k = C\sqrt{\log n}$ for some big enough constant $C > 0$, then we apply brute-force derandomization, and then we apply downward self-reduction again with k satisfying $\log k = \log^{2/3} n$. We summarize the changes in the parameters after each step in the table below.

	Initial circuit	Step 1	Step 2	Step 3
		Self-reduction with $\log k = \sqrt{\log n}$	Brute-force derandomization	Self-reduction with $\log k = \log^{2/3} n$
$s(n)$	$O(\log n)$	$O(\log n)$	$O(\log^{3/2} n)$	$O(\log n)$
$d(n)$	$O(\log n)$	$O(\log^{3/2} n)$	$O(\log^{3/2} n)$	$O(\log^{5/3} n)$
$r(n)$	$O(\log^3 n)$	$O(\log^{3/2} n)$	0	0
$\mathsf{err}(n)$	$\Omega(\log^2 n)$	$\Omega(\log n)$	∞	∞

Remark 13. *Note that with the two operations in hand, there are not that many options to apply them to a given initial construction. It is not hard to check that applying downward self-reduction two times in a row is not better than applying it once with the appropriate value of k. Clearly, there is no need to apply the derandomization step twice. From this, it is not hard to*

10

see that our sequence of operations is actually optimal. Once the optimal sequence of operations is established, it is not hard to check that our choice of parameters in downward self-reductions is optimal as well.

3.1 Proof of Lemma 12

In this subsection, we are going to prove Lemma 12. The high-level idea is to partially derandomize Valiant's construction. To make the presentation self-contained we first recall the idea behind this construction.

Suppose we have independent random bits x, y, z that are equal to 1 with probability p and consider $\mathrm{MAJ}_3(x, y, z)$. It is not hard to see that it outputs 1 with probability $f(p) = p^3 + 3p^2(1-p)$. Consider $p = \frac{1}{2} + \varepsilon$ for some $\varepsilon > 0$ and denote $\varepsilon' = f(p) - \frac{1}{2}$. Then

$$\varepsilon' = f(p) - \frac{1}{2} = f(p) - f(\frac{1}{2}) = f'(\alpha)(p - \frac{1}{2}) = f'(\alpha)\varepsilon$$

for some $\alpha \in [\frac{1}{2}, p]$. Note that $f'(p) = 6p - 6p^2 = 6p(1-p)$. It is easy to see that for $\alpha \in [\frac{1}{2}, \frac{2}{3}]$ we have $f'(\alpha) \geqslant \frac{4}{3}$. Thus, for $\varepsilon \in [\frac{1}{2}, \frac{2}{3}]$ we have $\varepsilon' \geqslant \frac{4}{3}\varepsilon$.

Now, we can use this in the following way. Consider MAJ_n for odd n and consider its arbitrary input x. Without loss of generality, assume that $\mathrm{MAJ}_n(x) = 1$. If we draw one variable from x uniformly at random, it is equal to 1 with probability at least $\frac{1}{2} + \frac{1}{n}$. Consider a MAJ_3 gate and feed to it three independently and uniformly drawn input variables. By the analysis above the output of such a MAJ_3 gate is equal to 1 with probability at least $\frac{1}{2} + \frac{4}{3} \cdot \frac{1}{n}$. Now we can repeat this: consider three such MAJ_3 gates and feed their outputs to another MAJ_3 gates. The result is equal to 1 with probability $\frac{1}{2} + \left(\frac{4}{3}\right)^2 \frac{1}{n}$. After $O(\log n)$ many iterations, we get a $O(\log n)$-depth randomized circuit consisting of MAJ_3 gates that output the correct value with probability at least $\frac{2}{3}$. Valiant's argument further improves this probability, but we will not need this part of the argument.

The randomized circuit above uses too many random bits. Now we are going to modify the construction in a way, that uses randomness more efficiently. We will use some ideas from [9].

Construct the following circuit consisting of MAJ_3 gates. The circuit contains $\Theta(\log n)$ layers, each containing $N = n^3$ gates. The bottom layer consists of input variables, each repeated $\frac{N}{n} = n^2$ times (it is redundant to copy variables several times, we do this exclusively for the sake of uniformity of the construction). In other layers, each gate computes the MAJ_3 function of some gates from the previous layer. To assign the inputs to each gate, for each layer j we draw three fresh (and independent of each other) t-wise

11

independent hash functions $f_j, g_j, h_j \colon [N] \to [N]$ for $t = \Theta(\log n)$. For a gate with number i in layer j we set its inputs to be gates with numbers $f_j(i)$, $g_j(i)$ and $h_j(i)$ in layer $(j-1)$.

Before we finish the construction of the circuit, let us analyze the current part. Consider some input $x \in \{0,1\}^n$, assume without loss of generality that $\mathrm{MAJ}_n(x) = 1$. Denote by $\frac{1}{2} + \varepsilon_i$ the fraction of gates on level i that output 1. For $i = 1$ we have $\varepsilon_1 \geqslant \frac{1}{n}$.

Each gate on level i receives three independent inputs from the previous level. Thus, the probability that it outputs 1 is at least $\frac{1}{2} + \frac{4}{3}\varepsilon_{i-1}$ (we have shown this above only for $\varepsilon_{i-1} \leqslant \frac{2}{3}$, but these values of ε_{i-1} are enough for our construction as well). Thus, the expected fraction of ones in level i is also at least $\frac{1}{2} + \frac{4}{3}\varepsilon_{i-1}$.

Now we would like to use concentration inequality to show that with high probability the fraction of correct values is not much smaller than its expectation. Note that the outputs of the gates on level i are t-wise independent.

Let $\varepsilon = \frac{1}{6n}$ and denote by X_i the output of i-th gate. Then by Theorem 9 we have

$$
\Pr\left[\left|\sum_i X_i/N - \left(\frac{1}{2} + (4/3)\varepsilon_{j-1}\right)\right| \geqslant \varepsilon\right] \leqslant 1.1\left(\frac{t}{N\varepsilon^2}\right)^{t/2} = 2^{-\Theta(\log^2 n)}.
$$

By union bound the probability that on each level $\varepsilon_j \geqslant \frac{4}{3}\varepsilon_{j-1} - \varepsilon$ is at least

$$
1 - O(\log n) \cdot 2^{-\Theta(\log^2 n)} = 1 - 2^{-\Theta(\log^2 n)}.
$$

Thus, we can show by induction on j that with probability at least $1 - 2^{\Theta(\log^2 n)}$ we have

$$
\varepsilon_j \geqslant \frac{4}{3}\varepsilon_{j-1} - \varepsilon \geqslant \frac{7}{6}\varepsilon_{j-1} + \frac{1}{6}\varepsilon_{j-1} - \frac{1}{6n} \geqslant \frac{7}{6}\varepsilon_{j-1},
$$

where in the last inequality we use that by induction hypothesis we have $\varepsilon_{j-1} \geqslant \left(\frac{7}{6}\right)^{j-1} \cdot \varepsilon_1 \geqslant \frac{1}{n}$.

Thus, just like in Valiant's argument, after $O(\log n)$ iterations, with porbability $1 - 2^{-\Theta(\log^2 n)}$, we have $\varepsilon_j \geqslant \frac{2}{3}$. At this point, it remains to apply to the last layer a circuit from Theorem 6.

It is easy to see that the size of the resulting circuit is $\mathrm{poly}(n)$, the depth is $O(\log n)$, error probability is $2^{-\Theta(\log^2 n)}$. As for the random bits, note that in the construction we need $O(\log n)$ t-wise independent hash functions from $[N]$ to $[N]$. By Theorem 8 there are families of such functions defined using $O(t \log N)$ random bits. In total we need

$$
O(\log n) \cdot O(t \log N) = O(\log^3 n)
$$

random bits.

This finishes the proof of Lemma 12.

Remark 14. *Instead of applying a circuit for Approximate Majority to the last layer, we could do the following: sample $m = O(\log^2 n)$ gates from the last layer uniformly at random and then compute the majority on these m gates using some simple circuit of depth $O(\log^2 m)$. By Chernoff's inequality, this adds at most $2^{-\Omega(m)} = 2^{-\Theta(\log^2 n)}$ to the error probability, and we need $O(\log^3 n)$ random bits. In turn, the increase in depth and size is negligible.*

4 k-Sorting Network Construction

4.1   Proof Strategy

Before we proceed to the proof we would like to illustrate the idea considering some specific value of k. For convenience, we assume that n is a cube of a natural number.

Lemma 15. *Assume that $n = t^3$ for natural t. Then there is a depth-4 k-sorting network with $k = 2t^2 = 2n^{2/3}$.*

We present the proof using a geometric interpretation of an input array as a three-dimensional cube. However, note that a similar result is implicit in [22] and it is essentially the same construction, just in different terms. We also note that it is known that this is the optimal (up to a constant factor) value of k for depth-4 sorting networks [11].

Proof. We represent entries of an input array as a 3-dimensional cube with the side t (see Figure 1a). We place the first t^2 entries of an array in the bottom layer of the cube, the next t^2 entries in the second layer of the cube and so on. In each layer the entries are positioned row by row.

To be more precise, assume that the array A is enumerated as $[a_1, \ldots, a_n]$. We reenumerate the same array as

$$[a_{111}, a_{112}, \ldots, a_{11t}, a_{121}, \ldots, a_{12t}, \ldots, a_{ttt}].$$

That is, entries of an array are enumerated by sequences $(x, y, z) \in \{1, \ldots, t\}^3$ in the lexicographic order. In Figure 1 a_{xyz} corresponds to a subcube with coordinates (x, y, z).

In the first layer of the sorting network we split the cube into vertical slices of width 1 and feed each slice to a t^2-comparator (see Figure 1b). To be more precise, for each $i = 1, \ldots, t$ we feed entries a_{xyi} for all x, y into

13

(a) Input array

(b) Step 1: cut the cube into vertical slices

(c) Step 2: cut the cube into vertical slices in the other direction

(d) Step 3: cut the cube into horizontal slices of width 2
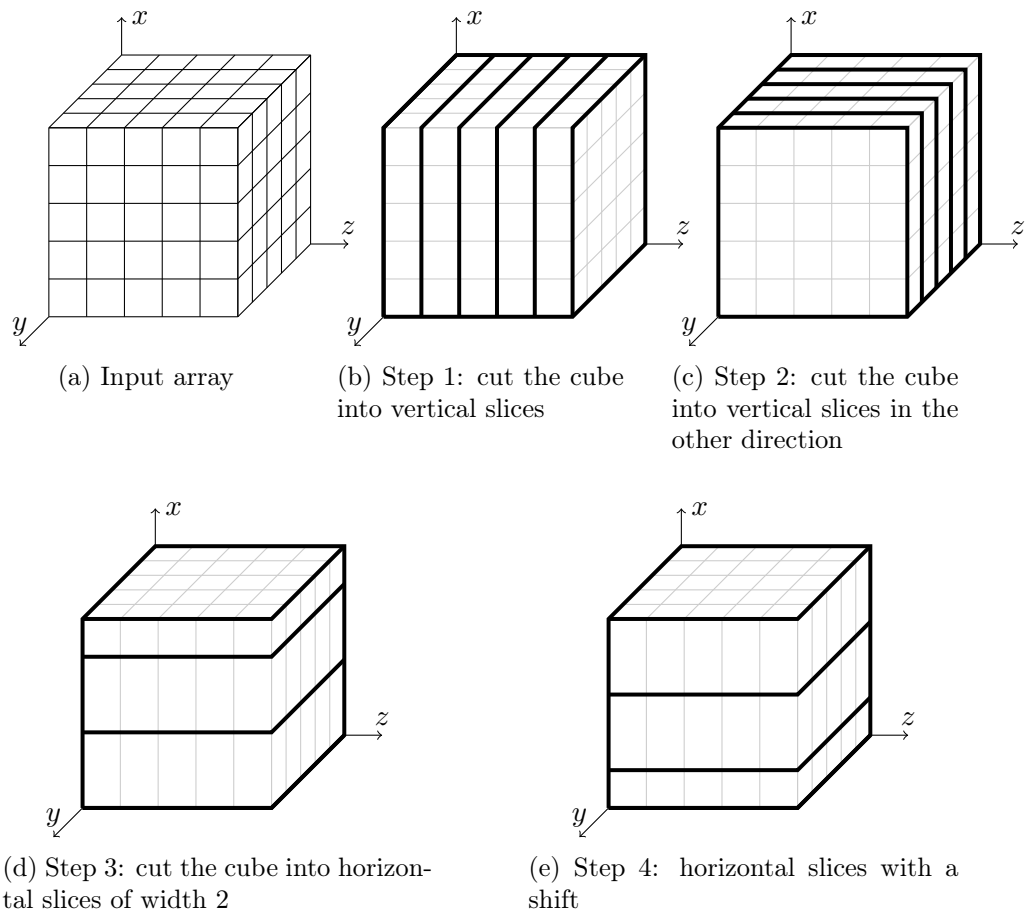
(e) Step 4: horizontal slices with a shift

Figure 1: Sorting network for $k = 2n^{2/3}$ (here $n = 125$, $k = 50$ and $t = 5$)

one comparator. On the second layer of the network we split the cube into vertical slices of width 1 in another direction and feed each slice to a t^2-comparator (see Figure 1c). In other works, for each $i = 1, \ldots, t$ we feed entries a_{xiz} for all x, z into one comparator. On the third layer we split the cube into horizontal slices of width 2 (for odd t the last slice is of width 1) and feed the slices to comparators of arity at most $2t^2$ (see Figure 1d). Finally, on the fourth layer of the network we split the cube into horizontal slices of width 2 again, but now the first slice is of width 1 (for even t the last slice is of width 1 as well). Thus, the slices on this layer are shifted compared to the previous one (see Figure 1e).

It remains to prove that this sorting network sorts correctly. Consider any input $x \in \{0,1\}^n$. Note that the cube consists of t^2 vertical columns with t entries in each column: each column A_{yz} is obtained by fixing y and z in a_{xyz} and considering all possible x. We are interested in the weight w_{yz} of each column, that is the number of 1s in it. For the input A the weights of the columns can be any numbers from 0 to t. Now consider the array after the first layer of the network. Note that now each vertical slice of the first layer of the network is sorted. This means that in each of these slices in the first several rows (from bottom to top) there are only 0s, then there might be a row containing both 0s and 1s, and then all remaining rows contain 1s. In particular, the weights of two columns in the same slice differ by at most 1.

Now consider the second layer of the network and consider two different slices $S_i = \{a_{x,i,z} \mid x, z \in [t]\}$ and $S_j = \{a_{x,j,z} \mid x, z \in [t]\}$. Note that each of them contains exactly one column from each slice of the first layer. We know that the weights of the columns in the same slice of the first layer differ by at most 1. Thus, in total, the number of 1s in two slices of the second layer differ by at most t. In other words, for each z the first slice contains the column A_{iz} and the second slice contains the column A_{jz}. We know that on the input of the second layer of the network $|w_{iz} - w_{jz}| \leqslant 1$. Thus,

$$|\sum_z w_{iz} - \sum_z w_{jz}| \leqslant t.$$

Denote by r_i the number of rows in slice S_i that consists of only 1s after the second layer of the network. We just showed that the slice S_i can have one more extra row of 1s, one less row of 1s or something in between. Overall, for the number r_j of rows consisting of 1s in S_j we have $|r_i - r_j| \leqslant 1$. As a result, the weights of columns in slices S_i and S_j can differ by at most 2. Since this is true for any i and j, we have that the weights of all columns in the cube after the second layer of the sorting network differ by at most 2.

To put it another way, there is a horizontal slice of width 2, such that below this slice we have only 0s and above this slice we have only 1s. Thus it remains to sort entries of this slice. Note that on layers 3 and 4 of the network there is a comparator that sorts exactly this slice. Note that by Lemma 4 all other comparators of layers 3 and 4 do not harm the sorting. $\square$

This argument can be extended to the cubes of arbitrary dimension d. More specifically, for $n = t^d$ and for $k = (d-1)t^{d-1}$ we can represent entries of an input array as a d-dimensional cube with side d, sort 'vertical' slices (we need to fix one of the coordinates in d-dimensional space as vertical) in all $d-1$ directions and then sort horizontal slices. This results into $(d-1)$ layers of the sorting network and for horizontal slices we need recursive calls for the arrays of size approximately $2dt^{d-1}$. Actually, it is expensive to make two recursive calls for horizontal layers, instead we use an additional trick to make just one recursive call.

Although our k-sorting network construction can be expressed in terms of high dimensional hypercubes, we prefer to give a more general exposition, using a concept of *s-sorted arrays*.

4.2 Merging s-Sorted Arrays

The following definition plays a key role in our sorting network construction.

Definition 16. A 0/1-array A of length n is *s-sorted* if there is an integer interval $I = \{i, \ldots, i+s-1\} \subseteq [n]$, such that $A[j] = 0$ for $j < i$ and $A[j] = 1$ for $j \geqslant i + s$. We call I *unsorted interval*.

As an immediate corollary of Lemma 4, we get the following.

Corollary 17. *Suppose a sorting network gets an s-sorted array with unsorted interval I. Then the output is also s-sorted with I as an unsorted interval.*

We give a construction of a depth-1 sorting network that "merges" p arrays of length n that are already s-sorted into one array which is $(sp + O(np^2/k))$-sorted, where k is the arity of the sorting network.

Lemma 18. *Assume that $k \geqslant tp$ for some integers t and p. Suppose we have p s-sorted arrays of size n each. Assume additionally that n is divisible by t. Then there is a depth-1 k-sorting network that merges these arrays into one array of size np that is $(sp + 2\frac{np}{t})$-sorted. If additionally we assume that s is divisible by n/t, then the resulting array is $(sp + \frac{np}{t})$-sorted.*

16

Proof. Represent each array as a table with $\frac{n}{t}$ columns and t rows. We assume the following ordering on the entries of this table: to compare two entries, we first compare the indices of their rows, and then the indices of their columns. Position the tables one under another in a unified table with tp rows. Note that $tp \leqslant k$ and apply k-comparator to each column in parallel. We claim that the resulting array in the large table is $(sp + 2\frac{np}{t})$-sorted.

To see that observe, that in each small table, an unsorted interval of length at most s occupies at most $\lceil \frac{st}{n} \rceil + 1$ rows (any other row either consists entirely of 0s or entirely of 1s). In the large table, this gives us at most $p\left(\lceil \frac{st}{n} \rceil + 1\right)$ non-constant rows. After sorting each column individually, 0-rows will move to the top, 1-rows will move to the bottom and all other $p\left(\lceil \frac{st}{n} \rceil + 1\right)$ rows will be in between on them. They constitute an unsorted interval and the size of it is at most

$$\frac{n}{t} \cdot p\left(\left\lceil \frac{st}{n} \right\rceil + 1\right).$$

For general s we can upper bound this as follows

$$\frac{n}{t} \cdot p\left(\left\lceil \frac{st}{n} \right\rceil + 1\right) \leqslant \frac{n}{t} \cdot p\left(\frac{st}{n} + 2\right) = sp + 2\frac{np}{t}.$$

If s is divisible by n/t, note that we can just drop the rounding operation and the size of an unsorted interval is at most

$$\frac{n}{t} \cdot p\left(\left\lceil \frac{st}{n} \right\rceil + 1\right) = \frac{n}{t} \cdot p\left(\frac{st}{n} + 1\right) = sp + \frac{np}{t}.$$

$\square$

Applying previous lemma several times we get the following.

Lemma 19. *Consider arbitrary n and k and denote $t = \lfloor \sqrt{k} \rfloor$. Then there is a k-sorting network of depth $\lceil \log_t n \rceil - 1$ that on any input outputs an s-sorted array for $s \leqslant \frac{2\lceil \log_t n \rceil n}{t}$.*

Proof. Denote $d = \lceil \log_t n \rceil$ and observe that $n \leqslant t^d$. Introduce the following notation:

$$n_i = \begin{cases} t^{i+1} & \text{for } i = 1, \ldots, d-2, \\ t^{d-1}p & \text{for } i = d-1, \end{cases}$$

where p is such that $t^{d-1}(p-1) < n \leqslant t^{d-1}p$. In particular, since $p \geqslant 2$, we have $p - 1 \geqslant p/2$ and

$$n > t^{d-1}(p-1) \geqslant t^{d-1}p/2.$$

For the convenience of presentation, we add $t^{d-1}p - n$ dummy inputs equal to 1 to the end of the array to make the size of the input to be equal to $t^{d-1}p$. By Lemma 4 these inputs will never change their position and can be removed from the sorting network.

We start with an unsorted array as an input and repeatedly apply Lemma 18 to get the array consisting of blocks that are s-sorted for some s. More specifically, after level i of the network we get the blocks of size n_i that are s_i sorted for

$$s_i = \begin{cases} (i-1)t^i & \text{for } i = 1, \ldots, d-2, \\ (d-2)t^{d-2}p & \text{for } i = d-1. \end{cases}$$

On the first step we split the input into blocks of size t^2 and apply comparators to them, the resulting blocks are 0-sorted.

On the i-th step for $i = 2, \ldots, d-1$ we already have blocks of size $n_{i-1} = t^i$ from the previous step that are s_{i-1}-sorted for $s_{i-1} = (i-2)t^{i-1}$. Note that $n_{i-1} = t^i$ is divisible by t and s_{i-1} is divisible by $n_i/t = t^{i-1}$. We apply Lemma 18 and for $i < d-1$ get blocks of size $n_{i-1}t = n_i$ that are s-sorted for $s = s_{i-1}t + n_{i-1} = (i-1)t^i$. For $i = d-1$ we have just p subarrays to merge and after the step we get the whole array of size $t^{d-1}p$ that is s-sorted for $s = (d-3)t^{d-2}p + \frac{t^{d-1}p}{t} = (d-2)t^{d-2}p$.

Finally, observe that

$$s \leqslant (d-2)t^{d-2}p \leqslant d\frac{2n}{t}$$

as desired. □

4.3   Computing Majority

Before constructing a sorting network we solve a simpler task of computing majority function.

Theorem 20. *For any n and for any k such that $\log k = \omega(\log \log n)$ (or, to put it differently, k is growing faster than any $\mathsf{polylog}(n)$), there exists a MAJ_k-circuit for MAJ_n of depth at most $(2 + o(1))\log_k^2 n$.*

The rest of the section is devoted to the proof of Theorem 20.

First observe that to compute MAJ_n correctly by a monotone circuit it is enough to compute it correctly on minterm and maxterm inputs: the computation on other inputs follows by monotonicity. Thus, we can assume in our construction that the input contains almost the same number of 0s and

1s. We will construct a sorting network that sorts all such inputs correctly. From the sorting network we get the circuit of the same depth.

Suppose we need to sort an array of size n with approximately the same number of 0s and 1s. We apply Lemma 19 to the array. This results in a Y-sorted array for $Y = \frac{2\lceil \log_t n \rceil n}{t}$ for $t = \lfloor \sqrt{k} \rfloor$. Since the number of 0s and 1s in the array is approximately equal, the smallest $\frac{n}{2} - Y$ and the largest $\frac{n}{2} - Y$ elements are sorted correctly (otherwise, the length of unsorted interval is larger than Y). Thus, it remains to sort specific interval of length $2Y$ and we can do it recursively.

Overall, we get the following recursive relation.

$$T(n) \leqslant \lceil \log_t n \rceil - 1 + T\left(2Y\right) \leqslant \log_t n + T\left(\frac{4\lceil \log_t n \rceil n}{t}\right).$$

To solve this recursive relation we use the following lemma.

Lemma 21. *Assume that $\log k = \omega(\log \log n)$. Suppose that $T(n) = const$ for n up to some constant and*

$$T(n) \leqslant 2 \log_k n + C + T\left(\left\lceil \frac{D(\log_k n)n}{\sqrt{k}} \right\rceil\right)$$

for some constants C and $D > 0$. Then $T(n) \leqslant (2 + o(1)) \log_k^2 n$.

Proof. To simplify the presentation, we ignore rounding of the argument of T first, and address it later. Denote $\alpha = \frac{\sqrt{k}}{D \log_k n}$.

We have

$$T(n) \leqslant 2 \log_k n + C + T\left(\frac{n}{\alpha}\right)$$
$$\leqslant 2 \log_k n + C + 2 \log_k \frac{n}{\alpha} + C + T\left(\frac{n}{\alpha^2}\right)$$
$$\leqslant 2 \sum_{i=0}^{\log_\alpha n} \left(\log_k \frac{n}{\alpha^i} + C\right)$$
$$= 2\left(\log_k n + (\log_k n - \log_k \alpha) + (\log_k n - 2 \log_k \alpha) + \ldots + 0\right) + 2C \log_\alpha n$$
$$\leqslant 2 \frac{\log_k n}{\log_k \alpha} \frac{\log_k n}{2} + 2C \log_\alpha n = \log_k^2 n \log_\alpha k + 2C \log_\alpha n.$$

It is easy to see that the term $2C \log_\alpha n$ is negligible, since $\alpha \gg k^{1/3}$.

We analyze $\log_\alpha k$ factor separately:

$$\log_\alpha k = \log_{\frac{\sqrt{k}}{D \log_k n}} k = \frac{\log_2 k}{\log_2 \frac{\sqrt{k}}{D \log_k n}} \leqslant \frac{\log_2 k}{\frac{1}{2} \log_2 k - D - \log_2 \log_k n}$$
$$= \frac{\log_2 k}{\frac{1}{2} \log_2 k - D - \log_2 \log_2 n + \log_2 \log_2 k}.$$

For $\log k = \omega(\log \log n)$ this term is $2 + o(1)$ and we have

$$T(n) \leqslant (2 + o(1)) \log_k^2 n.$$

To address the rounding operation, note that $\lceil \frac{n}{\alpha} \rceil \leqslant \frac{n}{\alpha} + 1 \leqslant \frac{2n}{\alpha}$ for $\frac{n}{\alpha} \geqslant 1$. Thus, in the presence of rounding we will have $\sum_i \log_k \frac{2n}{\alpha}$ in the calculation above instead of $\sum_i \log_k \frac{n}{\alpha}$. This amounts to substituting D by $2D$ and does not change the result of the calculation since D is an arbitrary constant. □

4.4 Constructing Sorting Network

In this section, we finish the proof of Theorem 2.

We adopt the same strategy as for the computation of majority. More specifically, we apply Lemma 19 recursively to get s-sorted array for smaller and smaller s. However, now our task is more tricky. In the proof of Theorem 20 when we get to an s-sorted array we know exactly where the unsorted interval is located (in the middle of the array). However, now we need to sort arbitrary input arrays and an unsorted interval can be anywhere.

We construct the network recursively. We assume that at the beginning of each step, we have an s-sorted array (at the beginning of the process $s = n$). Denote the unsorted interval by A, $|A| \leqslant s$. Split the array into consecutive blocks $B_1, \ldots, B_p$ of size s (the last block B_p) might be smaller.

The recursive step consists of two stages. In the first stage, we split the array into blocks $B_1 \cup B_2$, $B_3, \cup B_4$, and so on, each block of size $2s$ (one last block might be smaller). In the second stage, we split the array into blocks B_1, $B_2 \cup B_3$, $B_4 \cup B_5$, and so on (again the last block might be smaller than $2s$). Before describing each of the stages, observe that either in the first stage or in the second stage (or in both) the interval A falls completely into one of the blocks. Indeed, A can intersect with at most two consecutive blocks B_i, B_{i+1} and in one of the stages, they form a single block.

In the first stage, we apply Lemma 19 to each of the blocks $B_1 \cup B_2, B_3, \cup B_4, \ldots$ separately. As a result, each block is s'-sorted for $s' \leqslant$

$\frac{4\lceil\log_{\lfloor\sqrt{k}\rfloor} n\rceil s}{\lfloor\sqrt{k}\rfloor}$. Moreover, if the block consisted of only 0s and 1s, then it does not change.

If A is contained in one of the blocks of the first stage, we are already done: there is only one initially unsorted block that by Lemma 19 after the step is s'-sorted. By Corollary 17 this property remains true after the additional comparators we apply for the other case.

If A is split between two blocks of the first stage, then after the stage we have two consecutive unsorted blocks, each of them is s'-sorted. Denote unsorted parts by C_1, C_2. Note that by Corollary 17 $C_1, C_2 \subseteq A$ and thus, C_1 and C_2 fall into one block of the second stage. It is tempting to apply Lemma 19 to the blocks of the second stage as well. However, this application is too expensive and will not result in the desired bound.

Instead we do the following. We represent each block of the second stage (of size at most $2s$) as a table with $p = \lceil 2s/k \rceil$ columns and k rows, filled in row by row from top to bottom. For convenience, if the last row is not complete, add dummy variables equal to 1 to complete the row.

Each of the intervals C_1, C_2 occupy at most $\lceil s'/p \rceil + 1$ rows. There might be another row that contain a switch between blocks B_i and B_{i+1}. Each other row consist either entirely of 0s, or entirely of 1s. Denote the number of all 0 rows by a and the number of all 1 rows by b.

We apply a comparator to each column separately. As a result, each column will contain a zeros in the beginning, b ones in the end and some part in between. The number of rows in the middle part is at most $2\lceil s'/p \rceil + 3$. The number of entries in these rows is at most

$$s'' = p(2\lceil s'/p \rceil + 3) \leqslant 2s' + 5p \leqslant 3s'$$

for large enough input size. Thus, after the second stage we get s''-sorted array and we are done with the recursion step.

Thus, we get that $s'' \leqslant 12\frac{\lceil\log_t n\rceil s}{t}$ and we get the following recursive relation

$$T(n) \leqslant \log_t n + T\left(\frac{12\lceil\log_t n\rceil n}{t}\right).$$

We apply Lemma 21 again to get $T(n) \leqslant (2 + o(1))\log_k^2 n$.

This finishes the proof of Theorem 2.

4.5   Other Applications

In this section we give two more examples of results that follow from our construction.

Lemma 22. *There is a* MAJ_k*-circuit of depth* 4 *computing* MAJ_n *for* $k = O(n^{3/5})$.

Proof. Denote $r = \lceil n^{1/5} \rceil$. For simplicity we pad the input with constants 0 and 1 to make the size of the array r^5 without changing the output of majority. We will use k-sorters for $k = 4r^3$.

As in the proof of Theorem 20 it is enough to compute MAJ_n on minterms and maxterms, thus we can assume that there are approximately equal number of 0s and 1s in the input.

We will build a k-sorting network and the existence of the circuit follows. On the first layer of the network we split the input into blocks of size r^3 and sort them. On the second layer we use Lemma 18 with $p = r$ and $t = r^2$. As a result we get blocks of size r^4 that are r^2-sorted. On the third level we apply Lemma 18 again with the same values of p and t. As a result we have that the whole input is now $2r^3$-sorted. On the last layer of the network just as in the proof of Theorem 20 we apply $4r^3$-comparator to the middle of the array. $\qquad\square$

In [18] Knuth posed a problem of constructing a minimal depth k-sorting network for the input of size k^2. Parker and Parbery [25] gave a construction of depth 9. Here we slightly improve on this at the cost of using comparators of size $O(k)$.

Lemma 23. *There is a* k*-sorting network of depth* 8 *that sorts an array of size* n *with* $k = O(n^{1/2})$.

Proof. As usual pad an array with constants to make $n = r^4$ for some integer r. Thus $k = O(r^2)$.

We follow the same strategy as in Section 4.4. First we apply Lemma 19 that uses three layers of network and results in a s-sorted array for $s = O(r^3)$. Then, we apply Lemma 19 again to the blocks of size $O(r^3)$ to get a network of depth 2 that results in each block being $O(r^2)$-sorted. Then we apply one more layer to merge unsorted intervals in different blocks to get the array that is $O(r^2)$-sorted. Finally, we again split the array into blocks, this time of size $O(r^2)$ to complete the sorting using two layers. In total we use $3 + 2 + 1 + 2 = 8$ layers. $\qquad\square$

5 Conclusion

The obvious open problems are to come up with explicit constructions of sorting networks and monotone circuits for majority of smaller depth. One

specific problem, is to extend our $O(\log^{5/3} n)$ construction to sorting networks. The obstacle that we encountered is that there is no randomized construction of low depth sorting network that we can use as a start. Another interesting question is to extend our $O(\log^{5/3} n)$ construction to get a MAJ_k-circuit for MAJ_n of depth $O(\log_k^{5/3} n)$. Such a construction can be used instead of $O(\log_k^2 n)$-depth circuit in downward self-reduction to further improve the upper bound. Again the obvious obstacle is that it is not clear how to get a starting construction.

References

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

[2] Kazuyuki Amano and Masafumi Yoshida. Depth two $(n-2)$-majority circuits for n-majority. Preprint, 2017.

[3] S.W.A.H. Baddar and K.E. Batcher. *Designing Sorting Networks: A New Paradigm*. SpringerLink : Bücher. Springer New York, 2012.

[4] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[5] Richard Beigel and John Gill. Sorting n objects with a k-sorter. *IEEE Trans. Computers*, 39(5):714–716, 1990.

[6] Mihir Bellare and John Rompel. Randomness-efficient oblivious sampling. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 276–287. IEEE Computer Society, 1994.

[7] Daniel Bundala and Jakub Zavodny. Optimal sorting networks. In Adrian-Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2014.

[8] V. Chvátal. Lecture notes on the new aks sorting network. Technical report, 1992.

[9] Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron D. Rothblum. Efficient multiparty protocols via log-depth threshold formulae - (extended abstract). In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2013.

[10] Robert Cypher and Jorge L. C. Sanz. Cubesort: A parallel algorithm for sorting N data items with s-sorters. *J. Algorithms*, 13(2):211–234, 1992.

[11] Natalia Dobrokhotova-Maikova, Alexander Kozachinskiy, and Vladimir V. Podolskii. Constant-depth sorting networks. In Yael Tauman Kalai, editor, *14th Innovations in Theoretical Computer Science Conference, ITCS 2023, January 10-13, 2023, MIT, Cambridge, Massachusetts, USA*, volume 251 of *LIPIcs*, pages 43:1–43:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[12] Christian Engels, Mohit Garg, Kazuhisa Makino, and Anup Rao. On expressing majority as a majority of majorities. *SIAM J. Discret. Math.*, 34(1):730–741, 2020.

[13] Qingshi Gao and Zhiyong Liu. Sloping-and-shaking. *Science in China Series E: Technological Sciences*, 40(3):225–234, 1997.

[14] Shlomo Hoory, Avner Magen, and Toniann Pitassi. Monotone circuits for the majority function. In *Proceedings of the 9th international conference on Approximation Algorithms for Combinatorial Optimization Problems, and 10th international conference on Randomization and Computation*, pages 410–425, 2006.

[15] Pavel Hrubes, Sivaramakrishnan Natarajan Ramamoorthy, Anup Rao, and Amir Yehudayoff. Lower bounds on balancing sets and depth-2 threshold circuits. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 72:1–72:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[16] Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012.

[17] Nabil Kahalé, Frank Thomson Leighton, Yuan Ma, C. Greg Plaxton, Torsten Suel, and Endre Szemerédi. Lower bounds for sorting networks. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 437–446. ACM, 1995.

[18] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998.

[19] Yu. A. Kombarov. On depth two circuits for the majority function. In *Proceedings of Problems in theoretical cybernetics*, pages 129–132. Max Press, 2017.

[20] Alexander S. Kulikov and Vladimir V. Podolskii. Computing majority by constant depth majority circuits with low fan-in gates. *Theory Comput. Syst.*, 63(5):956–986, 2019.

[21] De-Lei Lee and Kenneth E. Batcher. A multiway merge sorting network. *IEEE Trans. Parallel Distributed Syst.*, 6(2):211–215, 1995.

[22] Frank Thomson Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Computers*, 34(4):344–354, 1985.

[23] Toshio Nakatani, Shing-Tsaan Huang, Bruce W. Arden, and Satish K. Tripathi. K-way bitonic sort. *IEEE Trans. Computers*, 38(2):283–288, 1989.

[24] Ian Parberry. The pairwise sorting network. *Parallel Process. Lett.*, 2:205–211, 1992.

[25] Bruce Parker and Ian Parberry. Constructing sorting networks from k-sorters. *Inf. Process. Lett.*, 33(3):157–162, 1989.

[26] Michael S Paterson. Improved sorting networks with o (logn) depth. *Algorithmica*, 5(1):75–92, 1990.

[27] Gleb Posobin. Computing majority with low-fan-in majority queries. *CoRR*, abs/1711.10176, 2017.

[28] Joel Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, 2009.

[29] Feng Shi, Zhiyuan Yan, and Meghanad D. Wagh. An enhanced multiway sorting network based on n-sorters. In *2014 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2014, Atlanta, GA, USA, December 3-5, 2014*, pages 60–64. IEEE, 2014.

[30] S. S. Tseng and Richard C. T. Lee. A parallel sorting scheme whose basic operation sorts$N$ elements. *Int. J. Parallel Program.*, 14(6):455–467, 1985.

[31] Salil P. Vadhan. Pseudorandomness. *Found. Trends Theor. Comput. Sci.*, 7(1-3):1–336, 2012.

[32] Leslie G. Valiant. Short monotone formulae for the majority function. *J. Algorithms*, 5(3):363–366, 1984.

[33] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18:337–375, 2009.

[34] Andrew Chi-Chih Yao. Bounds on selection networks. *SIAM J. Comput.*, 9(3):566–582, 1980.

[35] Lijun Zhao, Zhiyong Liu, and Qingshi Gao. An efficient multiway merging algorithm. *Science in China Series E: Technological Sciences*, 41(5):543–551, 1998.