The Computational Complexity Column

by

Michal Koucký

Computer Science Institute, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

`koucky@iuuk.mff.cuni.cz`

`https://iuuk.mff.cuni.cz/~koucky/`

Sorting Short Integers: The Exposition

Michal Koucký[*]
Computer Science Institute
Charles University, Prague
koucky@iuuk.mff.cuni.cz

Abstract

This expository article reviews recent and some not so recent results on sorting integers in various models of computation: from word RAM to Boolean circuits, with the main focus on the latter. We hope that even seasoned researcher will find our perspective refreshing.

1 Sorting using comparisons

Sorting is one of the most ubiquitous and versatile algorithmic primitives. It is taught in introductory computer science courses around the globe. The classical algorithms such as Heapsort and Quicksort take time $O(n \log n)$ [13]. They are comparison based algorithms which means that they rely only on operations that move the input items around and compare two individual items which one is larger. There is a well known $\Omega(n \log n)$ lower bound on the number of comparisons needed by any comparison based algorithm to sort n *distinct* items. The lower bound is derived from the number of different permutations of the items.

It is often overlooked that the lower bound is merely linear in the bit-size of the input. Indeed, any *reasonable* binary encoding of a sequence of n distinct items will use $\Omega(n \log n)$ bits. So in terms of the bit-size of the input the lower bound is linear. This fact can be exemplified in the following scenario which seemingly breaks the lower bound: sorting n items from a domain K of size $2^k < n$. Using ordinary balanced search trees of depth $O(k)$ we can sort such a sequence using $O(nk)$ comparisons. (Each node in the tree contains a list of items of the same value.) For $k \in o(\log n)$, this gives sorting using $o(n \log n)$ comparisons. One can show using the standard argument that $\Omega(nk)$ comparisons are needed. The bit-size of the input is nk, so in this case sorting is linear in the bit-size of the input.

[*]Partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 19-27871X.

In a more general setting we have n items $x_1, x_2, \ldots, x_n$, each item x_i encoded using w_i bits by some prefix-free code. Encoding of the whole input sequence gives input of bit-size $m = w_1 + w_2 \cdots w_n$. We claim that the sequence can be sorted using $O(m)$ comparisons. If there are K distinct values in the input sequence, each appearing $q_1, q_2, \ldots, q_K$ times, sorting the sequence using Splay trees leads to $O(K + \sum_{j=1}^{K} q_j \log(n/q_j))$ comparisons [34]. It follows from standard information theoretic arguments on the entropy of the sequence that $m \geq \sum_{j=1}^{K} q_j \log(n/q_j)$ [14].

The use of a prefix-free encoding is natural. If the items are encoded individually by a code that can be uniquely decoded then there is a prefix-free encoding of the same efficiency [14]. For example a fixed-size encoding is prefix-free but some sequences might be encoded more efficiently: Consider a sequence consisting of $n - \sqrt{n}$ copies of 0 together with one copy of each number from 1 to $\sqrt{n}$ in arbitrary order. A fixed-size encoding would use $O(n \log n)$ bits but the sequence can be encoded by a prefix-free code using $O(n)$ bits in total, and we can sort the sequence using $O(n)$ comparisons.

Hence, sorting arbitrary sequence of n items can be done using a number of comparisons that is *linear in the bit-size* of the encoding of the sequence. This matches our intuition well — each comparison can extract one bit of entropy from the input and there are at most m bits of entropy to extract. Bit-size of the input will become an important measure later when we consider Boolean circuits.

1.1 Sorting networks

A special class of comparison based sorting algorithms is represented by sorting networks. A *sorting network* is a collection of n horizontal wires which carry the input values from left to right. At various places, two wires can be connected by a vertical *comparator* which switches the values carried along the wires so that the larger value continues on the lower wire and the smaller value on the upper wire. The comparators should be organized so that the values leave the network sorted top to bottom. See Fig. 1 for an example.

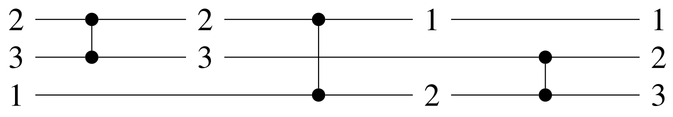

Figure 1: An example of a sorting network with three inputs (the horizontal lines), and three comparators (the vertical lines) [24].

The complexity of the network is the total number of comparators and the depth of the network is the maximum number of comparators on any path from left to right in the network. Comparator networks have been extensively studied and are used in various practical applications. Most notable constructions are the well-known construction of Batcher [8] of size $O(n \log^2 n)$ and depth $O(\log^2 n)$, and the famous construction of Ajtai, Komlós and Szemerédi [6] of size $O(n \log n)$ and depth $O(\log n)$. Up-to a constant factor the network of Ajtai, Komlós and Szemerédi (*AKS network*) is clearly optimal for sorting n distinct items by the above lower bound. However, it is also optimal for sorting input sequences consisting of only zeros and ones, as shown by the zero-one principle for sorting networks [25]: Any sorting network correctly sorting inputs of zeros and ones correctly sorts arbitrary inputs. Hence sorting items using sorting networks might be less efficient than sorting them by ordinary comparison based sorting algorithms.

Sorting networks are an example of an *oblivious* algorithm in which the sequence of performed operations does not depend on the actual input. Later we will see another example — Boolean circuits.

1.2 Beyond comparison

Beyond comparison based algorithms there is the well known Radixsort. This is an example of an algorithm that runs on the *word RAM*. In the word RAM model the memory is organized into cells of w bits each ($w \geq \log n$). At each time step the program executes some operation with a fixed number of memory cells. Which memory cells are involved in the operation can be determined by the content of special *address* cells of the machine. Usually we place a restriction on how complex each operation can be. Known algorithms usually involve relatively simple operations such as bit-wise Boolean operations, integer addition, multiplication, perhaps division. The input to the machine are n integers each of w bits presented in the first n memory cells. The output is expected to be there as well. When $w \in O(\log n)$, Radixsort can sort n integers in time $O(n)$, i.e., using $O(n)$ operations.

For large w, where $w \in \Omega(\log^2 n \log\log n)$, there are also algorithms for word RAM that run in time $O(n)$ [4, 9]. These algorithms are based on word level parallelism. They compress multiple input values into a single memory cell, and on the compressed input they run a comparison based sorting implemented in parallel. The parallel execution of the sorting algorithm provides the necessary speed-up to obtain a linear-time word RAM algorithm. The compression is usually quite involved and proceeds in several stages as the original input values have w bits.

For w in the range from $\omega(\log n)$ to $o(\log^2 n \log\log n)$ the fastest known algorithm of Han and Thorup [22] is randomized and operates in expected time $O(n\sqrt{\log\log n})$ or more precisely in time $O(n\sqrt{\log n \frac{w}{\log n}})$. The algorithm is a clever combination of several techniques used for sorting algorithms running in time $O(n \log\log n)$ [4, 18].

The first algorithm running in $O(n \log\log n)$ time was given by Andersson et al. [4]. It is obtained by iterating a linear time reduction of sorting n items of w bits each to sorting n items of $w/2$ bits each [4, 26]. The iteration stops when the integers reach bit-size $O(\log n)$ and we can sort them in linear time using e.g. Radixsort. The reduction splits the items into buckets based on their first $w/2$ most significant bits, and then sorts each bucket based on the $w/2$ least significant bits. The sorting of all the buckets can be done together by annotating each item by its bucket index.

Han [17, 18] developed a different approach and gave a linear time splitting procedure which takes n items of w bits and partitions them into sets $X_1, X_2, \ldots, X_t$ each of size at most $\sqrt{n}$ so that for any $i < j$, items in the set X_i are less or equal to the items in X_j. This procedure can be iterated for $O(\log\log n)$ steps to sort the integers completely. Han's splitting technique is a clever traversal of the trie built from the binary representations of the integers. The traversal is top to bottom and it iteratively splits the formed buckets into smaller ones. The splitting is done using hashing of prefixes into hash values of varying sizes depending on the entropy of the emerging buckets. The splitting is done for many items in parallel using word level parallelism which provides enough speed-up to finish the whole iterative process in linear time.

If there were a technique that would split the integers into sets $X_1, X_2, \ldots, X_t$ of size $2^{\log^{1-\epsilon} n}$ for some $\epsilon > 0$, one could sort in linear time.

Another algorithm running in time $O(n \log\log n)$ is presented in [27]. This algorithm selects a random subset of the input items of size $O(n/\log n)$ and sorts them. For each input item it then finds the proper interval in the sorted sample where the item belongs to. Finding the interval can be done using binary search on the longest common prefix shared with any of the sampled items in time $O(\log\log n)$. This creates small buckets corresponding to different intervals of the sample. Sorting each bucket gives a sorting algorithm running in time $O(n \log\log n)$.

1.3 Turing machines

Perhaps the most fundamental model of computation beside word RAM are Turing machines. Sorting n integers of w bits each can be done using comparison based sorting algorithm such as Quicksort on two-tape Turing machine in time $O(nw \log n)$. In the same time one can also implement Mergesort on three-tape Turing machine. On a single-tape Turing machine one cannot sort faster than $\Omega(n^2/\log^2 n)$ as otherwise one would break the quadratic lower bound on recognizing palindromes by one-tape Turing machines of Hennie [20].

For $w \leq \log n$, one can use binary Radixsort to sort integers on two-tape Turing machines in time $O(nw^2)$. (Splitting a list of integers into two based on a particular bit can be done by passing over the list twice.) It is believed that sorting on two-tape Turing machines requires time $\omega(nw)$. For restricted classes of algorithms this was proven by Stoss [35] and Paul [31].

2 Sorting by Boolean Circuits

Our main focus for the rest of the article is sorting using Boolean circuits. A *Boolean circuit* is a directed acyclic graph in which each node (*gate*) has in-degree at most two. Each node of in-degree zero is an *input gate* and it is labeled by one of the input bits $y_1, \dots, y_m$. Each node of in-degree one is labeled by a Boolean negation, and each node of in-degree two is labeled either by AND or OR. (The *in-degree* of a node is the number of its incoming edges and the *out-degree* is the number of its out-going edges.) On input $y \in \{0,1\}^m$ the circuit is evaluated by assigning values to gates and edges as follows: each edge of the circuit receives the value of its starting node, each input gate labeled by an input bit y_i is given the value of y_i, and each node labeled by a Boolean function g is assigned the value of g applied on the values of edges incoming to the node. The output of the circuit is given by the values of designated nodes.

The size of the circuit is the number of its gates and its depth is the length of the longest path from an input gate to some output gate. We want to minimize circuit size and depth. For more background on circuits see e.g. [23].

We will be interested in designing Boolean circuits computing the sorting function $\mathrm{SORT}_{n,w} : \{0,1\}^{nw} \to \{0,1\}^{nw}$ which takes as its input n integers, each encoded in binary using w bits, and outputs the same set of integers but sorted according to the numerical order. A variant of this problem is a partial sorting function according to k bits $\mathrm{SORT}_{n,w,k} : \{0,1\}^{nw} \to \{0,1\}^{nw}$ which takes the same input as $\mathrm{SORT}_{n,w}$ but outputs the set sorted according to the first k bits of the integers. Integers with the same value of the first k bits can appear in arbitrary order. Alternatively, one can think of a partial sort as sorting (key, value) pairs according to k-bit keys with values being $w - k$ bits long. Clearly, $\mathrm{SORT}_{n,w,w} = \mathrm{SORT}_{n,w}$. Our goal is to design circuits computing those functions. *Sorting circuits* will refer to such circuits.

Circuits are an oblivious model of computation as the sequence of performed operations does not depend on the actual input. One can build a sorting circuit from a sorting network by implementing each comparator by a small Boolean circuit. Indeed, it is fairly easy to build a circuit of size $O(w)$ and depth $O(\log w)$ that compares two w-bit integers and outputs them in a sorted order. Replacing each comparator in the AKS sorting network by a copy of such a circuit and connecting the wires appropriately one can get a circuit of size $O(nw \log n)$ and depth $O(\log n \log w)$ that sorts n integers of w bits each. We will refer to this circuit the *AKS sorting circuit.*

It was raised by Asharov et al. [7] whether one can design smaller circuits when w is small compared to $\log n$. Because of the zero-one law for sorting networks this will necessarily require a different technique than used by sorting networks. For every $\varepsilon > 0$, Asharov et al. [7] give a construction of circuits of size $O(nw^2(1 + \log^* n - \log^* w)^{2+\varepsilon})$ and polynomial depth sorting n integers of w bits each.

Their construction relies on a circuit for $\mathrm{SORT}_{n,w,1}$ which partially sorts the input according to a single bit. Using such circuits one can sort the integers completely bit by bit starting from the most significant bit. This recursive approach requires certain care. We describe next a different approach which gives a circuit of size $O(nw^2)$ and depth $O(\log n + w \log w)$.

2.1 Fast counting

Here we describe an approach to sorting based on Counting Sort from [24] which first counts the number of occurrences (*frequency*) of each possible integer value and then reconstructs the sorted integer sequence corresponding to the frequencies. This will give a sorting circuit of size $O(nw^2)$ and depth $O(\log n + w \log w)$.

Slow Counting. Let $W = 2^w$, where $W < n^{1/20}$. For a given value $y \in \{0, \dots, W-1\}$ we can determine the frequency of y among $x_1, \dots, x_n$ using a circuit of size $O(nw)$ and of depth $O(\log n + \log w)$. This is done by comparing y with each x_i for equality, and then summing up the resulting indicator vector. Comparing two w-bit strings can be done using a circuit of size $O(w)$ and of depth $O(\log w)$. Summing up n bits can be done using a circuit of size $O(n)$ and of depth $O(\log n)$. Hence we can calculate the frequency of all the values from 0 to $W - 1$ using a circuit of size $O(Wnw)$ and of depth $O(\log n + \log w)$. However, such a circuit is too big so we proceed differently.

Fast Counting. We divide the integer sequence into blocks of size W^8, and we sort each of the blocks by the AKS sorting circuit of size $O(W^8 w \log W^8)$ and of depth $O(\log W^8 \log w)$. Hence the total size of this stage of the algorithm is $O((n/W^8) \cdot W^8 w \log W^8) = O(nw^2)$ and the depth is $O(w \log w)$. This fits within our budget.

Now we partition each block into *parts* of W^4 integers. There are only W distinct integer values so except for W parts, integers in each part are the same. Any such part is called *monochromatic*. Since each part is sorted we can easily check whether it is monochromatic by comparing its first and last item.

At this stage it is also relatively inexpensive to compute the frequency of all the items in monochromatic parts: For each value, compute the number of monochromatic parts containing it, and multiply the number by W^4. Multiplication by W^4 only requires to pad each count by $4w$ zeros on the right. This will use a circuit of total size $O((n/W^4) \cdot W \cdot w) \subseteq O(n)$ and depth $O(\log(n/W^4)) \subseteq O(\log n)$ which is negligible for us.

It remains to count the number of items of each value in non-monochromatic parts. In each block we replace all the values in monochromatic parts by 0 and we sort each block again using the AKS sorting circuit. Except for the value 0, we can count the frequency of each value in non-monochromatic parts by counting its frequency in the last W^5 items in all the blocks. This can be done using a circuit of size $O((n/W^8) \cdot W^5 \cdot wW) \subseteq O(n)$ and of depth $O(\log n)$. To properly count the frequency of the value 0, we count zeros only in the first non-monochromatic part of each block. Those zeros correspond to the original zeros in the first non-monochromatic part of each originally sorted block. Counting those zeros can be done easily using the same technique as for the other values. Hence, it does not increase the circuit size beyond our budget.

At this point we have two frequency vectors, one counting all the values in monochromatic parts and one counting them in non-monochromatic parts. We can add them point-wise to obtain a vector of overall frequencies. This requires a circuit of a negligible size and depth.

Decompression. Decompression of the frequency vector can be done in a mirror fashion. We decompose the output positions into blocks of size n/W^8. There will be W^8 blocks and at most W of them should become non-monochromatic. For each output block we can determine whether it will be monochromatic and if so what value it will contain. Given that W^8 is substantially smaller than n this can be done by a circuit of small size and depth.

To recreate the non-monochromatic blocks we subtract from the frequencies the counts of items in monochromatic parts, and we use a naïve circuit of size $O((n/W^7) \cdot 2^w \cdot \text{poly}(w))$ to recreate a sequence of (n/W^7) integers corresponding to those frequencies. We split this sequence into blocks of size (n/W^8) and shuffle them with the monochromatic blocks in correct order. This last step is done using the AKS sorting circuit which sorts according to w bits but drags along each integer a string consisting of (n/W^8) integers. This last step requires a circuit of size $O(W^8 \cdot (nw/W^8) \log W^8)$ and depth $O(\log W^8 \log w)$.

Hence, we obtain a circuit of size $O(nw^2)$ and depth $O(\log n + w \log w)$ for sorting n integers w bits each. Lin and Shi [29] give an incomparable result discussed in the next section.

3 Partial Sorting

In this section we will focus on partial sorting of w-bit integers according to k most significant bits. This corresponds to sorting (key, value) pairs where key has k bits and value has $w - k$ bits. Here, w can be large. Comparison based sorting on such an input would use $O(nk)$ comparisons as seen in the first section. Hence, one could hope for partially sorting circuits of size $O(nwk)$. When $k \in \Omega(w)$ the result of previous section already provides a circuit of size $O(nw^2)$. For small k, one wants smaller circuits.

For arbitrary constant $\varepsilon > 0$, Asharov et al. [7] give a construction of circuits of size $O(nwk(1 + \log^* n - \log^* w)^{2+\varepsilon})$ sorting n integers of w bits each according to their first k bits. First, they design a circuit of size $O(nw(1 + \log^* n - \log^* w)^{2+\varepsilon})$ that sorts the integers according to a single bit. Then they use the circuit to sort the integers by successive bits from the most significant to the least significant. Koucký and Král [24] use the same strategy to first build a circuit for sorting according to one bit (see the next section) and then they apply it iteratively similarly to [7]. This gives a circuit of size $O(nwk(1 + \log^* n - \log^* w))$ and depth $O(\log^3 n)$ that sorts w-bit integers according to their first k bits. In a subsequent work, Lin and Shi [29] get circuits of depth $O(\log n + \log k)$ and size $O(nkw \cdot \mathrm{poly}(\log^* n - \log^* w))$ for $k \in O(\log n)$. Lin and Shi use for their construction the sorting strategy of Ajtai, Komlós and Szemerédi [6] along with the techniques of Asharov et al. [7].

3.1 Sorting according to one bit

In this section we will look at sorting w-bit integers according to a single bit. Sorting a sequence of w-bit integers according to one bit corresponds to the problem of moving designated set of input integers to the beginning or end of the sequence. The set is indicated by the bit according to which we sort. This problem is closely related to routing in graphs and in particular, to routing in superconcentrators.

A *superconcentrator* is a directed acyclic graph with n input nodes and n output nodes that satisfies the property: For any pair of subsets of A and B of input nodes and output nodes, respectively, of the same size ℓ, there are ℓ vertex disjoint paths from vertices in A to vertices of B. Such a superconcentrator can serve as a skeleton of a circuit sorting according to one bit as it can move designated items to their desired positions.

Aho, Hopcroft and Ullman [5] were among the first to observe the connection between routing and various algorithmic problems, and they defined the superconcentrators. Furst, Chandra and Lipton [11] have shown that many natural functions such as addition of two n bit integers require Boolean circuits to have some weak superconcentrator property. It follows from a simple information theoretic argument that the same must be true for sorting circuits. The original motivation to study superconcentrators was to prove lower bounds on their size (number of edges), and hence derive non-linear lower bounds on the size of Boolean circuits for specific functions. The hope was that the superconcentrator property requires graphs to have super-linear number of edges. This turns out not to be the case as shown by Valiant [36]. Indeed, there are superconcentrators with $O(n)$ edges which allow to route any set of items placed on selected inputs to any selected set of outputs (of the same size) along non-intersecting vertex disjoint paths [32,36]. Moreover those graphs have in-degree and out-degree bounded by a constant. Known constructions of superconcentrators rely heavily on expander graphs [21].

One could use any off-the-shelf superconcentrator to solve the sorting problem according to one bit by turning the superconcentrator into a Boolean circuit: One could replace each node of the superconcentrator by a *selector* circuit which would select from among the values coming into the node the one which comes along the edge of a routed path, and propagate the value further. Using a linear size superconcentrator and linear size selector circuits would give a circuit of size $O(nw)$. The only issue is who will tell the selector which edge is active so which value should be propagated.

Pippenger [33] resolved this routing issue in a rather efficient way. He gives a construction of linear size superconcentrator together with an efficient algorithm that determines the routing. The algorithm can be implemented by a small size circuit namely, there is a circuit of size $O(n \log n)$ and depth $O(\log^2 n)$ that gets as its input indicator vectors of sets A and B and outputs a vector indicating which edges should be used to connect A to B by $|A|$-many vertex disjoint paths in the accompanying superconcentrator. Put together, this gives a Boolean circuit of size $O(nw + n \log n)$ and depth $O(\log^2 n)$ that sorts w-bit integers according to one bit. We call this *Pippenger's partially sorting circuit.*

To get a smaller circuit Asharov et al. [7] open up the construction of Pippenger [33] and use his technique to build a smaller circuit from scratch. We will use Pippenger's partially sorting circuit as a black-box to build more efficient circuits.

The cost of Pippenger's partially sorting circuit is dominated by the size of the circuitry to calculate the routing. This takes a circuit of size $O(n \log n)$ but we can afford only $O(nw)$. So we use a similar technique as in Section 2.1 and we will apply this circuit only to blocks of inputs of $2^{O(w)}$ items.

Say we divide the input into blocks of integers of size $n' = 2^{2w}$, and we sort each block by Pippenger's partially sorting circuit. This will give a circuit of total size $O((n/n') \cdot [n'w + n' \log n']) = O(nw)$. Now we partition each block into parts of 2^w integers. All but one part in each block will be monochromatic meaning that each part contains items with the same value of the bit according to which we sort. Now we could try to use a similar technique as in Section 2.1 and apply some naïve algorithm on monochromatic and non-monochromatic parts. Unfortunately, that does not work here as even the items in monochromatic parts may vary. So we will proceed iteratively from here.

The iterative procedure will use parameters $W_0, W_1, \ldots, W_{\log^* n - \log^* w - 1}$ where $W_0 = w$ and $W_{i+1} = 2^{W_i}$. At iteration $i > 0$, we divide the current sequence into parts consisting of W_i items, and we form blocks of $2^{2W_i}/W_i$ parts. Hence, each block contains 2^{2W_i} integers. Within each block there will be at most $2 \cdot 2^{2W_i}/W_i^2$ non-monochromatic parts. In each block we will move the non-monochromatic parts to a side using Pippenger's partially sorting circuit that will consider each part as an item consisting of $W \cdot w$ bits. Then in each block we sort the items in non-monochromatic parts together using Pippenger's partially sorting circuit. The number of those items is at most $2 \cdot 2^{2W_i}/W_i$. We repartition those items into parts of size W_i, and sort all the parts (again as units) within the block using Pippenger's partially sorting circuit according to the appropriate bit of their first item. (We do not care where the single non-monochromatic part ends up.) Notice, if we repartition the block into parts of size $2^{W_i} = W_{i+1}$ then at most two parts in the block out of W_{i+1} parts will be non-monochromatic. At this point we are ready for next iteration $i + 1$.

After the last iteration the total number of parts will be relatively small and the number of items in the non-monochromatic parts will be also small. So we can use few extra Pippenger's partially sorting circuits to finish the sorting. An interested reader might consult [24] for precise details.

The parameters W_i are chosen so that at each iteration the number of items in non-monochromatic parts within a block is $1/W_i$ faction of all the items in the block so we can afford to sort them using Pippenger's partially sorting circuit. Similarly for the number of parts where we are concerned about the logarithmic term in the complexity of Pippenger's partially sorting circuit used to sort the parts. In total, each iteration requires a circuit of size $O(nw)$ and depth $O(\log^2 W_i)$.

Thus we obtain a circuit sorting n w-bit integers according to one bit of total size $O(nw(1 + \log^* n - \log^* w))$ and depth $O(\log^2 n)$. It can be used along the lines of Asharov et al. [7] to build a circuit of size $O(nwk(1 + \log^* n - \log^* w))$ and depth $O(\log^3 n)$ that sorts w-bit integers according to their first k bits.

4 Lower bounds

Sorting is one of the most popular candidate functions for which people try to prove lower bounds. This is rather natural as sorting deals with transfer of information, and information is an aspect of computation that is quantitatively relatively well understood compared to computation itself. Already in 70's researchers tried to prove lower bounds for sorting on Turing machines. Stoss [35] proved an $\Omega(n \log^2 n)$ lower bound for sorting on Turing machines by algorithms that only move items around. This was extended by Paul [31] to algorithms which do not perform any "magic" defined in terms of Kolmogorov complexity. For Turing machines there was only little progress since then.

A modern take on excluding the "magic" is the Network Coding Conjecture of Li and Li [28]. Network coding is a problem in a communication network with ℓ source nodes $s_1, \ldots, s_\ell$ and target nodes $t_1, \ldots, t_\ell$ where each source s_i wants to be transmitting a stream of information to target t_i. The intermediate nodes in the network can combine received messages in arbitrary manner before transmitting them further. The question is at what rate can the source-target pairs communicate when each link has some restricted capacity, and how this rate relates to the multicommodity flow in the corresponding network.

There are well known examples where the information rate can exceed the multicommodity flow in directed networks [1, 3]. The Network Coding Conjecture postulates that such situation cannot occur in undirected graphs [28]. So far the conjecture was established only in restricted settings. It is a compelling formulation of the "no magic" assumption.

There are multiple recent results establishing conditional lower bounds for sorting under the assumption that the Network Coding Conjecture is true such as lower bound for sorting in external memory [16] or for Boolean circuits [2, 7, 30]. Asharov et al. [7] show that under the Network Coding Conjecture, Boolean circuits sorting n integers w bits each partially according to k bits require size $\Omega(nk(w - \log n))$ even with no restriction on the depth of the circuits. Thus under the Network Coding Conjecture the Boolean circuits described in previous sections are almost optimal.

Sorting lower bound can also be derived from other assumptions. For example, Boyle and Naor [10] show that $\Omega(\log n)$ lower bound on the overhead of *offline oblivious RAM* would imply the super-linear lower bound $\Omega(n \log^2 n)$ on the size of sorting circuits. (It was erroneously thought that such lower bounds for oblivious RAM were already established.) Super-linear lower bounds on the size of sorting circuits of logarithmic depth would also follow from a *non-adaptive* n^{ε} lower bound on the number of queries for the function inversion problem with preprocessing of Hellman [12, 15, 19, 37].

All these partial results witness the centrality of sorting problem. In many ways, sorting seems to be the right candidate for proving super-linear size lower bounds for logarithmic depth circuits. Proving such a bound would constitute a major progress in computational complexity and we invite interested readers to take on this challenge. As Andrew Yao puts it: *You cannot win if you don't buy a lottery ticket.*

References

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. Network information flow. *IEEE Trans. Inf. Theory*, 46(4):1204–1216, 2000.

[2] P. Afshani, C. B. Freksen, L. Kamma, and K. G. Larsen. Lower Bounds for Multiplication via Network Coding. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *LIPIcs*, pages 10:1–10:12, 2019.

[3] M. Adler, N. J. A. Harvey, K. Jain, R. D. Kleinberg, and A. R. Lehman. On the capacity of information networks. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)*, pages 241–250, 2006.

[4] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[6] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log(n)$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[7] G. Asharov, W.-K. Lin, and E. Shi. Sorting short keys in circuits of size $o(n \log n)$. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA'21)*, pages 2249–2268. SIAM, 2021.

[8] K. E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314, 1968.

[9] D. Belazzougui, G. S. Brodal, and J. S. Nielsen. Expected linear time sorting for word size $\Omega(\log^2 n \log\log n)$. In *Proceedings of the 14th International Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume 8503 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2014.

[10] E. Boyle and M. Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS'2016)*, pages 357–368, 2016.

[11] A. K. Chandra, S. Fortune, and R. J. Lipton. Unbounded fan-in circuits and associative functions. *J. Comput. Syst. Sci.*, 30(2):222–234, 1985.

[12] H. Corrigan-Gibbs and D. Kogan. The function-inversion problem: Barriers and opportunities. In *Proceedings of the 17th International Conference on Theory of Cryptography, TCC 2019*, pages 393–421, 2019.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[14] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, USA, 2006.

[15] P. Dvořák, M. Koucký, K. Král, and V. Slívová. Data Structures Lower Bounds and Popular Conjectures. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *LIPIcs*, pages 39:1–39:15, Dagstuhl, Germany, 2021.

[16] A. Farhadi, M. Hajiaghayi, K. G. Larsen, and E. 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'19)*, STOC 2019, pages 997–1008, 2019.

[17] Y. Han. Improved fast integer sorting in linear space. *Inf. Comput.*, 170(1):81–94, 2001.

[18] Y. Han. Deterministic sorting in $o(n \log \log n)$ time and linear space. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 602–608, 2002.

[19] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[20] F. C. Hennie. One-tape, off-line turing machine computations. *Inf. Control.*, 8(6):553–578, 1965.

[21] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43:439–561, 2006.

[22] Y. Han and M. Thorup. Sorting integers in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'02)*, 2002.

[23] S. Jukna. *Boolean function complexity: advances and frontiers*, volume 27. Springer Science & Business Media, 2012.

[24] M. Koucký and K. Král. Sorting Short Integers. In *Proccedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *LIPIcs*, pages 88:1–88:17, Dagstuhl, Germany, 2021.

[25] D. E. Knuth. *The Art of computer programming, Volume 3: Sorting and searching*. Addison-Wesley, Reading, MA, 1973.

[26] D. G. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theor. Comput. Sci.*, 28:263–276, 1984.

[27] K. Král. *Complexity of dynamic data structures*. PhD Thesis. Charles University, 2021.

[28] Z. Li and B. Li. Network coding: The case of multiple unicast sessions. *Proceedings of the 42nd Allerton Annual Conference on Communication, Control, and Computing*, 2004.

[29] W.-K. Lin and E. Shi. Optimal sorting circuits for short keys. *arXiv preprint arXiv:2102.11489*, 2021.

[30] W. Lin, E. Shi, and T. Xie. Can we overcome the n log n barrier for oblivious sorting? In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*, pages 2419–2438, 2019.

[31] W. J. Paul. Kolmogorov complexity and lower bounds. In *Proceedings of the Conference on Algebraic, Arithmetic, and Categorial Methods in Computation Theory: Fundamentals of Computation Theory, FCT 1979*, pages 325–334, 1979.

[32] N. Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.

[33] N. Pippenger. Self-routing superconcentrators. *J. of Comp. Syst. Sci.*, 52(1):53–60, 1996.

[34] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[35] H. Stoß. Rangierkomplexität von permutationen. *Acta Informatica*, 2:80–96, 1973.

[36] L. G. Valiant. Graph-theoretic properties in computational complexity. *J. Comput. Syst. Sci.*, 13(3):278–285, 1976.

[37] E. Viola. Lower bounds for data structures with space close to maximum imply circuit lower bounds. *Theory of Computing*, 15(18):1–9, 2019.