IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 18 May 2023, accepted 30 June 2023, date of publication 10 July 2023, date of current version 17 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3293734

RESEARCH ARTICLE

Beyond 0-1: The 1-N Principle and Fast Validation of N-Sorter Sorting Networks

ROBERT B. KENT, (Life Member, IEEE), AND MARIOS S. PATTICHIS, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131, USA
Corresponding author: Robert B. Kent (rkent@unm.edu)

ABSTRACT The well-known 0-1 principle for traditional data-oblivious sorting networks states that a network with L inputs can be fully validated with an input vector set consisting of all 2^L unique vectors containing only the values 0 and 1 in each vector's L inputs. Researchers providing proofs of this principle tend to ignore the fact, which is emphasized here, that 0-1 vectors provide all distinct orderings of the two inputs to the 2-sorters which perform all of the sorting operations in the networks. The authors have recently described single-stage N-sorters, with $N > 2$, and multiway merge sorting networks which use these N-sorters. The new N-sorters and their networks are also data-oblivious. It is easily shown that 0-1 vectors are not sufficient to fully verify even a single-stage 3-sorter, the smallest such N-sorter, as the 0-1 vectors are unable to produce all distinct orderings of the 3 inputs. In order to verify these N-sorters, the authors propose the 1-N principle, which states that testing an N-sorter with all distinct orderings of N input values is necessary and sufficient to prove the N-sorter's correctness. An algorithm is defined which generates a vector set consisting of all distinct orderings of N inputs, which is then used to fully verify the associated single-stage N-sorter. In order to validate the authors' L-input sorting networks which use these N-sorters, methods have been created which produce validation vector sets that are dramatically reduced in size versus the unsorted vector sets they are derived from. For example, the ratios of the number of L! permutation vectors to the equivalent reduced vectors are >1,000,000 for L=12, and are much higher as L increases.

INDEX TERMS 0-1 principle, zero-one principle, sorting networks, 2-sorter, N-sorter, data-oblivious.

NOMENCLATURE

data-oblivious	Refers to a fixed sorting method whose internal operation is independent of its input values.
N	The number of inputs and outputs in a single-stage sorting device.
single-stage N-sorter	A data-oblivious hardware device consisting of one set of N inputs, one set of N outputs, and the internal logic required to transfer the inputs to the outputs in sorted order.
distinct ordering	A unique ordered sequence of all N values in an N-sorter's list of input values.
verification	Confirmation that a single-stage N-sorter correctly sorts all distinct orderings of its N inputs.
L	The number of inputs and outputs in a multistage sorting network.
sorting network	A data-oblivious multistage L-input hardware sorting device, in which a network of single-stage N-sorters is used to provide a sorted list of the L inputs to the network's L outputs.
validation	Confirmation that an L-input sorting network correctly sorts a set of L-valued input vectors.
vector	A set of N input values for an N-sorter, or L input values for a sorting network, which are applied together to the sorting device in order to test its functionality.

The associate editor coordinating the review of this manuscript and approving it for publication was Ilaria De Munari.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

RC-sorted rectangle A rectangular set of values in which each row and each column is sorted.

I. INTRODUCTION

The 1st edition of Donald Knuth's "Sorting and Searching" textbook [1] is generally cited as the original publication which presented the "zero-one" principle. Knuth stated that this principle is a special case of a theorem proposed by W.G. Bouricious in 1954. In the many papers that refer to this principle, the list of words used to name it include "zero-one", "0-1", "zero/one" and "0/1", at a minimum.

The 1st edition of the "Introduction to Algorithms" textbook, by Cormen et al. [2], refers to the 0-1 principle. However, by the 3rd edition, the 0-1 principle was replaced by the 0-1 sorting lemma [3]. The phrase **0-1 principle** is used here.

Traditional sorting networks consist of a fixed, hard-wired set of interconnected 2-sorters. As shown in Fig. 1, a 2-sorter is a fixed hardware device, whose operation does not depend on its inputs' values. A 2-sorter is therefore said to be data-oblivious. Likewise, the operation of a traditional sorting network, which contains only a hard-wired set of 2-sorters, is independent of the network's input values, and is also data-oblivious.

The 0-1 principle states that the operation of a traditional data-oblivious sorting network with L inputs is fully validated using a set of 2^L vectors, where the L inputs in each vector contain a unique ordering of the binary values 0 and 1. Mathematical proofs of this principle are found in [1] and [2] and in other publications. These proofs do not tend to focus on how the 0-1 vectors are processed by the only logic sorting structures utilized in the networks, the data-oblivious 2-sorters.

What is emphasized here is the concept that the 0-1 principle works for traditional sorting networks because the 0-1 vectors applied to the two inputs of each network 2-sorter fully verify all distinct orderings of those two inputs. The input distinct orderings for a 2-sorter, such as the one shown in Fig. 1, are shown below:

In_1 > In_0.
In_1 == In_0.
In_1 < In_0.

Note that an ordering does not depend on the specific input values. The In_1>In_0 ordering is true for (In_1,In_0) pairs (1,0), (2,1), (347,221), etc.

A 2-sorter is called by many different and often confusing names in the technical literature, too many to mention here. The simple and direct name **2-sorter** is used here.

As shown in Fig. 1, a single-stage hardware 2-sorter has two inputs, here labelled In_1 and In_0, and two outputs, Out_1 and Out_0. The two inputs are compared in a block which determines if In_1≥In_0. The comparison result is then used as the select line for two 2-to-1 multiplexers, one

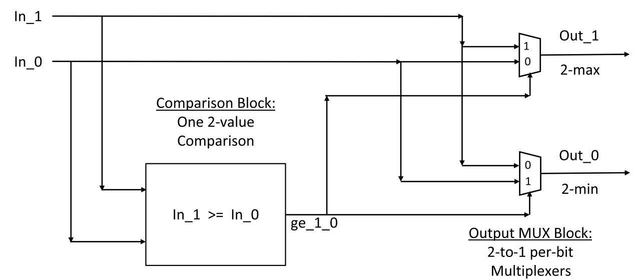


FIGURE 1. Typical single-stage hardware 2-sorter.

TABLE 1. Truth Table for a 2-sorter's 2^2=4 0-1 Vectors.

Vector	In_1	In_0	ge_1_0	Out_1	Out_0	Ordering	Comment
1	0	0	1	0	0	In_1 == In_0	
2	0	1	0	1	0	In_1 < In_0	
3	1	0	1	1	0	In_1 > In_0	
4	1	1	1	1	1	In_1 == In_0	Vec 1 Duplicate

ge_1_0 is the (In_1≥In_0) comparison result.

which selects the max input to go to output Out_1, and the other which selects the min input to go to output Out_0.

Table 1 shows that a 2-sorter's three distinct orderings are fully verified by the 2^2=4 0-1 vectors for the 2-sorter. The In_1==In_0 ordering is tested twice, in vectors 1 and 4.

The authors have recently described the design of single-stage hardware N-sorters, with N>2 [4], [5], and sorting networks which use these N-sorters [6]. Like the 2-sorter shown in Fig. 1, an N-sorter design is fixed, independent of the values of its inputs, and is therefore data-oblivious. The multiway merge sorting networks described in [6] are also data-oblivious, as they are hard-wired devices consisting of a set of interconnected N-sorters and 2-sorters.

It can be easily shown that 0-1 vectors are not sufficient to fully verify an N-sorter with N>2. In fact, it will soon be shown that the 2^3=8 0-1 vectors for a 3-sorter, the smallest N-sorter, cannot fully verify the 3-sorter's operation. Therefore, it should be obvious that 0-1 vectors are not sufficient to fully validate sorting networks containing these N-sorters.

An important contribution of this work is the introduction of the 1-N principle, which states that testing all distinct orderings for an N-sorter's N inputs are necessary and sufficient to fully verify the N-sorter's operation. In addition, a method is defined here which produces the set of 1-N distinct ordering vectors that satisfy this principle.

An N-sorter can be verified with a vector set that contains all N^N sequences of N input values, but such a vector set tends to contain many duplicate orderings, which the 1-N principle states are not required for fully verifying the N-sorter's correctness. It will be shown that for a 10-sorter, the 10^{10} vector set is nearly 100 times larger than the distinct orderings vector set that fully verifies the 10-sorter.

Although the 1-N principle can be used to create validation vector sets for L-input sorting networks using N-sorters, such 1-N sorting network vector sets tend to be very large. For the L-input N-sorter networks introduced by the authors [6], methods are defined here which produce validation vector sets that are much smaller than the 0-1 and L! source vector sets that the reduced vector sets are derived from.

The ratio of the number of 9! vectors to an associated reduced vector set is 8640. Ratios for several 12! vector sets versus the comparable reduced sets are all over one million. As L increases, the ratios continue to increase dramatically. Defining the methods used to produce these impressive vector set reductions is a major contribution of this work.

The rest of this paper is organized as follows: Section II presents an extensive background analysis and critique of many of the publications which refer to the 0-1 principle. A detailed presentation of N-sorter distinct orderings is found in Section IV, starting with an analysis of the input orderings for a single-stage 3-sorter versus those of a 3-input traditional sorting network. Section IV-C defines the 1-N principle, specifies how an N-sorter's distinct ordering vector set is constructed, and presents data on vector set sizes for N-sorters up to 10-sorters. Section V contains a detailed discussion concerning validation of sorting networks using N-sorters, and introduces the methods for constructing the very small vector sets which have been used to validate sorting networks defined in [6] and [7]. In Appendix A, the details of constructing the distinct ordering vectors for a 3-sorter are shown.

More discussion and data concerning the construction of the small vector sets used for sorting network validation are presented in the supplemental material linked to this paper. The supplemental material also includes a detailed discussion of the accompanying video, which shows several examples of how the dramatically reduced vector sets are produced.

II. BACKGROUND

As mentioned in the Introduction, Donald Knuth's first "Sorting and Searching" text [1] is typically cited as the original publication which proposed the 0-1 principle. Knuth stated that this principle is a special case of a theorem proposed by W.G. Bouricious in 1954. A short, single-paragraph, proof of the 0-1 principle was also presented in [1].

At least one earlier publication by David Van Voorhis presented a version of the 0-1 principle [8]. However, Van Voorhis presents a proof for the principle which he says was suggested by Knuth, so it seems that defining and proving the 0-1 principle at that time was somewhat of a communal project at Stanford. In a later publication by Van Voorhis [9], he states that several researchers have independently proven the 0-1 principle, but then specifically references only [1], not his own earlier [8].

Researchers who have joined [1] and [8] in presenting proofs for the 0-1 principle include [2], [10], [11], [12], and [13]. In general, these proofs and similar discussions tend to ignore the main concept emphasized here, that the 0-1 principle works correctly because traditional comparison-based sorting networks exclusively use 2-sorters in the sorting process, and a 2-sorter's 0-1 vectors fully test all distinct orderings of the 2-sorter's two inputs.

It is suggested in [1] that, instead of using the full 0-1 vectors set, a traditional L-input sorting network can be fully validated using an L! permutation vector set in which each of L distinct numbers occurs once and only once in each vector's input list. In fact, the main reason stated in [1] for using the full 0-1 vector set, versus the L! vector set, is that there are typically many fewer vectors in the 0-1 vector set. The suggested superiority of the L! vector set has often been repeated, but never challenged [14], [15], [16], [17], until now.

Although every valid L-input sorting network must be able to pass the L! vector set, passing this set of vectors is not sufficient to fully validate the network. The problem is that none of L! vectors contains any duplicate values, so none of the 2-sorters in the network are ever presented with the both-inputs-equal ordering, found in vectors 1 and 4 in Table 1. The 2-sorters only see Table 1 vectors 2 and 3 distinct orderings. The 2^L 0-1 vector set is therefore superior and sufficient, because the four 0-1 vectors for a 2-sorter test the both-inputs-equal ordering, as well as the two inputs-not-equal orderings.

It should be clear that the Fig. 1 2-sorter correctly processes the both-inputs-equal ordering, but it is possible to build a 2-sorter that correctly sorts the inputs-not-equal orderings, but does not correctly sort all both-inputs-equal input vectors. An example of such a (deliberately ?) defective 2-sorter is one that correctly sorts the two inputs-not-equal orderings, but always sets the outputs of a both-inputs-equal ordering to 0. This defective 2-sorter will pass Table 1 vector 1, but fail vector 4.

The all-inputs-equal ordering becomes more serious for the single-stage N-sorters defined in [4] and [5], with N> 2, as the input-to-output mapping is dependent on multiple comparisons, not just one comparison. In fact, it can be shown that the sort-3a 3-sorter in [18] fails a full all-inputs-equal test. When all 3 inputs are equal, sort-3a sets the max and min outputs to 0, no matter what the common value is on the inputs.

A number of publications extend the 0-1 principle to circuits other than sorting networks. These extensions are presumably all valid, as long as all operations in the circuit are performed by 2-sorters. An example of this type of 0-1 extension is proposed for comparison-based switching networks in [19].

A number of researchers have proposed building sorting networks using single-stage devices which sort more than 2 values, typically without dealing with how such single-stage devices could be built. The sorting networks using these single-stage devices cannot be fully validated using only 0-1 vectors, and the various researchers have dealt with this issue in a number of ways. The proposed single-stage devices have a number of different names, usually similar to N-sorter, but with a different letter used instead of N.

In a later printing of his classic text [20], Knuth discusses single-stage m-sorters, 4-sorters in particular. In the m-sorter network discussion, only 0-1 values are used, even though the 0-1 principle doesn't apply to networks using single-stage 4-sorters. The discussion in [20] using 0-1 values appears to be based on the flawed k-sorter network 0-1 discussion in [21]. Other publications which incorrectly use 0-1 vectors for N-sorter network validation include [22], [23], and [24].

The Cubesort researchers [25] state that single-stage N-sorters, which they do not define, can be used to sort more than 2 values at a time. However, they note that the 0-1 vectors can be used to validate the networks using these N-sorters, as long as the single-stage N-sorters are replaced during the validation process with equivalent 2-sorter networks that sort N values. Once the network containing only 2-sorters is validated, the original network containing verified single-stage N-sorters has been validated as well.

III. DATA-OBLIVIOUS FPGA SORTING STRUCTURES

The authors' data-oblivious sorting devices [4], [5], [6] are not limited to a particular type of hardware, but the target hardware used in these papers were modern AMD-Xilinx FPGAs. Traditional sorting networks, such as Kenneth Batcher's Odd-Even Merge Sort and Bitonic Merge Sort [7], are also easily implemented in these FPGAs.

All of these data-oblivious devices can be fully implemented in combinatorial logic, using the 6-input FPGA look up tables (LUTs). Thus, no internal clocking or memory is used in the sorting device operation. The primary metrics used to judge the device performance are LUT usage and the combinatorial propagation delay of the slowest input-to-output signal. These base combinatorial logic designs were used to produce the data reported in the authors' 3 papers.

These data-oblivious devices can also be easily implemented in fully pipelined designs. In this case, clocking is used to capture LUT output signals in associated flip-flops (FFs), as needed. There is one neighboring FF available for each LUT output in the FPGA, so no addressable memory is required. For an L-input device implemented in the FPGA, a new set L inputs can be applied to the pipeline inputs every clock cycle, e.g., every 2 nS when using a 500 MHz clock. Likewise, a set of L outputs can be read out of the device every clock cycle.

The number of LUTs used for both the combinatorial and fully pipelined implementations is the same. The FFs used in the fully pipelined design are simply not used in combinatorial logic design. The speed performance metrics for the fully pipelined design are throughput and latency. Throughput is now L values every clock cycle, e.g., every 2 nS as noted above, for any L-input device that can be constructed in the FPGA.

Latency is the number of clock cycles required between the application of the L unsorted inputs and the read out of the fully sorted set of L outputs. Latency is minimized in these designs by minimizing the number of LUTs in series between input-to-output signals. Minimizing the number of series LUTs in input-to-output paths is the primary goal in designing fast combinatorial networks, so latency is minimized simply by using the paths already optimized for combinatorial network speed.

Software/CPU sorting algorithms can be data-oblivious or they can respond to the data values of particular inputs in order to speed up the sorting process. In either case, the designs will be implemented using CPU clocking, and algorithm metrics become memory usage and the time to fully sort a list of inputs.

It is difficult for a CPU sorting algorithm to match the parallelism found in hardware sorting networks. Furthermore, it does not seem possible for a Software/CPU algorithm to match the fully pipelined hardware implementation, particularly its very high throughput, so it is difficult to reasonably compare software/CPU sorting performance to that of the data-oblivious sorting hardware which has been discussed here and in the authors' recent papers.

Data-oblivious software/CPU algorithms may make use of the 0-1 principle, as long as all operations in the algorithm are 2-sorter operations. Should future software/CPU algorithms attempt to implement the authors' N-sorters [4], [5] or multiway merge sorting networks [6], the verification and validation methods described in the following Sections IV and V should prove valuable in proving the correctness of such algorithms.

IV. DISTINCT ORDERINGS OF N VALUES

The sorting networks that are covered by the 0-1 principle only use 2-sorters as the hardware sorting devices in each stage of the sorting process. The success of the 0-1 principle is based on the fact that the 0-1 vectors for a 2-sorter fully test all distinct orderings of the two inputs.

Passing a 0-1 vector set is not sufficient to verify the correctness of the single-stage hardware N-sorters defined in [4] and [5], with N>2. This is because the N-sorter 0-1 vectors do not cover all distinct orderings of the N input values. In Section IV-A, this fact will be shown for the smallest N-sorter, the 3-sorter. Section IV-B presents the sequential operation of a multistage 2-sorter network which sorts 3 values, which is then compared to the single-stage 3-sorter behavior discussed in Section IV-A.

The authors have developed an algorithm and matching software program, which are used to generate the full distinct ordering vector set for an N-sorter. Test vectors produced by the program have been used to verify the correctness of all N-sorters from 3-sorters up to 10-sorters. The algorithm basics are presented in Section IV-C, as well as data showing the vector reduction of an N-sorter's distinct ordering vector set versus an N^N comprehensive vector set.

A. SINGLE-STAGE 3-SORTER INPUT ORDERINGS

To verify the correctness of an N-sorter, all distinct orderings of the N inputs must be tested. A vector set with all N^N vectors containing values 1 to N will include vectors with all distinct

TABLE 2. All 27 Vector States and Orderings of the Single-stage 3-sorter Input Value Sets (0, 1, 2) and (1, 2, 3).

		Values 0-2										|<- (ge_2_1, ge_2_0, ge_1_0) ->|											Values 1-3			
Vector	In_2	In_1	In_0	ge_2_1	ge_2_0	ge_1_0	Out_2	Out_1	Out_0	0_0_0	0_0_1	0_1_0	0_1_1	1_0_0	1_0_1	1_1_0	1_1_1	Ordering	Comment	In_2	In_1	In_0
1	0	0	0	1	1	1	0	0	0								X	1 Value #1		1	1	1
2	1	1	1	1	1	1	1	1	1								X	1 Value #1	Vec 1 Duplicate	2	2	2
3	2	2	2	1	1	1	2	2	2								X	1 Value #1	Vec 1 Duplicate	3	3	3
4	0	0	1	1	0	0	1	0	0					X				2 Values #1		1	1	2
5	0	1	0	0	1	1	1	0	0				X					2 Values #2		1	2	1
6	0	1	1	0	0	1	1	1	0		X							2 Values #3		1	2	2
7	1	0	0	1	1	1	1	0	0								X	2 Values #4		2	1	1
8	1	0	1	1	1	0	1	1	0							X		2 Values #5		2	1	2
9	1	1	0	1	1	1	1	1	0								X	2 Values #6		2	2	1
10	0	0	2	1	0	0	2	0	0					X				2 Values #1	Vec 4 Duplicate	1	1	3
11	0	2	0	0	1	1	2	0	0				X					2 Values #2	Vec 5 Duplicate	1	3	1
12	0	2	2	0	0	1	2	2	0		X							2 Values #3	Vec 6 Duplicate	1	3	3
13	2	0	0	1	1	1	2	0	0								X	2 Values #4	Vec 7 Duplicate	3	1	1
14	2	0	2	1	1	0	2	2	0							X		2 Values #5	Vec 8 Duplicate	3	1	3
15	2	2	1	1	1	1	2	2	0								X	2 Values #6	Vec 9 Duplicate	3	3	2
16	1	1	2	1	0	0	2	1	1					X				2 Values #1	Vec 4 Duplicate	2	2	3
17	1	2	1	0	1	1	2	1	1				X					2 Values #2	Vec 5 Duplicate	2	3	2
18	1	2	2	0	0	1	2	2	1		X							2 Values #3	Vec 6 Duplicate	2	3	3
19	2	1	1	1	1	1	2	1	1								X	2 Values #4	Vec 7 Duplicate	3	2	2
20	2	1	2	1	1	0	2	2	1							X		2 Values #5	Vec 8 Duplicate	3	2	3
21	2	2	0	1	1	1	2	2	1								X	2 Values #6	Vec 9 Duplicate	3	3	1
22	2	1	0	1	1	1	2	1	0								X	3 Values #1		3	2	1
23	2	0	1	1	1	0	2	1	0							X		3 Values #2		3	1	2
24	1	2	0	0	1	1	2	1	0				X					3 Values #3		2	3	1
25	1	0	2	1	0	0	2	1	0					X				3 Values #4		2	1	3
26	0	2	1	0	0	1	2	1	0		X							3 Values #5		1	3	2
27	0	1	2	0	0	0	3	2	1	X								3 Values #6	No 0-1 !	1	2	3
Count of Xs										1	4	0	4	4	0	4	10					

ge_2_1 = (In_2 ≥ In_1) ge_2_0 = (In_2 ≥ In_0) ge_1_0 = (In_1 ≥ In_0)

orderings, but the N^N vector set will tend to include multiple vectors whose ordering is a duplicate. For the 2-sorter's 2^2 vectors shown in Table 1, both vectors 1 and 4 exhibit the all-inputs-equal ordering. Since vector 1 is the first vector with this ordering, it is defined as the distinct ordering vector, and vector 4 is marked as a duplicate.

In [4], it was shown that its single-stage 3-sorter passed a vector set of the 3^3=27 vectors that contain all orderings of 3 input values. The 3-sorter's set of 27 vectors are shown in Table 2, with the 2nd through 4th columns containing all possible (In_2, In_1, In_0) orderings of values (0,1,2). The values in the 2nd through 4th columns are used in the 0-1 discussions in this section, and in Section IV-B. The last 3 columns in Table 2 contain (In_2, In_1, In_0) orderings of values (1,2,3). The values in the last 3 columns are used in the 1-N discussions in Section IV-C.

Table 2 has two columns with no data, highlighted in blue. The blue columns indicate the two states of the three comparison result signals, (ge_2_1, ge_2_0, ge_1_0), which are not possible.

In Table 2, all vectors having a duplicate ordering are highlighted in green, and marked as a duplicate in the Comment column. The first vector with a particular ordering, a distinct ordering, is not marked as a duplicate, and that vector's row is not highlighted in green. For example, the first 3 vectors all exhibit the all-inputs-equal ordering. Vector 1, with all inputs equal to 0, is the first vector with this ordering, and is not highlighted. Vectors 2 and 3, with all inputs equal to 1 and 2 respectively, are highlighted and identified as duplicates.

Vectors 10 to 21 have various duplicate orderings of vectors 4 to 9. For example, vectors 10 and 16 have the same ordering as vector 4, with In_2 equal to In_1, and In_0 higher than In_2 and In_1. While the 2-sorter only had 1 duplicate in 4 vectors, the 3-sorter has 14 duplicates in 27 vectors. As will be seen, the number of duplicate ordering vectors increases dramatically as N increases.

The 0-1 vectors for the 3-sorter are the 2^3=8 vectors 1-2 and 4-9. Only vector 2, the all-inputs-are-1s vector is a duplicate. This behavior is a constant for all N: the (2^N-2) 0-1 vectors with both 0 and 1 input values are distinct ordering vectors, the all-inputs-are-0s is also a distinct ordering vector, but the ordering of the all-inputs-are-1s vector is a duplicate of the all-inputs-are-0s vector.

The final 6 vectors in Table 2 are the 3! vectors in which each of the input values (0,1,2) appears once and only once. The last vector, vector 27, is highlighted in yellow. The (In_2, In_1, In_0) input order is uniformly increasing for this vector, and it is the only vector with each of the 3 (ge_2_1, ge_2_0, ge_1_0) comparison signals equal to 0.

Note that none of the 0-1 vectors can produce this (ge_2_1, ge_2_0, ge_1_0) comparison signal state. When there are 3 0-1 input values in a vector, at least two inputs

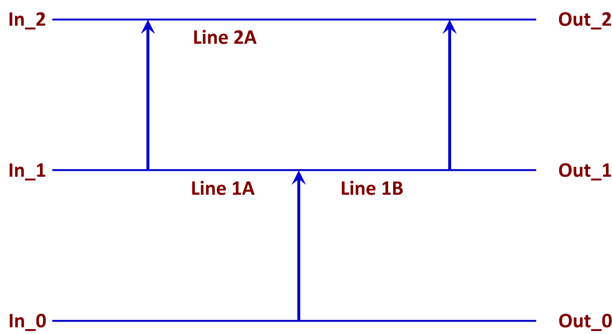


FIGURE 2. Sorting network of 3 values using 2-sorters.

must have the same input value. Because of this, at least one of the (ge_2_1, ge_2_0, ge_1_0) signals will be a 1. It is clear that 0-1 vector inputs are incapable of testing the uniformly increasing input distinct ordering for the 3-sorter. In short, 0-1 vectors are not sufficient to verify all distinct orderings of a 3-sorter, the smallest N-sorter with N>2.

B. SORTING 3 VALUES IN A 2-SORTER NETWORK

The previous section defined a set of 13 vectors whose 3 inputs produced all distinct orderings of those 3 inputs. A single-stage 3-sorter must produce a correctly sorted output list for each of the 13 vectors in order to pass verification.

This section presents a 3-stage sorting network, using 2-sorters, which sorts 3 inputs. The operation of this 3-stage network is compared to that of the single-stage 3-sorter, with a particular focus on how it deals with the full 0-1 vector set.

Just as [1] is considered the original 0-1 principle publication, it was also the first to present a type of diagram used to define a 2-sorter sorting network, often called a Knuth diagram. Fig. 2 is a Knuth diagram for a 3-stage sorting network used to sort 3 values. The arrows in the 2-sorter vertical lines point to where the max of the 2 values will be placed.

As with the single-stage 3-sorter, there are 3 comparisons in this network. However, the 3-sorter comparisons are all concurrent, and operate on the input values directly. In the Fig. 2 network, only one comparison is performed at a time. The set of network comparisons form a sequence over 3 stages in series.

For the Fig. 2 network, only the first comparison, ge_2_1=(In_2≥In_1), operates directly on the network inputs. The second comparison, ge_L1A_0=(Line 1A≥In_0), operates on one input and one value internal to the network, and the last comparison, ge_L2A_L1B=(Line 2A≥Line 1B), operates on two internal values.

Table 3 shows results for the 8 0-1, plus the 6 (=3!) input vectors when applied to the Fig. 2 sorting network. These 14 vectors consist of the distinct ordering vectors shown in Table 2, plus vector 2. Vector 2 is a duplicate ordering vector but is a part of the 8 0-1 vectors. The 8 0-1 vectors, 1-2 and 4-9, are separated by a horizontal line from the 3! vectors in Table 3. Once again, there are two blue empty columns, which indicate that comparison sequence for that column is impossible to achieve.

Since the 3-input sorting network uses only 2-sorters, it should be fully validated by using the 8 0-1 vectors. At first glance, it may appear that 0-1 vectors are not sufficient, as the vector 27 comparison sequence is not found in the 0-1 vectors. This is the same vector that was shown to have a problem for the 3-sorter in Table 2. However, unlike the single-stage combined state of the 3 comparisons shown in Table 2, the comparison sequence in the 3 stages in Table 3 does not matter. Only the single comparison state in each of the 3 sequential stages is important.

Each comparison result for vector 27, whose inputs are uniformly increasing, is a 0. But vectors 5 and 6 test the same (In_2≥In_1)== 0 comparison state found in the first vector 27 comparison. Vectors 4, 6, and 8 test vector 27's (Line 1A≥In_0)== 0 second comparison state, and vector 4 tests vector 27's (Line 2A≥Line 1B)== 0 final comparison state.

It is also easy to see that the 3-input Fig. 2 network correctly sorts the uniformly increasing input vector, as shown in the sequential steps listed below:

(0,1,2): The (In_2 , In_1 , In_0) inputs. Increasing.
(1,0,2): In 1st 2-sorter, 0 and 1 swap.
(1,2,0): In 2nd 2-sorter, 0 and 2 swap.
(2,1,0): In last 2-sorter, 1,2 swap. Sorted. Decreasing.

C. THE 1-N PRINCIPLE AND DISTINCT ORDERINGS

A single-stage N-sorter can be verified using an input vector set containing all N^N orderings of N input values. Table 1 displays all 4 2^2 input orderings containing values 0 and 1, the 0-1 vectors, for a 2-sorter. Table 2 displays the 27 3^3 input orderings for a 3-sorter. The 1-N discussion in this section uses the (1,2,3) value set found in the rightmost 3 columns of Table 2.

A set of N^N vectors contain duplicate orderings, which are not required to verify the correctness of the sorter. The 2-sorter's 2^2 0-1 vectors only contain 1 duplicate ordering out of 4 vectors, but over half of the 3-sorter's 3^3 vectors, 14 of 27, are unneeded duplicate orderings. As will be seen, the number of duplicate orderings increases dramatically as N increases.

A major contribution of this work is the introduction of the 1-N principle, which states that a single-stage N-sorter is fully verified when it is successfully tested with all distinct orderings of N input values. In support of the 1-N principle, an algorithm and matching software program have been created, which are used to produce the complete set of distinct orderings for an N-sorter's N input values.

Each execution of the software program, called **distinct_orderings_J_in_N**, produces the complete set of distinct ordering vectors which contain J distinct values in each vector's list of N inputs. The program is executed N times, with J ranging from 1 to N, and the N output vector

TABLE 3. The 8 0-1 Vectors and 6 (=3!) Vectors for the 3-input Sorting Network Using 2-sorters.

Vector	In_2	In_1	In_0	ge_2_1	ge_L1A_0	ge_L2A_L1B	Line 2A	Line 1A	Line 1B	Out_2	Out_1	Out_0	0_0_0	0_0_1	0_1_0	0_1_1	1_0_0	1_0_1	1_1_0	1_1_1	Ordering	Comment
1	0	0	0	1	1	1	0	0	0	0	0	0								X	1 Value #1	
2	1	1	1	1	1	1	1	1	1	1	1	1								X	1 Value #1	Duplicate
4	0	0	1	1	0	0	0	0	1	1	0	0					X				2 Values #1	
5	0	1	0	0	1	1	1	0	0	1	0	0				X					2 Values #2	
6	0	1	1	0	0	1	1	0	1	1	1	0		X							2 Values #3	
7	1	0	0	1	1	1	1	0	0	1	0	0								X	2 Values #4	
8	1	0	1	1	0	1	1	0	1	1	1	0						X			2 Values #5	
9	1	1	0	1	1	1	1	1	1	1	1	0								X	2 Values #6	
22	2	1	0	1	1	1	2	1	1	2	1	0								X	3 Values #1	
23	2	0	1	1	0	1	2	0	1	2	1	0						X			3 Values #2	
24	1	2	0	0	1	1	2	1	1	2	1	0				X					3 Values #3	
25	1	0	2	1	0	0	1	0	2	2	1	0					X				3 Values #4	
26	0	2	1	0	0	1	2	0	1	2	1	0		X							3 Values #5	
27	0	1	2	0	0	0	1	0	2	2	1	0	X								3 Values #6	Ok !
Count of Xs													1	2	0	2	2	2	0	4		

The (ge_2_1, ge_L1A_0, ge_L2A_L1B) comparison signals form a 3-stage sequence of logic states, one state per stage.
The 0-1 vectors, 1-2 and 4-9, fully verify the single 2-sorter comparison state in each of the 3 stages.

sets are then concatenated into a complete distinct ordering vector set, which is then used to verify the N-sorter operation.

Generation of the distinct ordering vectors for a 3-sorter requires 3 executions of the program, such as in a bash shell, as shown below. The first program execution creates vector file vecs_3_sorter, when J=1. The second and third program executions append their J=2,3 vector lists to vecs_3_sorter, resulting in the final complete distinct ordering vector set.

```
distinct_orderings_J_in_N  1  3   >  vecs_3_sorter
distinct_orderings_J_in_N  2  3  >>  vecs_3_sorter
distinct_orderings_J_in_N  3  3  >>  vecs_3_sorter
```

These three program executions create the 13 unique distinct orderings shown in the last 3 columns of Table 2, using the 1-N input values (1,2,3). The first 3-sorter program execution above creates the single all-1s vector. The second 3-sorter program, with J=2, creates the 6 1-2 vectors listed in vectors 4 to 9. The third 3-sorter program, with J=3, creates 6 1-2-3 vectors, the 3! permutation vectors, from vectors 22 to 27.

A simplified **distinct_orderings_J_in_N** program flow is shown in Algorithm 1, and the details of how the program generates the vectors for each of the 3-sorter runs shown above are given in Appendix A. The program creates all input vectors in which values J down to 1, and only values J to 1, occur at least once in a vector of length N. Each complete vector is a distinct ordering vector for an N-sorter, and it is sent to the program's standard output.

In Algorithm 1, each call to recursive subroutine SET_VEC_LOC processes one of the (N-1) down to 0 locations (VecLocs) in a vector. For a particular VecLoc vector location, each value from J down to 1 is sequentially tested to see whether, if it is the value at location VecLoc, a distinct ordering vector can still be created.

If a distinct ordering vector is not possible when a value is tested at a specific VecLoc, the program flow moves on. If a distinct ordering vector is possible, and VecLoc>0, the subroutine is recursively called, this time to process the VecLoc-1 vector location. If a distinct ordering vector is possible, and VecLoc=0, the distinct ordering vector is not only possible, it is now complete, and it is then written out to program's standard output.

For example, when J and N are both 3, the recursive subroutine SET_VEC_LOC is first called with VecLoc=2, the leftmost vector location, and an empty vector. In the first iteration of the distinct_value loop, the value 3 is added to the

Algorithm 1 Program **distinct_orderings_J_in_N**

```
▷ This program creates and outputs distinct ordering vectors.
▷ A distinct ordering vector contains each value from 1 to J.
▷ A distinct ordering vector contains N total values.
    Input: J, number of distinct values in each input vector.
    Input: N, total number of values in each input vector.
 1: vector = empty
 2: SET_VEC_LOC( (N-1), vector)
      ▷ VecLoc location in vector ranges from (N-1) to 0
 3: procedure set_vec_loc(VecLoc,vector)
 4:     for distinct_value = J downto 1 do
 5:         vector[VecLoc]=distinct_value
 6:         if distinct ordering vector still possible then
 7:             if VecLoc == 0 then
 8:                 write vector to standard output
 9:             else
10:                 SET_VEC_LOC( (VecLoc-1), vector)
11:             end if
12:         end if
13:     end for
14: end procedure
```

TABLE 4. N-sorter Distinct Orderings: J Distinct Values in Input Vectors with N Total Values.

J	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9	N=10
1	1	1	1	1	1	1	1	1	1
2	2	6	14	30	62	126	254	510	1,022
3		6	36	150	540	1,806	5,796	18,150	55,980
4			24	240	1,560	8,400	40,824	186,480	818,520
5				120	1,800	16,800	126,000	834,120	5,103,000
6					720	15,120	191,520	1,905,120	16,435,440
7						5,040	141,120	2,328,480	29,635,200
8							40,320	1,451,520	30,240,000
9								362,880	16,329,600
10									3,628,800
Total	3	13	75	541	4,683	47,293	545,835	7,087,261	102,247,563
N^N	4	27	256	3,125	46,656	823,543	16,777,216	387,420,489	10,000,000,000
N^N/Total	1.33	2.08	3.41	5.78	9.96	17.41	30.74	54.66	97.80

leftmost vector location, VecLoc 2, and then SET_VEC_LOC is called in order to move onto VecLoc 1.

For VecLoc 1, the distinct_value 3 is first tested to see if it may be added to the vector at location 1. The answer is no. When J and N are both 3, each value from 3 down to 1 must occur once and only once in the vector, and 3 has already been added to the vector at location 2.

In the next loop iteration for VecLoc 1, distinct_value 2 is tested to see if it can be correctly added to the vector at location 1. Since 2 is not yet in the vector, the answer is yes. It is added to the vector and SET_VEC_LOC is called to process VecLoc 0, the rightmost vector location, now with a vector containing 2 out of the 3 values.

For VecLoc 0, values 3 and 2 are rejected, as they are already in the vector, but value 1 is accepted and written into the vector at location 0. This distinct ordering vector is then written to the standard output.

In order to confirm that the distinct ordering algorithm and program work correctly, a second program was created which also produces the complete distinct ordering vector set for an N-sorter. This second confirmation program initially produces the N^N vectors containing all possible vectors with N inputs, in which the input values range from 1 up to N. Each vector's ordering is then mapped to a base ordering vector. If the base ordering vector is new, it is added to the set of distinct ordering vectors. If the base ordering vector is a duplicate, nothing is done and the program moves onto the next one of the N^N vectors.

In order to create a base ordering vector from a selected N^N vector, all of the minimum values in the selected vector are mapped to value 1 in the base ordering vector. The values in the selected vector that are the next highest above the minimum are mapped to value 2 in the base ordering vector, and so on.

Referring to the 3^3 vectors in Table 2, vectors 1-3 all map to an all-1s base ordering vector when using the (1,2,3) values in the table's last 3 columns. Vector 1 is this base ordering vector, so it is kept by the confirmation program, but vectors 2 and 3 are ignored.

Likewise, vectors 4, 10, and 16 all have the same base ordering vector, in which In_2 and In_1 are both 1s, and In_0 is a 2. The confirmation program will keep vector 4 as the base ordering vector, and ignore vectors 10 and 16. In a similar manner, all vectors 10 to 21 can be shown to have the same base ordering vector as one of vectors 4 to 9, so vectors 10 to 21 will all be ignored.

The distinct ordering vector sets created by the **distinct_orderings_J_in_N** program and the confirmation program have been compared for 3-sorters up to 9-sorters. In all cases, the two vector sets are identical.

Table 4 shows the counts of the distinct ordering vectors produced by the **distinct_orderings_J_in_N** program mentioned just above. The data for the first two columns, with N=2 and N=3, have already been discussed in the Introduction and in Section IV-A.

In the J=1 row, the single distinct ordering vector is always the all-1s vector. In the J=2 row, the distinct ordering vectors are the 2^N-2 vectors containing only values 1 and 2.

The lowest data value in a column's yellow highlighted section of results is the count of vectors when J=N, where each vector contains 1 and only 1 of the values 1 to N. This N! vector set is the type of permutation vector set that has been discussed earlier.

Although the total number of distinct ordering vectors grows significantly as N increases, the number of N^N vectors grows even more dramatically. While the number of N^N vectors is roughly twice as many as the distinct ordering vectors when N=3, the ratio is almost 100 for N=10.

V. VALIDATING N-SORTER SORTING NETWORKS

As has been stressed earlier, the 2^L 0-1 vector set fully validates a traditional L-input sorting network, because the 0-1 input values applied to each 2-sorter in the network fully cover all possible distinct orderings of 2 inputs. For sorting

networks using N-sorters with N>2, such as the authors' multiway merge sorting networks [6], a similar validation principle holds.

When an L-input sorting network uses N-sorters with N> 2, the largest N-sorter in the network is called the $N_{largest}$-sorter. The network can be fully validated with a vector set comprised of all distinct $N_{largest}^{L}$ vectors containing values 1 to $N_{largest}$.

For the merge processes in the type of sorting networks described in [6], the input sorted lists to be merged are combined into a rectangular structure, with each of the **Ncols** input sorted lists becoming a column in the structure. The process of merging Ncols sorted lists into a single sorted output list is here called an Ncols-way merge.

For an Ncols-way merge process, 2-sorters and N-sorters up to Ncols-sorters are implemented in the network. Therefore $N_{largest}$=Ncols, and the L-input Ncols-way merge network will be fully validated using a vector set containing all Ncols^{L} vectors with L inputs, with the input values ranging from 1 to Ncols.

For example, a 16-input 4-way merge network will be fully validated using a vector set with 4^{16}, over 4 billion, vectors. A traditional 16-input sorting network, using only 2-sorters, will require a much more manageable 2^{16}, 64K, vectors. It should be easy to see that validating an Ncols-way network using Ncols^{L} L-length input vectors, with values ranging from 1 to Ncols, becomes unworkable as Ncols and L increase.

However, as suggested in [25], the number of vectors required to validate a network using N-sorters can be reduced to the number of required 0-1 vectors, if each of the N-sorters in the network is replaced by an N-input sorting network which sorts the N values using 2-sorters. For example, a replacement 3-way L-input [6] network, using 2-sorters and 3-sorters, will have its single-stage 3-sorters replaced by the 3-input sorting network shown in Fig. 2, and then validated using the 2^{L} 0-1 vector set.

If the replacement 2-sorter network passes validation, then the network design itself is proven valid. The original network using N-sorters is therefore also valid, once the N-sorters themselves have been verified using the 1-N principle.

In short, L-input N-sorter networks can be fully validated using the 2^{L} 0-1 vector sets used for traditional 2-sorter network validation. However, the number of vectors required is an exponential function of L, and validating these networks using 0-1 vectors also eventually becomes unmanageable as L increases.

A. USING RC-SORTED RECTANGLES TO CREATE VERY SMALL VECTOR SETS FOR N-SORTER SORTING NETWORK VALIDATION

As noted just above, neither the 1-N principle nor [25]'s replacement of N-sorters with N-input 2-sorter networks are able to enable L-input N-sorter network validation using small vector sets as L increases. In order to solve this issue, the authors have developed methods which produce dramatically smaller vector sets for validation of their Ncols-way merge sorting networks with Ncols>2 [6].

These much smaller vector sets still validate the sorting network operation as fully as the unsorted vector sets from which they are derived. Although the authors' Ncols-way merge networks are novel when Ncols>2, they are equivalent to Kenneth Batcher's 2-way Odd-Even Merge Sort networks when Ncols=2 [7], and therefore Odd-Even Merge networks can also be validated with the very small vector sets that are introduced here.

In the authors' Ncols-way merge networks [6], Ncols sorted lists are arranged as the columns in a rectangle, and then several stages of operations are performed on the rectangle in order to put the rectangle values in final sorted order. When a rectangle is constructed, the max value in each sorted column goes to the top of rectangle, and the min column value goes to the rectangle bottom, at row 0.

In the first merge stage operation, the rectangle rows are sorted. It has long been known that, after such an operation, each column of this modified rectangle is still sorted, in addition to the just-sorted rows [26]. The rectangles after this row sort stage are here called RC-sorted rectangles. After row sort, the RC-sorted rectangles are processed in the remaining stages of the merge methodology, producing the final rectangle with the values in the defined sorted order.

Batcher's 2-way Bitonic Merge Sort [7] can also be implemented in a 2-column rectangular structure, with the first stage operation also being a row sort stage. In the case of Bitonic Merge Sort rectangles, however, one of the lists has its min value at the top of the rectangle, and its max value at the bottom row 0. Because of this, the RC-sorted state described in [26] does not apply to Bitonic Merge rectangles after the rows are sorted.

Instead of working with parent lists of unsorted input vectors, the method described here creates a complete set of RC-sorted rectangles, in which the rectangle values are derived from the definition of the parent vector set. The number of RC-sorted rectangles is much smaller than the number of unsorted parent vectors that the RC-sorted rectangles are derived from, even though none of the information from the set of parent unsorted vectors has been lost.

For example, there are 9!=362, 880 permutation vectors with values 9 down to 1 occurring only once in each vector. But if each vector is broken into 3 groups of 3, each group is sorted, each sorted group becomes a column in a 3 × 3 rectangle, and each rectangle row is subsequently sorted, there are only 42 distinct rectangle orderings. When the remaining network sorting steps are applied to each of the 42 RC-sorted rectangles, a valid sorting network process always produces the same final rectangle, in fully sorted order.

Program **create_all_RC-sorted_rectangles** uses Algorithms 2 and 3 to create all RC-sorted rectangles for a list of L values, where each value from L down to 1 occurs once and only once in the rectangle. For the set of 42 3 × 3 rectangles, the first rectangle created using this program is the left rectangle shown in Fig. 3. One of the RC-sorted rectangles will

Algorithm 2 Program **create_all_RC-sorted_rectangles**

```
▷ Creates all RC-sorted rectangles, with Ncols columns and
▷       L values, L to 1 occurring only once in each rectangle.
▷ Rectangle columns from left (Ncols-1) down to right 0.
▷ Rectangle rows from top (Nrows-1) down to bottom 0.
▷ Each column max is at the top, min at the bottom.
▷ Each row max is at the left, min at the right.
▷ L is an integer multiple of Ncols.
    Input: L, the number of distinct values in the rectangle.
    Input: Ncols, the number of columns in the rectangle.
 1: Nrows = L / Ncols
 2: Create empty Ncols x Nrows rectangle
 3: Set upper left rectangle location to L
 4: ADD_NEXT_VALUE( (L-1), rectangle)
 5: procedure ADD_NEXT_VALUE(value,rectangle)
 6:     if value == 1 then
 7:         Set lower right rectangle location to 1
 8:         Write rectangle or vector to standard output
 9:         Set lower right rectangle location to empty
10:     else
11:         CREATE_LOCS_LIST(rectangle,Ncols,Nrows)
12:         ForEach location in the locations list do
13:             Set location to value
14:             ADD_NEXT_VALUE( (value-1), rectangle)
15:             Set location to empty
16:         end ForEach
17:     end if
18: end procedure
```

Algorithm 3 Function **CREATE_LOCS_LIST**

```
▷ Using input PartRect, a partially populated rectangle array,
▷         and rectangle parameter inputs Ncols and Nrows,
▷ Create/output list of valid rectangle locations for next value.
    Input: PartRect, partially populated rectangle.
    Input: Ncols, the number of columns in the rectangle.
    Input: Nrows, the number of rows in the rectangle.
    Output: list of valid rectangle locations for next value
 1: For c = (Ncols-1) downto 0 do
 2:     r = (Nrows-1)                                ▷ The top row
 3:     select PartRect[r][c]
 4:     while (PartRect[r][c] is populated) AND (r > 0) do
 5:         r = r - 1
 6:         select PartRect[r][c]
 7:     end while
 8:     if PartRect[r][c] is NOT populated then
 9:         if c == Ncols-1 then
10:             add PartRect[r][c] to the valid location list
11:         else if PartRect[r][c+1] is populated then
12:             ▷ [r][c+1] is immediately to the left of [r][c]
13:             add PartRect[r][c] to the valid location list
14:         end if
15:     end if
16: end For
```

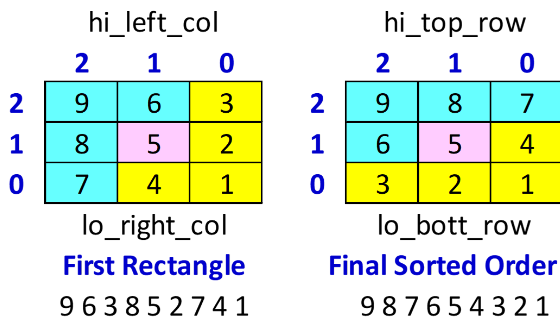


FIGURE 3. Two RC-sorted 3 × 3 rectangles and vectors.

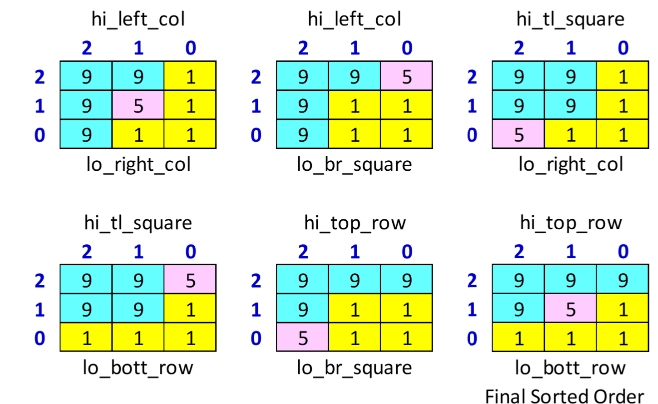


FIGURE 4. Six 3 × 3 median RC-sorted rectangles.

match the final sorted rectangle order, and this is the rectangle shown to the right in Fig. 3.

A 1-D vector is associated with each rectangle, and the vector is constructed using the final rectangle sorted order. In the system described in [6], the rectangle order is a row major order, with the max of each row at the left row edge, and the max column value at the highest row. Each Fig. 3 rectangle has its 9-value vector shown below the rectangle. The Fig. 3 rectangle colors and the hi/lo labels are discussed below.

When targeting specific locations in the sorted output list, even fewer RC-sorted validation rectangles can be constructed and processed with the final stages of the sorting process. For example, using 3×3 rectangles, all possible RC-sorted rectangles with 4 9s, 1 5, and 4 1s, are easily constructed. There are 6 such rectangle orderings after the row sort operation, as shown in Fig. 4. The value 5 is the median of the 9 values, and will be at the center of the 3×3 rectangle in the final sorted order, which is the order found in the lower right of the 6 Fig. 4 rectangles.

These six rectangles are constructed in a manner similar to that shown in Algorithms 2 and 3, except that the two algorithms produce rectangles in which values L down to 1 are found once and only once in the rectangle. The Fig. 4 rectangles have only 3 distinct values, 9, 5, and 1.

TABLE 5. L! / RC-sorted Vector Reduction Ratios (log_{10}).

L	Ncols =2	Ncols =3	Ncols =4	Comment
8	3.46		3.46	Reduction Ratio = 2880
9		3.94		Reduction Ratio = 8640
12	6.56	6.02	6.02	Reduction Ratios > 1e6
15		8.34		
16	10.17		8.94	
18	12.12	10.16		

L! Reduction Ratio = L! vectors / equivalent RC-sorted vectors.
L = Total values in rectangle and vector.

Note that the 3 × 3 rectangle median value is always found along a diagonal from the lower left to the upper right rectangle locations, and the overall rectangle median is equal to the median of the 3 values along this diagonal. In the next sorting network stage, this diagonal is sorted, and the median value 5 is then placed in the 3 × 3 rectangle middle location, for each of the 6 RC-sorted rectangles.

In Figs. 3 and 4, the blue color is used to highlight the cells where the values are greater than the median value of 5. Pink is used for the median 5 cell, and yellow is used for the cells in which the values are less than 5. The shapes enclosing the four hi cells, the cells with values > 5, are labelled hi_left_col when the left column has only hi values, hi_tl_square when the hi values form a 2×2 square in the top left of the rectangle, and hi_top_row when the top row contains only hi values. The three lo labels are defined in a similar manner.

In the video accompanying this work, the construction steps of all six Fig. 4 rectangles are shown, as well as the steps in the construction of all 42 Fig. 3-style rectangles in which the values 9 down to 1 occur only once. The coloring of the set of 42 rectangles in the video shows that each one falls into one of the six Fig. 4 color patterns. More information concerning the construction of these small rectangle/vector sets is found in the supplemental material associated with this work.

The ratio of the 9!=362, 880 9-sorter permutation vectors to the comparable 42 RC-sorted vectors, 8640, is here called the 9! RC-sorted vector reduction ratio for the 3 × 3 rectangles. Additional L! vector reduction ratios are shown in Table 5, listed in $\log_{10}$ format. These values clearly increase dramatically as L increases. For higher L values, it is also clear that Ncols=2 has the highest vector reduction ratios.

For the 3 × 3 9! vector sets, as well as for the 12! vector sets with Ncols=2,3,4 listed Table 5, analyses were run using the full unsorted L! vector set, then dividing each vector in Ncols equal lists, sorting each list, constructing rectangles with each sorted list as a column, and then sorting the rectangle rows. In each case, the unique RC-sorted rectangles produced by this comprehensive process were identical to the RC-sorted rectangles quickly produced by the **create_all_RC-sorted_rectangles** program.

The accompanying supplemental material and video continue the **create_all_RC-sorted_rectangles** discussion and

TABLE 6. Creating 3-sorter Vectors with 1 Distinct Value.

#	2 (VecLocs)	1 (VecLocs)	0 (VecLocs)	Comment
	1			VecLoc=2; D_val=1; locs_remain[2][1]=2
	1	1		VecLoc=1; D_val=1; locs_remain[1][1]=1
1	1	1	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1

distinct_orderings_J_in_N 1 3
locs_remain[1][1]=1 means that, when going from VecLoc 1 to VecLoc 0, D_val 1 can be written to 1 remaining location

TABLE 7. Building 3-sorter Vectors with 2 Distinct Values.

#	2 (VecLocs)	1 (VecLocs)	0 (VecLocs)	Comment
	2			VecLoc=2; D_val=2; locs_remain[2][2,1]=1,2
	2	2		VecLoc=1; D_val=2; locs_remain[1][2,1]=0,1
	2	2	2	VecLoc=0; **INV**; D_val=2; locs_remain[1][2]=0
1	2	2	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1
	2	1		VecLoc=1; D_val=1; locs_remain[1][2,1]=1,1
2	2	1	2	VecLoc=0; **OUT**; D_val=2; locs_remain[1][2]=1
3	2	1	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1
	1			VecLoc=2; D_val=1; locs_remain[2][2,1]=2,1
	1	2		VecLoc=1; D_val=2; locs_remain[1][2,1]=1,1
4	1	2	2	VecLoc=0; **OUT**; D_val=2; locs_remain[1][2]=1
5	1	2	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1
	1	1		VecLoc=1; D_val=1; locs_remain[1][2,1]=1,0
6	1	1	2	VecLoc=0; **OUT**; D_val=2; locs_remain[1][2]=1
	1	1	1	VecLoc=0; **INV**; D_val=1; locs_remain[1][1]=0

distinct_orderings_J_in_N 2 3
locs_remain[1][2]=0 means that, when going from VecLoc 1 to VecLoc 0, D_val 2 can be written to 0 remaining locations.
That is, D_val 2 is not allowed to be written to the vector at VecLoc 0.

show how the program directly creates RC-sorted rectangles, including the 6 Fig. 4 rectangles, and the 42 rectangle set from which the two rectangles in Fig. 3 were taken. The supplemental material also shows that the 3 × 3 2^9=512 0-1 (really 1-2) vectors are reduced to 20 RC-sorted rectangles.

VI. CONCLUSION

The 0-1 principle states that a traditional L-input sorting network can be fully validated with the 2^L set of unique input vectors containing only the values 0 and 1. It is stressed here that this is due to the fact that 0-1 vectors fully verify 2-sorters, the only hardware sorting devices utilized in the networks.

It is easily shown that 0-1 vectors cannot be used to fully verify single-stage N-sorters with N>2, using the smallest such N-sorter, a 3-sorter, as an example. Therefore, sorting networks which use these N-sorters cannot be validated using 0-1 vectors.

The 1-N principle is introduced here in order to define the distinct ordering vectors which are necessary and sufficient to verify N-sorter operation. An algorithm and associated software program have been created in order to build the distinct ordering verification vectors for an N-sorter.

An N^N vector set also fully verifies an N-sorter, but this comprehensive vector set has many duplicate orderings, which are not needed for verification. The N^N vector set

TABLE 8. Building 3-sorter Vectors with 3 Distinct Values.

	VecLocs			
#	2	1	0	Comment
	3			VecLoc=2; D_val=3; locs_remain[2][3,2,1]=0,1,1
	3	3		VecLoc=1; **INV**; D_val=3; locs_remain[2][3]=0
	3	2		VecLoc=1; D_val=2; locs_remain[1][3,2,1]=0,0,1
	3	2	3	VecLoc=0; **INV**; D_val=3; locs_remain[1][3]=0
	3	2	2	VecLoc=0; **INV**; D_val=2; locs_remain[1][2]=0
1	3	2	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1
	3	1		VecLoc=1; D_val=1; locs_remain[1][3,2,1]=0,1,0
	3	1	3	VecLoc=0; **INV**; D_val=3; locs_remain[1][3]=0
2	3	1	2	VecLoc=0; **OUT**; D_val=2; locs_remain[1][2]=1
	3	1	1	VecLoc=0; **INV**; D_val=1; locs_remain[1][1]=0
	2			VecLoc=2; D_val=2; locs_remain[2][3,2,1]=1,0,1
	2	3		VecLoc=1; D_val=3; locs_remain[1][3,2,1]=0,0,1
	2	3	3	VecLoc=0; **INV**; D_val=3; locs_remain[1][3]=0
	2	3	2	VecLoc=0; **INV**; D_val=2; locs_remain[1][2]=0
3	2	3	1	VecLoc=0; **OUT**; D_val=1; locs_remain[1][1]=1
	2	2		VecLoc=1; **INV**; D_val=2; locs_remain[2][2]=0
	2	1		VecLoc=1; D_val=1; locs_remain[1][3,2,1]=1,0,0
4	2	1	3	VecLoc=0; **OUT**; D_val=3; locs_remain[1][3]=1
	2	1	2	VecLoc=0; **INV**; D_val=2; locs_remain[1][2]=0
	2	1	1	VecLoc=0; **INV**; D_val=1; locs_remain[1][1]=0
	1			VecLoc=2; D_val=1; locs_remain[2][3,2,1]=1,1,0
	1	3		VecLoc=1; D_val=3; locs_remain[1][3,2,1]=0,1,0
	1	3	3	VecLoc=0; **INV**; D_val=3; locs_remain[1][3]=0
5	1	3	2	VecLoc=0; **OUT**; D_val=2; locs_remain[1][2]=1
	1	3	1	VecLoc=0; **INV**; D_val=1; locs_remain[1][1]=0
	1	2		VecLoc=1; D_val=2; locs_remain[1][3,2,1]=1,0,0
6	1	2	3	VecLoc=0; **OUT**; D_val=3; locs_remain[1][3]=1
	1	2	2	VecLoc=0; **INV**; D_val=2; locs_remain[1][2]=0
	1	2	1	VecLoc=0; **INV**; D_val=1; locs_remain[1][1]=0
	1	1		VecLoc=1; **INV**; D_val=1; locs_remain[2][1]=0

distinct_orderings_J_in_N 3 3
locs_remain[2][3]=0 means that, when going from VecLoc 2 to VecLoc 1, D_val 3 can be written to 0 remaining locations.
That is, D_val 3 is not allowed to be written to the vector at VecLoc 1.

grows very rapidly as N increases, and the number of 10^{10} vectors is almost 100 times the number of distinct ordering vectors produced by the 1-N algorithm for a 10-sorter.

The 1-N principle enables the creation of vector sets that fully verify N-sorters, but unlike the 0-1 principle, the 1-N principle does not enable creation of fairly small vector sets that fully validate sorting networks which use N-sorters. However, for the type of N-sorter sorting networks which the authors have recently defined, RC-sorted rectangles are used to dramatically lower the number of vectors required for network validation.

An algorithm and software program have been defined which quickly produce a full set of RC-sorted rectangles and associated vectors, which have been used to validate many L-input N-sorter sorting networks. The vector reduction ratios for L! unsorted vector sets, versus the equivalent RC-sorted vector sets, are all over 1 million for L=12, and increase dramatically for higher L values.

APPENDIX A
GENERATING THE DISTINCT ORDERING VECTORS FOR A 3-SORTER

The detailed operations for the 3 executions of program **distinct_orderings_J_in_N** listed in Section IV-C are shown in Tables 6 to 8. Each time this program is run, it writes to standard output all distinct ordering vectors that contain J distinct values (values 1 to J) in a vector length N=3.

Each table shows how Algorithm 1 creates the distinct ordering vectors. The operations in the three tables are performed in Algorithm 1's recursive subroutine SET_VEC_LOC. The loop variable **distinct_value** in SET_VEC_LOC is shortened in the tables to **D_val**.

For each location VecLoc in the vector to be created, each D_val is tested to see whether a "distinct ordering vector still possible" if D_val becomes the vector value at VecLoc. In the tables, the global two-dimensional array variable **locs_remain** is used to answer the "still possible" question.

A **locs_remain[VecLoc][D_val]** value indicates how many remaining vector locations D_val can be written to, after a distinct value has been specified at location VecLoc. The locs_remain values from location VecLoc are used at the next location, VecLoc−1. If the locs_remain value is 0, D_val is not allowed to be written to the vector at location VecLoc-1, as this would produce an invalid distinct ordering vector. If the locs_remain value is > 0, D_val will be written to the vector at the VecLoc-1 location.

The blue **OUT** in the Comment section indicates that a distinct ordering vector, highlighted in light blue, has been created by adding an allowed value at VecLoc 0. The vector is then sent to the program's standard output.

A red **INV** indicates that D_val, also in red and highlighted in yellow, will produce an invalid distinct ordering vector if it is added to the vector at this VecLoc. The program therefore bypasses D_val at this VecLoc, and moves on.

When there is only 1 distinct value to be written to the vector, Table 6 shows that the program very quickly creates an all-1s vector. Table 7 lists the program steps when there are 2 distinct values in the vector input list, values 1 and 2, each of which must occur at least once in each vector's 3 total values.

When J=N, as in Table 8, each value from 1 to N must occur once and only once in the vector list. This is the N! permutation vector list discussed in the main text.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison Wesley, 1973.

[2] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, 1st ed. Cambridge, MA, USA: MIT Press, 1990.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[4] R. B. Kent and M. S. Pattichis, "Design, implementation, and analysis of high-speed single-stage N-sorters and N-filters," *IEEE Access*, vol. 9, pp. 2576–2591, 2021.

[5] R. Kent and M. Pattichis, "Use of carry chain logic and design system extensions to construct significantly faster and larger single-stage N-sorters and N-filters," *IEEE Access*, vol. 10, pp. 79689–79702, 2022.

[6] R. B. Kent and M. S. Pattichis, "Design of high-speed multiway merge sorting networks using fast single-stage N-sorters and N-filters," *IEEE Access*, vol. 10, pp. 77980–77992, 2022.

[7] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comput. Conf.*, 1968, pp. 307–314, doi: 10.1145/1468075.1468121.

[8] D. C. Van Voorhis, "A generalization of the divide-sort-merge strategy for sorting networks," Stanford Electron. Labs., Stanford Univ., Stanford, CA, USA, Tech. Rep. 16, Aug. 1971.

[9] D. C. Van Voorhis, "An economical construction for sorting networks," in *Proc. Nat. Comput. Conf. Expo.*, 1974, pp. 921–927, doi: 10.1145/1500175.1500347.

[10] G. Schnitger, *Parallel and Distributed Algorithms*. Frankfurt, Germany: Institut Für Informatik. 2006.

[11] H. Casanova, A. Legrand, and Y. Robert, *Parallel Algorithms*, 1st ed. Boca Raton, FL, USA: CRC Press, 2008, doi: 10.1201/9781584889465.

[12] J. R. Smith, *The Design and Analysis of Parallel Algorithms*. New York, NY, USA: Oxford Univ. Press, 1993.

[13] G. Stachowiak, "Fast periodic correction networks," *Theor. Comput. Sci.*, vol. 354, no. 3, pp. 354–366, Apr. 2006.

[14] S.-S. Choi and B.-R. Moon, "A graph-based Lamarckian–Baldwinian hybrid for the sorting network problem," *IEEE Trans. Evol. Comput.*, vol. 9, no. 1, pp. 105–114, Feb. 2005, doi: 10.1109/TEVC.2004.841682.

[15] S. W. Baddar and K. E. Batcher, *Designing Sorting Networks: A New Paradigm*. Berlin, Germany: Springer, 2012.

[16] Z. Vasicek and V. Mrazek, "Trading between quality and non-functional properties of median filter in embedded systems," *Genetic Program. Evolvable Mach.*, vol. 18, no. 1, pp. 45–82, Mar. 2017.

[17] M. Bidlo and M. Dobes, "Evolutionary development of growing generic sorting networks by means of rewriting systems," *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 232–244, Apr. 2020, doi: 10.1109/TEVC.2019.2918212.

[18] C. Chakrabarti, "Sorting network based architectures for median filters," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 40, no. 11, pp. 723–727, Nov. 1993, doi: 10.1109/82.251840.

[19] Y. Azar and Y. Richter, "The zero-one principle for switching networks," in *Proc. 36th Annu. ACM Symp. Theory Comput.*, New York, NY, USA, Jun. 2004, pp. 64–71, doi: 10.1145/1007352.1007369.

[20] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3, 2nd ed. Boston, MA, USA: Addison Wesley, Sep. 2017.

[21] M. Ajtai, J. Komlos, and E. Szemeredi, "Halvers and expanders (switching)," in *Proc. 33rd Annu. Symp. Found. Comput. Sci.*, Oct. 1992, pp. 686–692, doi: 10.1109/SFCS.1992.267782.

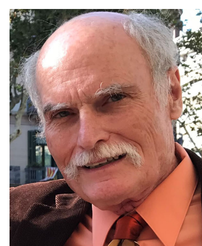
[22] B. Parker and I. Parberry, "Constructing sorting networks from k-sorters," *Inf. Process. Lett.*, vol. 33, no. 3, pp. 157–162, 1989, doi: 10.1016/0020-0190(89)90196-8.

[23] L. Zhao, Z. Liu, and Q. Gao, "An efficient multiway merging algorithm," *Sci. China Ser. E, Technol. Sci.*, vol. 41, no. 5, pp. 543–551, Oct. 1998.

[24] F. Shi, Z. Yan, and M. Wagh, "An enhanced multiway sorting network based on n-sorters," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2014, pp. 60–64.

[25] R. Cypher and J. L. C. Sanz, "Cubesort: A parallel algorithm for sorting N data items with S-sorters," *J. Algorithms*, vol. 13, no. 2, pp. 211–234, 1992, doi: 10.1016/0196-6774(92)90016-6.

[26] D. Gale and R. M. Karp, "A phenomenon in the theory of sorting," in *Proc. 11th Annu. Symp. Switching Automata Theory*, Oct. 1970, pp. 51–59, doi: 10.1109/SWAT.1970.1.



ROBERT B. KENT (Life Member, IEEE) received the B.S. degree in physics from the University of Notre Dame, Notre Dame, IN, USA, in 1970, and the M.S. degree in electrical engineering from The University of Utah, Salt Lake City, UT, USA, in 1983. He is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, The University of New Mexico.

He was with various semiconductor companies: National Semiconductor, from 1983 to 1990, Intel Corporation, from 1990 to 1998, Philips Semiconductor, from 1998 to 1999, and Xilinx Inc., from 1999 to 2011. He was an Independent Contractor, also in the semiconductor field, from 2012 to 2017. His main research interests include the design of single-stage N-sorters and N-filters in hardware, particularly in FPGAs, and the use of these sorters and filters in sorting networks or other hardware sorting systems.

MARIOS S. PATTICHIS (Senior Member, IEEE) received the B.Sc. degree (Hons.) in computer sciences, and the Bachelor of Arts degree (Hons.) in mathematics and a minor in electrical engineering, the M.S. degree in electrical engineering, and the Ph.D. degree in computer engineering from The University of Texas at Austin, in 1991, 1993, and 1998, respectively.

He was the lead PI and a Board Member of the Configurable Space and Microsystems Innovations and Applications Center (COSMIAC), The University of New Mexico (UNM). At UNM, he is also the Director of the Image and Video Processing and Communications Laboratory (ivPCL). He is currently a Professor with the Department of Electrical and Computer Engineering, UNM. His current research interests include image and video processing, video communications, dynamically reconfigurable computer architectures, and biomedical image analysis.

Dr. Pattichis was a fellow of the Center for Collaborative Research and Community Engagement, UNM College of Education, from 2019 to 2020. He was elected as a fellow of the European Alliance of Medical and Biological Engineering and Science (EAMBES). He was a recipient of the 2016 Lawton-Ellis and the 2004 Distinguished Teaching Awards from the Department of Electrical and Computer Engineering, UNM. For his development of the digital logic design laboratories with UNM, he was recognized by Xilinx Corporation, in 2003, and by the UNM School of Engineering's Harrison Faculty Excellence Award, in 2006. He was the General Chair of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI) and the General Co-Chair of the SSIAI, in 2020. He has served as a Senior Associate Editor for the IEEE Transactions on Image Processing and IEEE Signal Processing Letters, an Associate Editor for IEEE Transactions on Image Processing and IEEE Transactions on Industrial Informatics, and a Guest Associate Editor for two additional special issues published in the IEEE Transactions on Information Technology in Biomedicine, a Special Issue published by *Teachers College Record*, a Special Issue published by the IEEE Journal of Biomedical and Health Informatics, and a Special Issue published in *Biomedical Signal Processing and Control*.