Chapter 9

Sorting Networks

By Sariel Har-Peled, December 2, 2021[1]

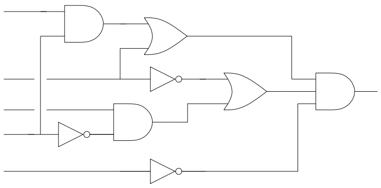
> The world is what it is; men who are nothing, who allow themselves to become nothing, have no place in it.
>
> A bend in the river, V. S. Naipul

9.1. Model of Computation

It is natural to ask if one can perform a computational task considerably faster by using a different architecture (i.e., a different computational model).

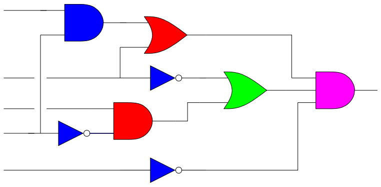
The answer to this question is a resounding yes. A cute example is the ***Macaroni sort*** algorithm. We are given a set $S = \{s_1, \ldots, s_n\}$ of n real numbers in the range (say) $[1, 2]$. We get a lot of Macaroni (this are longish and very narrow tubes of pasta), and cut the ith piece to be of length s_i, for $i = 1, \ldots, n$. Next, take all these pieces of pasta in your hand, make them stand up vertically, with their bottom end lying on a horizontal surface. Next, lower your handle till it hit the first (i.e., tallest) piece of pasta. Take it out, measure it height, write down its number, and continue in this fashion till you have extracted all the pieces of pasta. Clearly, this is a sorting algorithm that works in linear time. But we know that sorting takes $\Omega(n \log n)$ time. Thus, this algorithm is much faster than the standard sorting algorithms.

This faster algorithm was achieved by changing the computation model. We allowed new "strange" operations (cutting a piece of pasta into a certain length, picking the longest one in constant time, and measuring the length of a pasta piece in constant time). Using these operations we can sort in linear time.



If this was all we can do with this approach, that would have only been a curiosity. However, interestingly enough, there are natural computation models which are considerably stronger than the standard model of computation. Indeed, consider the task of computing the output of the circuit on the right (here, the input is boolean values on the input wires on the left, and the output is the single output on the right).

Clearly, this can be solved by ordering the gates in the "right" order (this can be done by topological sorting), and then computing the value of the gates one by one in this order, in such a way that a gate being computed knows the values arriving on its input wires. For the circuit above, this would require 8 units of time, since there are 8 gates.



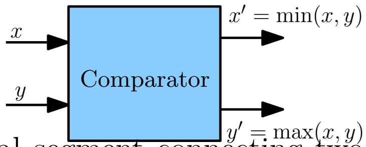
However, if you consider this circuit more carefully, one realized that we can compute this circuit in 4 time units. By using the fact that several gates are independent of each other, and we can compute them in parallel, as depicted on the right. Furthermore, circuits are inherently parallel and we should be able to take advantage of this fact.

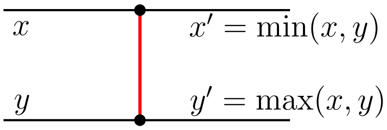
[1] This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

So, let us consider the classical problem of sorting n numbers. The question is whether we can sort them in *sublinear* time by allowing parallel comparisons. To this end, we need to precisely define our computation model.

9.2. Sorting with a circuit – a naive solution

We are going to design a circuit, where the inputs are the numbers and we compare two numbers using a comparator gate. Such a gate has two inputs and two outputs, and it is depicted on the right.

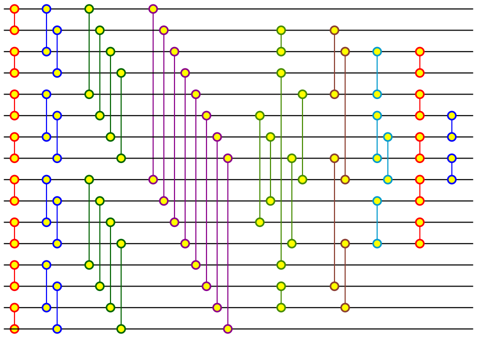




We usually depict such a gate as a vertical segment connecting two wires, as depicted on the right. This would make drawing and arguing about sorting networks easier.

Our circuits would be depicted by horizontal lines, with vertical segments (i.e., gates) connecting between them. For example, see complete sorting network depicted on the right.

The inputs come on the wires on the left, and are output on the wires on the right. The largest number is output on the bottom line. Somewhat surprisingly, one can generate circuits from known sorting algorithms.



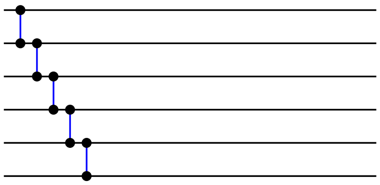
9.2.1. Definitions

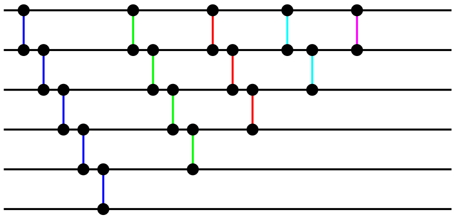
Definition 9.2.1. A ***comparison network*** is a DAG (directed acyclic graph), with n inputs and n outputs, where each gate (i.e., done) has two inputs and two outputs (i.e., two incoming edges, and two outgoing edges).

Definition 9.2.2. The ***depth*** of a wire is 0 at the input. For a gate with two inputs of depth d_1 and d_2 the depth on the output wire is $1 + \max(d_1, d_2)$. The ***depth*** of a comparison network is the maximum depth of an output wire.

Definition 9.2.3. A ***sorting network*** is a comparison network such that for any input, the output is monotonically sorted. The ***size*** of a sorting network is the number of gates in the sorting network. The ***running time*** of a sorting network is just its depth.

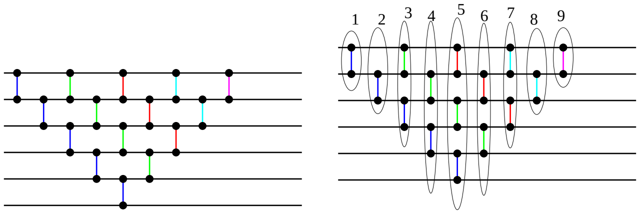
9.2.2. Sorting network based on insertion sort





Consider the sorting circuit on the left. Clearly, this is just the inner loop of the standard insertion sort. As such, if we repeat this loop, we get the sorting network on the right. It is easy to argue that this circuit sorts correctly all inputs (we removed some unnecessary gates).

An alternative way of drawing this sorting network is depicted in Figure 9.1 (ii). The next natural question, is how much time does it take for this circuit to sort the n numbers. Observe, that the running time of the algorithm is how many different time ticks we have to wait till the result stabilizes in all the gates. In our example, the alternative drawing immediately tell us how to schedule the computation of the gates. See Figure 9.1 (ii).



(i) (ii)

Figure 9.1: The sorting network inspired by insertion sort.

In particular, the above discussion implies the following result.

Lemma 9.2.4. *The sorting network based on insertion sort has $O(n^2)$ gates, and requires $2n - 1$ time units to sort n numbers.*

9.3. The Zero-One Principle

The ***zero-one principle*** states that if a comparison network sort correctly all binary inputs (i.e., every number is either 0 or 1) then it sorts correctly all inputs. We (of course) need to prove that the zero-one principle is true.

Lemma 9.3.1. *If a comparison network transforms the input sequence $a = \langle a_1, a_2, \ldots, a_n \rangle$ into the output sequence $b = \langle b_1, b_2, \ldots, b_n \rangle$, then for any monotonically increasing function f, the network transforms the input sequence $f(a) = \langle f(a_1), \ldots, f(a_n) \rangle$ into the sequence $f(b) = \langle f(b_1), \ldots, f(b_n) \rangle$.*

Proof: Consider a single comparator with inputs x and y, and outputs $x' = \min(x, y)$ and $y' = \max(x, y)$. If $f(x) = f(y)$ then the claim trivially holds for this comparator. If $f(x) < f(y)$ then clearly

$$\max\bigl(f(x), f(y)\bigr) = f\bigl(\max(x, y)\bigr) \text{ and}$$
$$\min\bigl(f(x), f(y)\bigr) = f\bigl(\min(x, y)\bigr),$$

since $f(\cdot)$ is monotonically increasing. As such, for the input $\langle x, y \rangle$, for $x < y$, we have output $\langle x, y \rangle$. Thus, for the input $\langle f(x), f(y) \rangle$ the output is $\langle f(x), f(y) \rangle$. Similarly, if $x > y$, the output is $\langle y, x \rangle$. In this case, for the input $\langle f(x), f(y) \rangle$ the output is $\langle f(y), f(x) \rangle$. This establish the claim for a single comparator.

Now, we claim by induction that if a wire carry a value a_i, when the sorting network get input $a_1, \ldots, a_n$, then for the input $f(a_1), \ldots, f(a_n)$ this wire would carry the value $f(a_i)$.

This is proven by induction on the depth on the wire at each point. If the point has depth 0, then its an input and the claim trivially hold. So, assume it holds for all points in our circuits of depth at most i, and consider a point p on a wire of depth $i + 1$. Let G be the gate which this wire is an output of. By induction, we know the claim holds for the inputs of G (which have depth at most i). Now, we the claim holds for the gate G itself, which implies the claim apply the above claim to the gate G, which implies the claim holds at p. ∎

Theorem 9.3.2. *If a comparison network with n inputs sorts all 2^n binary strings of length n correctly, then it sorts all sequences correctly.*

Proof: Assume for the sake of contradiction, that it sorts incorrectly the sequence $a_1, \ldots, a_n$. Let $b_1, \ldots b_n$ be the output sequence for this input.

Let $a_i < a_k$ be the two numbers that are output in incorrect order (i.e. a_k appears before a_i in the output). Let

$$f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i. \end{cases}$$

Clearly, by the above lemma (Lemma 9.3.1), for the input

$$\langle f(a_1), \ldots, f(a_n) \rangle,$$

which is a binary sequence, the circuit would output $\langle f(b_1), \ldots, f(b_n) \rangle$. But then, this sequence looks like

$$000..0????f(a_k)????f(a_i)??1111$$

but $f(a_i) = 0$ and $f(a_j) = 1$. Namely, the output is a sequence of the form ????1????0????, which is not sorted.

Namely, we have a binary input (i.e., $\langle f(b_1), \ldots, f(b_n) \rangle$) for which the comparison network does not sort it correctly. A contradiction to our assumption. ■

9.4. A bitonic sorting network

Definition 9.4.1. A ***bitonic sequence*** is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.

Example 9.4.2. The sequences $(1, 2, 3, \pi, 4, 5, 4, 3, 2, 1)$ and $(4, 5, 4, 3, 2, 1, 1, 2, 3)$ are bitonic, while the sequence $(1, 2, 1, 2)$ is not bitonic.

Observation 9.4.3. *A binary bitonic sequence (i.e., bitonic sequence made out only of zeroes and ones) is either of the form $0^i 1^j 0^k$ or of the form $1^i 0^j 1^k$, where 0^i (resp, 1^i) denote a sequence of i zeros (resp., ones).*

Definition 9.4.4. A ***bitonic sorter*** is a comparison network that sorts all bitonic sequences correctly.

Definition 9.4.5. A ***half-cleaner*** is a comparison network, connecting line i with line $i+n/2$. In particular, let Half-Cleaner$[n]$ denote the half-cleaner with n inputs. Note, that the depth of a Half-Cleaner$[n]$ is one, see Figure 9.2.

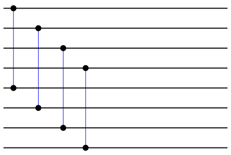


Figure 9.2

It is beneficial to consider what a half-cleaner do to an input which is a (binary) bitonic sequence. Clearly, in the specific example depicted in Figure **??**, we have that the left half size is clean and all equal to 0. Similarly, the right size of the output is bitonic.

Specifically, one can prove by simple (but tedious) case analysis that the following lemma holds.

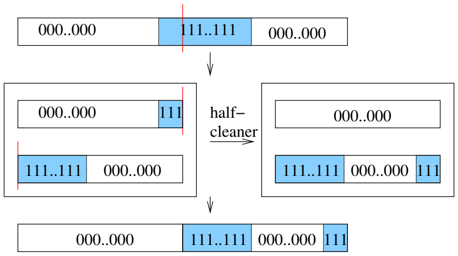


Figure 9.3

00000000	00000000	00011111	00011111	10000001	00001001		
00000001	00000001	00100000	00000010	10000011	00001011	11100111	01101111
00000010	00000010	00110000	00000011	10000111	00001111	11101111	11101111
00000011	00000011	00111000	00001011	10001111	10001111	11110000	00001111
00000100	00000100	00111100	00001111	10011111	10011111	11110001	00011111
00000110	00000110	00111110	00101111	10111111	10111111	11110011	00111111
00000111	00000111	00111111	00111111	11000000	00001100	11110111	01111111
00001000	00001000	01000000	00000100	11000001	00001101	11111000	10001111
00001100	00001100	01100000	00000110	11000011	00001111	11111001	10011111
00001110	00001110	01110000	00000111	11000111	01001111	11111011	10111111
00001111	00001111	01111000	00001111	11001111	11001111	11111100	11001111
00010000	00000001	01111100	01001111	11011111	11011111	11111101	11011111
00011000	00001001	01111110	01101111	11100000	00001110	11111110	11101111
00011100	00001101	01111111	01111111	11100001	00001111	11111111	11111111
00011110	00001111	10000000	00001000	11100011	00101111		

Figure 9.4: All bitonic sequences of length 8, and what the half-cleaner does to them.

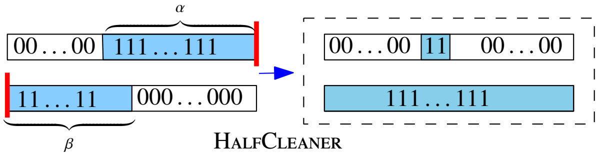
Lemma 9.4.6. *If the input to a half-cleaner (of size n) is a binary bitonic sequence then for the output sequence we have that*

(i) the elements in the top half are smaller than the elements in bottom half, and

(ii) one of the halves is clean, and the other is bitonic.

Proof: If the sequence is of the form $0^i1^j0^k$ and the block of ones is completely on the left side (i.e., its part of the first $n/2$ bits) or the right side, the claim trivially holds. So, assume that the block of ones starts at position $n/2 - \beta$ and ends at $n/2 + \alpha$.

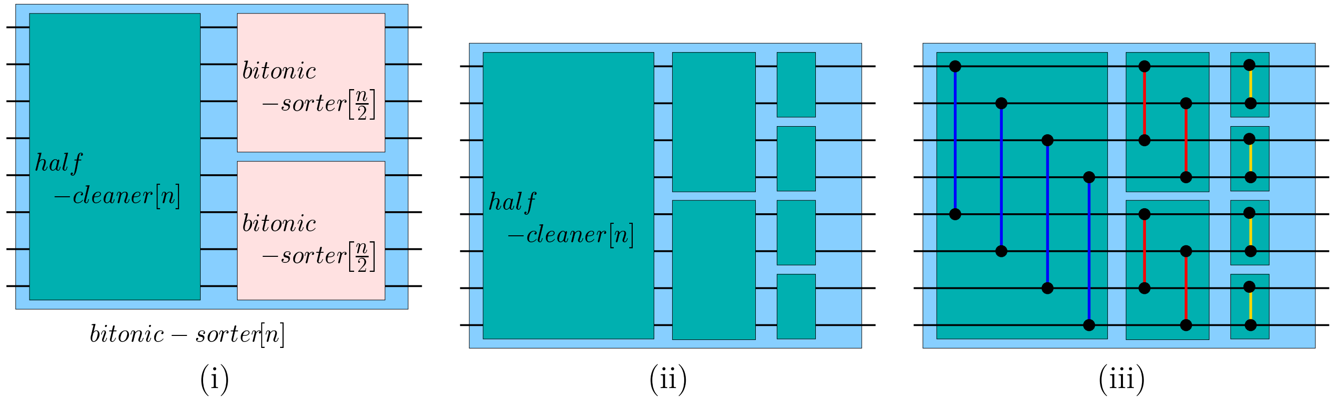
If $n/2 - \alpha \geq \beta$ then this is exactly the case depicted above and claim holds. If $n/2 - \alpha < \beta$ then the second half is going to be all ones, as depicted on the right. Implying the claim for this case.



A similar analysis holds if the sequence is of the form $1^i0^j1^k$. ■

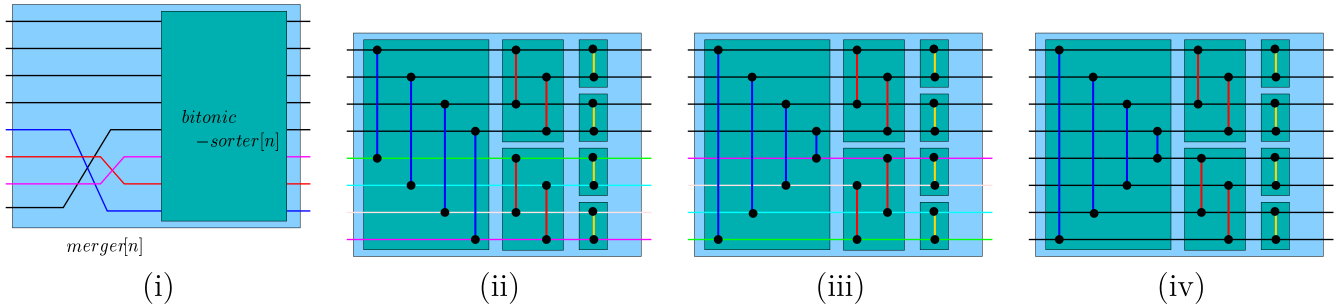
This suggests a simple recursive construction of BitonicSorter$[n]$, see Figure 9.5, and we have the following lemma.

Lemma 9.4.7. *BitonicSorter*$[n]$ *sorts bitonic sequences of length* $n = 2^k$*, it uses* $(n/2)k = (n/2)\lg n$ *gates, and it is of depth* $k = \lg n$*.*



(i) (ii) (iii)

Figure 9.5: Depicted are the (i) recursive construction of BitonicSorter$[n]$, (ii) opening up the recursive construction, and (iii) the resulting comparison network.



(i) (ii) (iii) (iv)

Figure 9.6: (i) Merger via flipping the lines of bitonic sorter. (ii) A BitonicSorter. (ii) The Merger after we “physically” flip the lines, and (iv) An equivalent drawing of the resulting Merger.

9.4.1. Merging sequence

Next, we deal with the following merging question. Given two *sorted* sequences of length $n/2$, how do we merge them into a single sorted sequence?

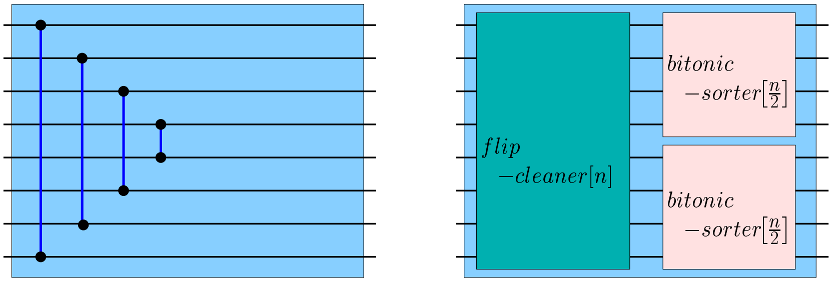
The idea here is concatenate the two sequences, where the second sequence is being flipped (i.e., reversed). It is easy to verify that the resulting sequence is bitonic, and as such we can sort it using the *BitonicSorter*$[n]$.

Specifically, given two sorted sequences $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$, observe that the sequence $a_1, a_2, \ldots, a_n, b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b_1$ is bitonic.

Thus, to merge two sorted sequences of length $n/2$, just flip one of them, and use *BitonicSorter*$[n]$, see Figure 9.6. This is of course illegal, and as such we take *BitonicSorter*$[n]$ and physically flip the last $n/2$ entries. The process is depicted in Figure 9.6. The resulting circuit Merger takes two sorted sequences of length $n/2$, and return a sorted sequence of length n.

It is somewhat more convenient to describe the Merger using a FlipCleaner component. See Figure 9.7

Lemma 9.4.8. *The circuit Merger$[n]$ gets as input two sorted sequences of length $n/2 = 2^{k-1}$, it uses $(n/2)k = (n/2)\lg n$ gates, and it is of depth $k = \lg n$, and it outputs a sorted sequence.*



(i) (ii)

Figure 9.7: (i) FlipCleaner$[n]$, and (ii) Merger$[n]$ described using FlipCleaner.

9.5. Sorting Network

We are now in the stage, where we can build a sorting network. To this end, we just implement ***merge sort*** using the *Merger*$[n]$ component. The resulting component Sorter$[n]$ is depicted in Figure 9.8 using a recursive construction.

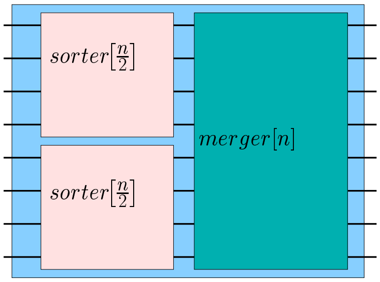


Figure 9.8

Lemma 9.5.1. *The circuit Sorter$[n]$ is a sorting network (i.e., it sorts any n numbers) using $G(n) = O(n \log^2 n)$ gates. It has depth $O(\log^2 n)$. Namely, Sorter$[n]$ sorts n numbers in $O(\log^2 n)$ time.*

Proof: The number of gates is

$$G(n) = 2G(n/2) + Gates(Merger[n]).$$

Which is $G(n) = 2G(n/2) + O(n \log n) = O(n \log^2 n)$.

As for the depth, we have that $D(n) = D(n/2) + \text{Depth}(Merger[n]) = D(n/2) + O(\log(n))$, and thus $D(n) = O(\log^2 n)$, as claimed. ■

9.6. Faster sorting networks

One can build a sorting network of logarithmic depth (see [AKS83]). The construction however is very complicated. A simpler parallel algorithm would be discussed sometime in the next lectures. BTW, the AKS construction [AKS83] mentioned above, is better than bitonic sort for n larger than 2^{8046}.

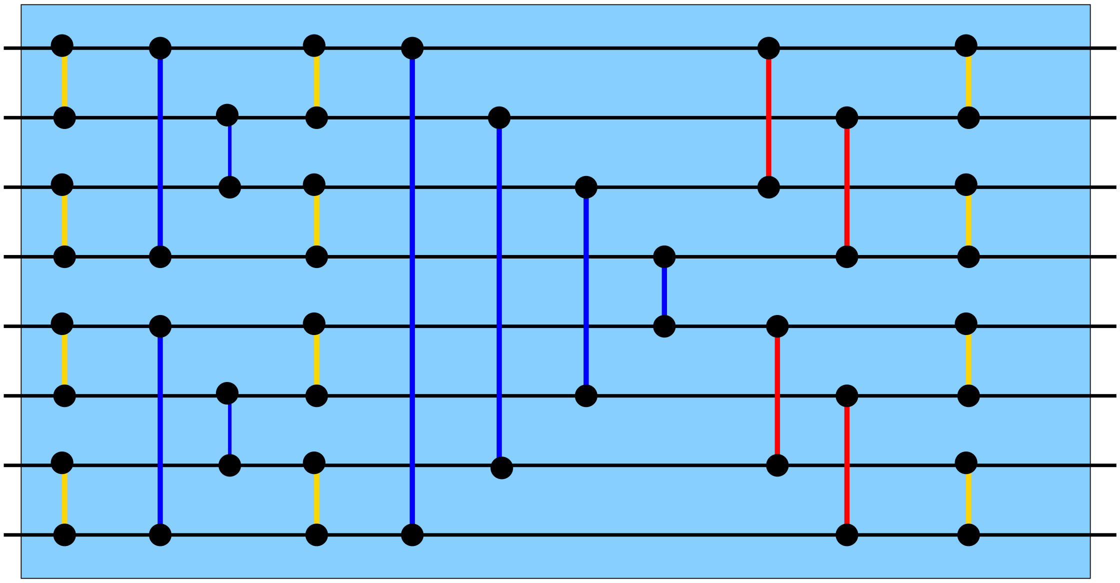


Figure 9.9: The resulting sorting network for input of size n: $\mathit{Sorter}[8]$.

Bibliography

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. 15th Annu. ACM Sympos. Theory Comput.* (STOC), pages 1–9, 1983.