Principles of Distributed Computing
Sample Solution to Exercise 8

1 Sorting Networks

We will assume that a comparator outputs the smaller value on the top and the larger value on the bottom wire.

a) Wrong. Consider the following input, from top to bottom: $I = (0, 0, 1, 0, 0, 0)$ which produces the output $O = (0, 0, 0, 1, 0, 0)$. This is obviously not sorted.

b) Correct. A comparator leaves a sorted sequence intact.

c) Correct. A correct sorting network needs to be able to sort *any* input sequence, and a comparator added to the front merely changes the input for the sorting network.

d) Correct. Assume otherwise: There is a correct sorting network S such that there is no comparator between horizontal lines i and $i + 1$. Consider the input sequence I consisting of 0's, then a 1 at the ith spot, a 0 at spot $i+1$, and the rest 1s. Now every comparator will leave I intact, because I is already sorted except for the horizontal lines i and $i + 1$. But since there is no horizontal line, the output to S is the same as the input, $O = I$, which is not sorted. Thus, no such S exists.

e) Wrong. Consider the following network with three horizontal lines depicted in Figure 1. It does not sort $I = (1, 1, 0)$ because the 0 does not have a chance to get to the top horizontal line, so $O = (1, 0, 1)$.

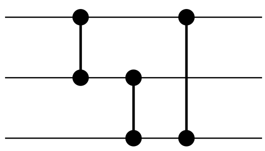

Figure 1: A network with a comparator between each pair of horizontal lines.

f) Wrong. This can be seen in two ways. The first is by a simple counter example as in Figure 2. If we leave out the marked comparator, then we have a correct sorting network (easy to see or otherwise check all 0-1 sequences). Adding it gives us the same problem as the network in Figure 1 above.

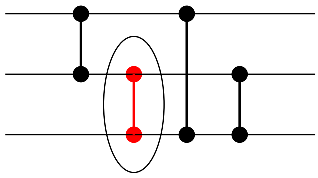

Figure 2: A network with an additional comparator added inside.

Another indication for why this statement is wrong comes from considering Batcher's Sorting Network from the lecture. It relies on creating and sorting bitonic sequences. A comparator can easily destroy a bitonic sequence, such as $(0, 1, 1, 0)$ into $(0, 1, 0, 1)$.

g) Wrong. Consider the sorting network in Figure 3. We can verify that it sorts any input correctly. If we use the network backwards, then the input (1,0,0) is not sorted correctly.

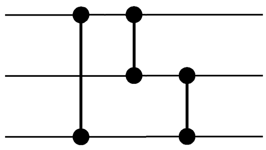

Figure 3: A network that does not sort correctly when fed backwards.

2 Alternative 0-1 Sorting Lemma proof

a) Consider $x \leq y$ and suppose we apply $f(x)$ and $f(y)$ to the inputs of the comparator (the exact order does not matter). Since f is monotonically increasing, $x \leq y$ implies $f(x) \leq f(y)$. The operation of the comparator thus yields the value $\min(f(x), f(y)) = f(x) = f(\min(x, y))$ on the upper output and the value $\max(f(x), f(y)) = f(y) = f(\max(x, y))$ on the lower output. The argument is similar for $y \leq x$.

b) We use induction on the depth of each wire in a general comparison network to prove the following statement: if a wire has the value a_i when the input sequence a is applied to the network, then it has the value $f(a_i)$ when the input sequence $F(a)$ is applied. This holds also for the output wires thus it proves the statement. For the basis, consider a wire at depth 0, that is, an input wire a_i. Obviously, when F is applied before the network, the input wire has the value $f(a_i)$. For the inductive step, consider a wire at depth d, where $d \geq 1$. The wire is the output of a comparator at depth d, and the input wires to this comparator are at a depth strictly less than d. By the inductive hypothesis, if the input wires to the comparator have values a_i and a_j when the input sequence a is applied, then they have $f(a_i)$ and $f(a_j)$ when the input sequence $F(a)$ is applied. From the first question, we know that the output wires of this comparator then have $f(\min(a_i, a_j))$ and $f(\max(a_i, a_j))$, which concludes the proof.

c) We provide a slightly different proof of the 0-1 Sorting Lemma than the one given in the lecture notes. By contradiction, suppose the comparison network sorts all 0-1 sequences, but there exists a sequence of arbitrary numbers that the network does not sort correctly. Let this sequence be $a = (a_1, a_2, \cdots, a_n)$, containing a_i and a_j such that $a_i < a_j$ while the network's output sequence places a_j before a_i. We define the following monotonically increasing function:

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

Since the network's output sequence places a_j before a_i when $a = (a_1, a_2, \cdots, a_n)$ is the input, it follows from the previous question that it places also $f(a_i)$ before $f(a_j)$ in the output sequence when $(f(a_1), f(a_2), \cdots, f(a_n))$ is input. However, $f(a_j) = 1$ and $f(a_i) = 0$, and thus the comparison network fails to sort the 0-1 sequence $(f(a_1), f(a_2), ..., f(a_n))$ correctly. Contradiction.

3 Recursive Sorting Networks

a) Figure 4 displays a solution with $n-1$ comparators.

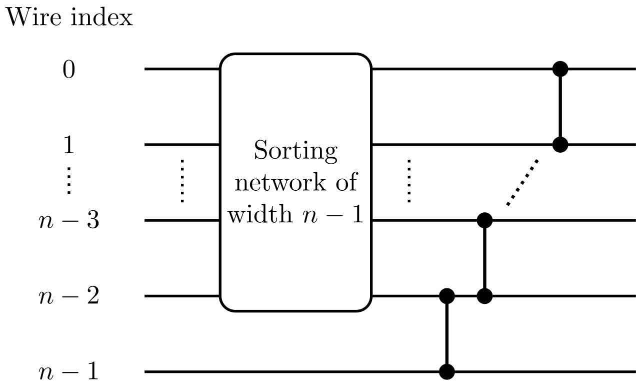

Figure 4: Recursive sorting network.

To prove the correctness of the solution, it is easy to see that the last wire's input will move up incrementally in the sorted list of size $n-1$ (output of the black-box sorting network) until it is compared to a value that is strictly smaller. All the values that are strictly larger will move down one wire and keep their relative order. All the value that are strictly lower will not be affected by the remaining comparators (as they are already sorted). The resulting list is therefore sorted correctly. This solution is not unique. Indeed, the network in Figure 5 is another possible solution.

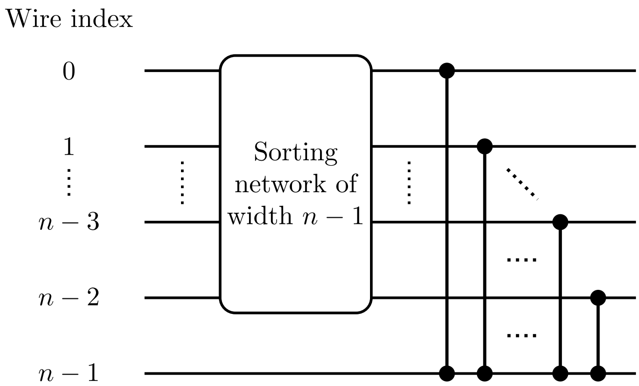

Figure 5: Another valid solution.

b) By contradiction, assume that there exists a solution with $k < n-1$ comparators. Consider the graph where each of the n output wires is a vertex and each of the k comparators is an edge. We know from a previous lecture that this graph has at least 2 connected components (since any connected graph of n vertices has at least $n-1 > k$ edges). Because there are at least two connected components, there exists a wire j that is disconnected from the last wire. Consider any input sequence where the j-th smallest value is unique and enters the network on the last wire (i.e., bypasses the black-box sorting network). This value will be misplaced as it will never reach the j-th output wire, and thus the network cannot sort all input sequences. A contradiction. Note that a similar argument holds if the black-box sorting network is placed on the bottom $n-1$ wires, or if the comparators are inverted (sorting in descending order).

c) It can be shown that Figure 4 is the only possible extension of a sorting network of width $n-1$ that only uses $n-1$ comparators of width 1 (if you have a different solution, please check carefully). Assuming that the black-box sorting network of width $n-1$ in Figure 4 is built recursively, starting with a single comparator linking wires 0 and 1, the resulting algorithm is non-other than a distributed version of *insertion sort*! Indeed, starting with a single value, the algorithm maintains a sorted sub-list, inserting each new value at its correct (temporary) position before processing the value on the next wire.

d) Figure 6 displays a solution with $n-1$ comparators. It is also not unique (the mirror of Figure 5 is another valid solution). To see that it's correct, note that the maximum value of

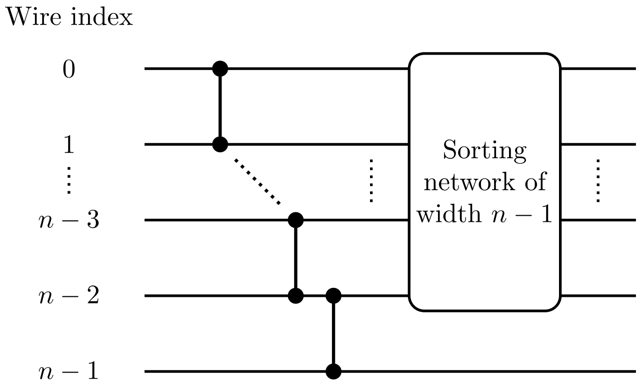

Figure 6: Recursive sorting network (inverted).

the input wires (or one of them if not unique) will be redirected to the last wire, regardless of where it enters the network. The black-box sorting network will output a sorted list with all the remaining values, which concludes the proof. With an argument similar to the one above, we prove that we need at least $n-1$ comparators by contradiction. Assuming that there exists a solution with strictly fewer comparators, we build a similar disconnected graph, and we construct an input sequence where the maximum value is unique and enters the networks on a wire that is disconnected to the last wire. This value will therefore never be sorted correctly. A contradiction. Finally, if the sorting network is build recursively, starting with a single comparator of width 1, the resulting algorithm is a distributed version of *bubble sort*! Indeed, the algorithm iteratively sends the maximum value to the bottom wire, and then leaves that wire untouched while processing the remaining wires.