Sorting with Networks of Data Structures

Therese Biedl [1], Alexander Golynski [1], Angèle M. Hamel [*,2],
Alejandro López-Ortiz [1], J.Ian Munro [1]

Abstract

We consider the problem of sorting a permutation using a network of data structures as introduced by Knuth and Tarjan. In general the model as considered previously was restricted to networks that are directed acyclic graphs (DAGs) of stacks and/or queues. In this paper we study the question of which are the smallest general graphs that can sort an arbitrary permutation and what is their efficiency. We show that certain two node graphs can sort in time $\Theta(n^2)$ and no simpler graph can sort all permutations. We then show that certain three node graphs sort in time $\Omega(n^{3/2})$, and that there exist graphs of k nodes which can sort in time $\Theta(n \log_k n)$, which is optimal.

Key words: sorting networks, data structures
PACS:

1 Introduction

Sorting networks have a long history in computer science. In the early 1970's, Tarjan [40], Knuth [32], Even and Itai [25], and Pratt [35] each explored the idea of using data structures such as stacks, queues, and deques as abstract machines to sort or rearrange input permutations with a goal of obtaining the identity as the output permutation. They further considered connecting these data structures in networks and sorting permutations through these networks of data structures. The area has been rejuvenated by mathematicians who considered a number of special cases, particularly the problem of sorting with two stacks in series. In particular we mention Atkinson [1,3–7], Bóna [11–15], Bousquet-Mélou [18–20], Guibert [27,28], West [24,26,38,43–45] and others [10,17,21,22,29–31,33,34,36,37,39,46].

* Corresponding author.
[1] Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada, N2L 3G1.
[2] Department of Physics and Computer Science, Wilfrid Laurier University, Waterloo, ON, N2L 3C5.

Historically, the model considered directed acyclic graphs (DAGs) of stacks and/or queues, e.g. [6], [44], as well as other structures such as sorted stacks [6], pop stacks [10,7], bounded capacity stacks [5], bounded capacity priority queues [9], generalised stacks [4], and forkstacks [1]. As such, the focus centred on exactly which permutations were sortable. Indeed, the cyclic case was dismissed in a single line in Tarjan [40]: "A circuit in the switchyard will allow us to sort any sequence; thus we do not allow circuits." Thus from a mathematical point of view the central questions have been: 1) given a network of data structures, can we enumerate how many permutations are sortable by the network, and 2) can we characterize which permutations are sortable by this network. The characterization is usually done in terms of pattern avoidance.

From a computer science point of view, two natural questions that arise are, 1) which networks can sort every input permutation, and 2) what is the time complexity of sorting in a network that can sort every input permutation. Further, to this line of study a long standing open problem in this setting is to determine the exact number of stacks connected *in-series* required to sort a sequence of n numbers [Knuth 1998, Section 5.2.4, Problem 20, rated 47]. In this paper we are given the network and investigate first whether it can sort all permutations. Then the question becomes, how efficiently can it do so.

Section 2 introduces the definition of sorting network. Section 3 explores networks with two or three nodes and classifies them according to efficiency. Section 4 looks at sorting with k stacks in series and Section 5 looks at an online model of sorting.

2 Definitions

A *sorting network* is defined as follows: We are given a network—or directed graph—$N = (V, E)$, where V is the set of nodes and E is the set of directed edges. Self-loops (edges from a node back to itself) are allowed, but multiple edges in the same direction are not allowed. One node is the input I and another the output O. Each node is labeled as being either a stack, sorted stack, queue or dequeue. Here, a *sorted stack* is a stack on which the elements have to be in sorted order, with the smallest element on top. A sorting routine then proceeds as follows:

- Initially, the node I contains all elements, in an arbitrary permutation (the *input permutation*).
- One step of the sorting routine consists of the following:
  - · Pick a node v which currently contains at least one element in its data structure.
  - · Remove one element from v's data structure.
  - · Pick an outgoing edge $v \to w$ ($w = v$ is possible if v has a self-loop.)
  - · Add the element to w's data structure.
- Deques allow to add or remove an element at either end, so during a sorting step, if v or w is labelled as a deque, we also must specify where to remove and, respectively, where to add.
- These steps repeat until all elements are on the data structure of node O in sorted order. By sorted order we mean that if all elements are output from the front they would be listed in ascending sorted order.

Figure 1 gives a sorting network for the well-known *Towers of Hanoi* puzzle, where a tower of sorted disks must be moved from the first peg to the third peg, without ever placing a larger disk onto a smaller one. This corresponds to a sorting network with three sorted stacks. There are two variants of the Towers of Hanoi puzzle, depending on whether moves from the first peg to the third peg are allowed or not; we show here the network where they are not allowed. Also note that in this puzzle the input is already in sorted order (and in fact, must be in sorted order since we have a sorted stack as data structure in node I.)

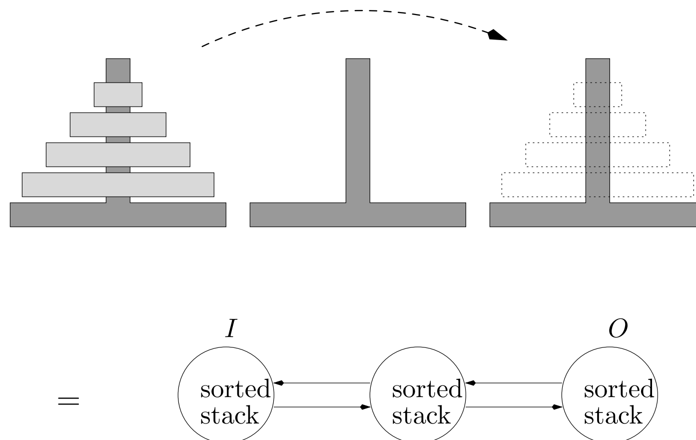

Fig. 1. The Towers of Hanoi puzzle expressed as a sorting network.

3 Classification of Sorting Networks

The natural place to start is with a single node. However, note that only the identity permutation, can be sorted on a network with one node using the allowed data structures. We turn then our focus to all networks of two nodes that are acyclic except for self-loops. The number of such networks to be considered can be reduced by means of a few normalization operations.

- A node labeled neither I nor O must be on a directed path from I to O; otherwise it can be deleted without affecting the sorting power of the network.
- Since the network is acyclic, nodes I and O must be different nodes.
- For a 2-node network we hence have two nodes I and O and a directed edge from I to O. We may or may not have self-loops at I and O.
- If a node labeled as stack or sorted stack has a self-loop, then the self-loop can be deleted without affecting the sorting power of the network, since a move using this self-loop would return the element to the same position as before. We term all remaining loops after this *essential*.
- If I is labeled as stack or sorted stack, then it can be labeled as a queue instead after reversing the input permutation and deleting a self-loop (if any) at I.

  To see the correctness of this, observe that we can delete the self-loop by the above observation. After this, I has no incoming edge (since the network is acyclic), and hence the

only purpose I serves is as an initial container to "feed" the elements into the network. A queue without self-loop will do the exact same thing, except that it feeds the elements into the network from the other end. Thus queue and stack are equivalent in this situation if we reverse the input permutation.

- Similarly, if O is labeled as a stack or sorted stack, then it can be labeled as queue instead after deleting a self-loop (if any) at O.

With this, there are only 16 sorting networks to consider: Each of I and O may be a deque or a queue, and each of I and O may have a self-loop or not. We denote these networks as (A, B), with $A, B \in \{Q, Q^+, D, D^+\}$, where Q and D stands for queue and deque, and Q^+ and D^+ stands for queue and deque with a self-loop. See Figure 2 for some examples.

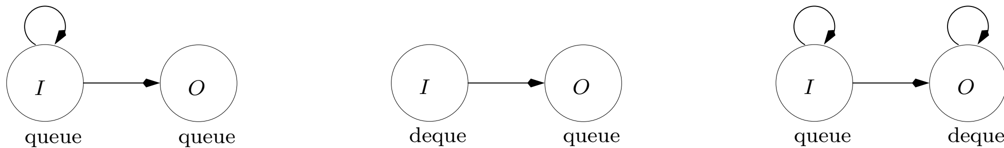

Fig. 2. Networks $(Q^+, Q), (D, Q)$ and (Q^+, D^+).

Our result below summarizes and extends results in Tarjan [40]:

Theorem 1 *No two node network without a self-loop can sort all permutations of n elements for $n \geq 6$.*

Proof: Particular cases are:

- (Q, Q): For $n \geq 2$ there are permutations of n elements that cannot be sorted. For example, if $1, 2$ is in the first queue so that the first element output will be 2, then this cannot be sorted as it will be input into the second queue in exactly the same order so that, again, 2 would be the first number output.
- (Q, D) and (D, Q): For $n \geq 4$ there are permutations of n elements that cannot be sorted. For example, if we have (Q, D) and $2, 3, 1, 4$ is in the queue, then 4 is output first. Each subsequent element is added at the beginning or end at the deque. This yields $2^3 = 8$ possible arrangements in the deque, none of which are in sorted order from either end of the deque. A similar argument with the same input, $2, 3, 1, 4$, works for the case (D, Q).
- (D, D): For $n \geq 6$ there are permutations of n elements that cannot be sorted. For example, if we have $2, 3, 6, 1, 4, 5$ in the first deque it can be seen by enumerating all the cases that it cannot be arranged in sorted order on the second deque. $\square$

Theorem 2 *Any two node network with exactly one essential self-loop, as well as the network $(Q+, Q+)$, needs $\Theta(n^2)$ to sort permutations of size n in the worst case.*

Proof: We show this explicitly by first observing that any such network can sort in time $O(n^2)$ and then giving matching lower bounds for each of the two-node networks with at least one self-loop.

For the upper bound, consider a cyclic-shift algorithm as follows. Assume the self-loop is on the input structure. The permutation is cyclically shifted within the input node using the self-loop until element 1 can be removed, then the input is shifted until 2 can be removed, etc. If the self-loop is in the output node, the sequence is shifted until the next element in the input can be inserted in its proper position in the sorted permutation. Each cyclic shift has a cost of at most $O(n)$ and hence the total cost over the n elements in the permutation is $O(n^2)$. A similar algorithm sorts in the same time if the self-loop is in the output data structure.

For the lower bound, we deal with the following cases separately:

(1) (D^+, Q), (Q^+, Q), (Q^+, D), (D^+, D) sorting networks
(2) (Q, D^+) and (Q, Q^+) sorting networks
(3) (D, D^+) and (D, Q^+) sorting networks
(4) the (Q^+, Q^+) sorting network

Case 1: (D^+, Q), (Q^+, Q), (Q^+, D), (D^+, D) For the lower bound, we show the proof for the (D^+, D) case. The (Q^+, Q), (Q^+, D) and (D^+, Q) cases are similar. For ease of description we assume n is divisible by 4. We show a permutation of n elements such that no matter how the algorithm inserts them in the deque D, it must use n^2 moves to remove them from the deque D^+. The permutation in the deque consists of four blocks. The first block contains the even elements from $n/2+2$ to n, the second block contains the even elements from $n/2$ to 2, the third block contains the odd elements from $n/2+1$ to n, the fourth block contains the odd elements from $n/2-1$ to 1. Observe that the deque at all times must hold a consecutive run of elements $i, i+1, \ldots, j-1, j$ in sequential order, since there is no self-loop in the deque nor any possibility of inserting a value ℓ if it were missing in the sequential list $\ldots \ell-2, \ell-1, \ell+1, \ell+2, \ldots$ of elements already in the deque.

Thus after element i and/or j is outputted the values in the deque D^+ must be rotated using the self-loop in either direction so as to reach either the value $i-1$ or $j+1$. That is, the data structure D^+ can be pictured as a wheel that can turn in either direction. But the permutation is constructed such that for the first $n/4$ values outputted, $i-1$ is located at least $n/4$ elements away from i in either direction immediately after it was outputted (the same holds for $j+1$ and j). Hence to output one of $i-1$ or $j+1$, at least $n/4$ moves are required. Thus the total number of moves is at least $\sum_{i=1}^{n/4} n/4 = n^2/16 = \Omega(n^2)$. □ (Case 1)

Case 2: (Q, D^+) and (Q, Q^+)

We show the proof for the (Q, D^+) case. The (Q, Q^+) case is similar.

For ease of description we assume n is divisible by 4. We show a permutation of n elements such that the algorithm must use n^2 moves to insert them in the deque D^+. The permutation in the queue consists of four blocks. The first block contains the elements from $n/4$ to 1 in that order at the front of the queue (so that 1 would be output first, followed by 2, etc.). The second block contains the elements $\frac{3n}{4}$ to $\frac{2n}{4}+1$ in that order. The third and fourth blocks contain the elements n to $\frac{3n}{4}+1$ and $\frac{2n}{4}$ to $\frac{n}{4}+1$ interleaved, i.e. n, $\frac{2n}{4}$, $n-1$, $\frac{2n}{4}-1$, $\ldots$, $\frac{3n}{4}+1, \frac{n}{4}+1$.

Then all these elements must be inserted into D^+ so that they are in order $n, n-1, \ldots, 2, 1$.

Once the elements from the first and second blocks have been inserted we must insert the interleaved elements. But in order to do that we must rotate the deque with self-loops. But between any two elements in the interleaved block there are at least $n/4$ other elements coming from either the first or second blocks (e.g. after we have inserted the first interleaved element $\frac{n}{4}+1$ we must rotate past either all of elements $n/4$ to 1 or past all of elements $\frac{3n}{4}$ to $\frac{2n}{4}$ in order to insert $\frac{3n}{4}+1$). This is repeated for each of $2 \times n/4$ elements, thus the total number of moves is at least $2 \cdot n^2/16$. □ (Case 2)

Case 3: (D, D^+) and (D, Q^+)

The proof is an argument similar to the previous case but with blocks of size $n/6$ arranged with 1 to $n/6$ in the first block, $\frac{3n}{6}+1$ to $\frac{4n}{6}$ in the second block, $\frac{n}{6}+1$ to $\frac{2n}{6}$ interleaved with $\frac{4n}{6}+1$ to $\frac{5n}{6}$ in the third and fourth blocks, $\frac{5n}{6}+1$ to n in the fifth block, and $\frac{2n}{6}+1$ to $\frac{3n}{6}$ in the sixth block. □ (Case 3)

Case 4: (Q^+, Q^+)

For the lower bound, we show this by exhibiting a specific permutation that requires $\Omega(n^2)$ moves. Consider the sequence in which the elements are arranged in reverse order i.e. $n, n-1, \ldots, 3, 2, 1$. Observe that any pair of consecutive elements in the input form an inversion, i.e. $a_i > a_{i+1}$ whereas in the sorted output we require $a_i < a_{i+1}$ for all i. The sorting algorithm removes an element, x, from the first queue, using the self-loop if necessary, and places it in the second queue. Then it removes a second element, y, and places it in the second queue. Now, either $y < x$ or $y > x$. If $y > x$ then this means that y appears before x in the input queue and we must use the self-loop to move elements so that we can access y. We call this element movement a "go-around."

If $y < x$ then when we insert it in the second queue it will be after x (i.e. closer to the rear) and we will have an inversion in the second queue. This inversion can be undone now or later by executing a go-around on the output queue. So in either case, we need a go-around either at the input queue or at the output queue.

If a queue is not very full a go-around is not costly. As well, if the last element placed in the output queue is close in the current sorted rank to the largest one contained in it, a go-around is not costly either. Observe, however, that in a not-too-empty queue a consecutive pair transfer must incur at least one go-around that involves many moves, either in the input queue or the output queue. More formally, let j be the number of elements in the second queue. Initially $j = 0$, and as we add elements to the second queue until j is $n/4$. At that point there will be $3n/4$ elements in the first queue. Both queues thus have at least $n/4$ elements. A pair of go-arounds in the input or output queue shifts at least $n/4$ elements and this remains the case for the next $n/2$ movements of elements. Thus $(n/2)(n/4)/2 = n^2/16 \in \Omega(n^2)$ moves are required. □ (Case 4)

This concludes the proof of Theorem 2. □

Theorem 3 *The sorting networks (Q^+, D^+), (D^+, Q^+), and (D^+, D^+) require $\Omega(n^{3/2})$ moves to sort a permutation, in the worst case.*

Proof: First consider (Q^+, D^+). The other two cases are similar.

We will use a Kolmogorov complexity argument. Consider a Kolmogorov random permutation (i.e. incompressible). The sequence of moves between Q and D uniquely sorts this permutation and hence encodes it. This sequence can be counted as follows. Suppose to remove element i from Q and insert it in D we use self-loops to move the k_i^Q positions in the queue, Q, and then move k_i^D positions in the deque, D, for some value of k_i^D and k_i^D. We also need one bit to record whether we move the deque to the right or to the left. Then the total number of moves is the following summation which, by a Kolmogorov argument, is bounded from below by $\lg n! = n \lg n + O(n)$:

$$\sum_{i=1}^{n} (\lg k_i^Q + \lg k_i^D) + \sum_{i=1}^{n} 1 = \lg n! \geq n \lg n + O(n) \tag{1}$$

But also

$$\sum_{i=1}^{n} (k_i^Q + k_i^D) = T,$$

where T is the total number of moves. Then by convexity, if we replace all the k's by $T/2n$ this maximizes the summation:

$$\sum_{i=1}^{n} \left(\lg \frac{T}{2n} + \lg \frac{T}{2n} \right) + \sum_{i=1}^{n} 1 \tag{2}$$

Putting together (1) and (2) we have:

$$\begin{aligned} 2n \lg \frac{T}{2n} + n &\geq n \lg n + O(n) \\ 2n \lg T - 2n \lg n - 2n + n &\geq n \lg n + O(n) \\ \lg T &\geq \frac{3}{2} \lg n + O(1). \end{aligned}$$

Take the exponential of both sides to get

$$T \geq \sqrt{2} n^{3/2}$$

The argument for (D^+, Q^+) is identical. The one for (D^+, D^+) is similar but uses two bits to record the left-right directions for the two deques. □

Because of its similarity with the previous proof we now show a lower bound for a sorting network with three nodes that will be useful later. Consider the (Q, D^+, Q) sorting network consisting of a read-only input queue, a deque with a self-loop, and a write-only output queue. Edges go between the input queue and the deque, between the deque itself (the self-loop) and between the deque and the output queue.

Theorem 4 *The number of operations performed in a* (Q, D^+, Q) *sorting network is* $\Omega(n^{3/2})$ *in the worst case.*

Proof: We consider a permutation whose last entry is 1 and as such all elements of the sequence have to be input into the deque before any element can be output. From the second insert or, before pushing an element from the input into the deque, the sorting algorithm might move elements from one end of the deque to the other. This is encoded as $+k$ if k elements were moved from the end closest to the input to the end closest to the output, and as $-k$ if k elements were moved in the opposite direction. Similarly, the outputting is encoded by the moves that take place between outputs from the end closest to the output to the output queue. We encode a zero if no move takes place. Clearly the sorting sequence uniquely identifies the input permutation. It uses $2n$ integers $i_1, \ldots, i_{2n}$ and $2n$ signs $s_1, \ldots, s_{2n}$ for this description.

From this we can derive a lower bound using Kolmogorov complexity. Consider a Kolmogorov random permutation. The sequence of moves uniquely sorts this permutation and hence it encodes it. Thus the sequence of moves cannot be all very short as otherwise the encoding itself would be very short. The length of the encoding is represented below. Note that the $2 \lg \lg i_j$ term comes from the self-delimiting encoding of the number of bits required to represent i_j. Also note that since encoding each sign s_j consumes one bit per number moved, a $+1$ term is also added. The total amount of space required is then

$$\sum_{j=1}^{2n} \left[2 \lg \lg i_j + \lg i_j + 1 \right]. \tag{3}$$

It follows from convexity that summation (3) is maximized when $i_j = T/2n$ for all j, where $\sum_{j=1}^{2n} i_j$ equals the total time T spent by the algorithm. Now, since the permutation is Kolmogorov random we have $2n \lg(T/2n) + 2n = 2n \lg T - 2n \lg n \geq n \lg n - o(n \lg n)$; that is, $2 \lg T \geq 3 \lg n - o(\lg n)$ or $T \geq n^{3/2}$ as claimed. □

4 k–Stack Sorting

We consider now a model of k stacks connected in series, with bidirectional flow between consecutive stacks. The input is connected to all k stacks and all k stacks are connected to the output. For three stacks we can use a bottom up version of mergesort to sort a permutation in $O(n\ log\ n)$ time. To see this, place the first two elements in each of two stacks and then place them, in sorted order, in the third stack. Now place the next two elements, one in the stack with the two previous elements and one in one of the other two stacks. Now place these two elements in sorted order in the empty stack. Now we have two sorted sequences of length two in two separate stacks that can be merged into one sorted sequence of length four in the empty stack. Repeat the construction with the next two pairs of elements in the input permutation to produce a second sorted sequence of length four and finally merge the two sequences of length four. Continue this process until the entire permutation is sorted and in one of the three stacks.

Finally, pop this stack. This procedure can easily be generalized to k stacks where as a first step we produce sorted sequences of length $k-1$ instead of length two.

Theorem 5 *Sorting with k interconnected stacks takes at most $O(n \log_{k-1} n)$ steps and at least $\Omega(n \log_{4k-2} n)$ steps.*

Proof: The algorithm consists of recursively merge sorting $k-1$ groups from each of $k-1$ stacks into the last k stack. This shows the upper bound. For the lower bound, we observe that there are $n!$ permutations. On the other hand, if we are sorting using k stacks using $T(n)$ operations, there are at most $(4k-2)^{T(n)}$ different sorting sequences of length $T(n)$ (each of the k stacks except the last two are connected to three other elements and the input is connected to all k stacks. At any point there are then $3(k-2)+2(2)+k = 4k-2$ choices for moves). From the standard information theoretical argument we know that the number of possible output permutations must be at least as large as the total number of permutations, namely $n!$. Thus we obtain $n! \leq k^n(4k-2)^{T(n)}$ which implies $n \log n + O(n) \leq T(n) \log(4k-2) + n \log k$ and hence

$$T(n) \geq n \frac{\log n}{\log(4k-2)} + O(n) = \Omega(n \log_{4k-2} n)$$

as required. □

Thus far we have exhibited sorting networks of three main time complexities: $O(n^2)$, $O(n^{3/2})$, and $O(n \log n)$. This last we generalized to k-stacks obtaining an algorithm with complexity $O(n \log_{O(k)} n)$. We have seen that sorting networks of three stacks fully interconnected suffice to mergesort n numbers in $O(n \log n)$ time and its generalization to k-stacks. We now further explore the boundary between sorting networks of different complexity.

An interesting alternative is an active stack model. This model allows two kinds of stacks: active and inactive. A fixed number, say t, of stacks must be active at any time. A stack must be active if it contains elements, but an empty stack can be active (usually because we anticipate putting elements into it).

The t active stacks must be the rightmost of all stacks that have ever held an element. Additionally, the leftmost of the t active stacks can only send elements to the one immediately to its right and cannot receive elements from the others.

We can look at this model as one in which the t active stacks move through the sequence of k stacks and in which we deactivate a (necessarily empty) stack on the left as we activate a new stack on the right, or we can look at this model as one in which there are t physical stacks that are reused k times, using a technique similar to register renaming.

We can use the model with three active stacks to sort n elements à la mergesort, similar to the procedure described at the beginning of Section 4 but with a twist, reusing the stacks. First, split the list into two halves of size $n/2$. Use two stacks to sort one half by first placing one element in each stack. Then the third element must fit somewhere in the order relative to the position of the other two: either before both, between both, or after both. If it is before or after both, move the element it should be adjacent to to the other stack and place the third

element in the now empty stack. If it is between both, place the third element on top of either stack. Now repeat this procedure for each successive elements, shifting elements back and forth between the two stacks to make room in the proper place for the new element. Once the half is sorted it can be left in one of the stacks, temporarily dormant at the bottom, while the same two stacks are used to sort the second half in the same way. Then, with each half in a different stack, they can be merged to a third stack. The argument can also be generalized to sort groups of size $n^{1/k}$, as will be discussed in the proof of Theorem 6.

Theorem 6 *The number of operations performed by two stacks connected in series with three stacks active is* $\Theta(n^{1+1/k})$ *in the worst case.*

Proof: First consider the lower bound. In this case we repeat the argument in Theorem 4 for k stacks instead of 2 and combining inequalities (3) and (7) we obtain $kn \lg(T \,/\, kn) + kn \lg k = kn \lg T - kn \lg n \geq n \lg n - o(n \lg n)$ which implies $\lg T \geq (k+1)/k \lg n - o(1/k \lg n)$ and hence $T \geq n^{1+1/k} - o(n^{1/k}) = \Omega(n^{1+1/k})$ as required.

Now consider the upper bound. Split into elementary groups of size $n^{1/k}$. Sort each of them using an action of shifting elements back and forth between two stacks taking quadratic time on the number of elements, i.e. $n^{2/k}$ per group for a total time of $n/n^{1/k} \times n^{2/k} = n^{1+1/k}$.

Now at step i for $i = 1..k$ we merge the previous sorted groups of size $n^{i/k}$, grouping $n^{1/k}$ of them at a time resulting in sorted groups of size n^{i+1}/k. Each merging step takes time n^{i+1}/k and there are $n/n^{i/k}$ such groups, so the total time for step i is $n^{1+1/k}$, and since we do k of them the total time is:

$$kn^{1+1/k} = O(n^{1+1/k}). \qquad \square$$

5 Online Sorting

In this section we study an online-variant of (Q, D^+, Q) sorting where we have to insert each given element into the deque before we are given the next element. There are two models of online sorting. In one model we know that the input elements are a permutation of integers; in particular, once we are given the elements i and $i + 1$, we know that no element between them will follow. We call this the *permutation model.* In the other model, we know nothing about the input elements; they could appear repeatedly, or for any two distinct elements another element between them could follow. We call this the *sorting model.*

Consider now a cursor model. In this model a cursor points at the position in the data structure where we can input or output elements we are focused on. Essentially it locates and points to the proper location. Under this model, we define two types of operation: an **insert** operation and an **extract** operation. The two operations are complementary. In both cases we move an element from one data structure to another. One data structure must have a self-loop and any structure with a self-loop can be interpreted as a string with a cursor running through it: just loop the elements so that the position the cursor is pointing to is the input or output position

desired. In the case of insert we picture a cursor running through the second data structure, selecting where to insert the element that is output from the first data structure according to the data structure rules. In extract, we picture a cursor running through the first data structure, selecting which element to output to the second data structure–then for the second data structure the element must be placed in it according to the data structure rules (i.e. no cursor). A **self-loop move** is a transfer of an element from one side of the data structure to the other.

Observe that the cyclic shift algorithm in the proof of Theorem 2 in Section 3 is actually an online-algorithm in the sorting model: we simply put all input in the order received onto the deque, and then sort it during the extract phase with a cost of $O(n^2)$ moves. Since we do allow repeated elements here, we will have to first run through the input checking for repetitions.

In this section, we show that this bound, surprisingly, is tight. Moreover, the lower bound is in the permutation model, which is in the stronger model for lower bounds.

Theorem 7 *Any online (Q, D^+, Q) algorithm takes $\Omega(n^2)$ moves in the worst case, even in the permutation model.*

Proof: For ease of description, we assume that n is divisible by 8. We will show how to construct a permutation of n numbers such that no matter how the online algorithm inserts them, we can force it to use $n^2/64$ moves, either during the insertion phase or the extraction phase.

We start by sending the $n/2$ odd numbers in arbitrary order. Now assume that the algorithm has inserted these odd numbers into the deque. In what follows, we will consider the deque again as a string, and the current position for inserting into the deque as the cursor-position in the string.

We split the string of $n/2$ odd numbers into four parts of $n/8$ numbers each. The *left block* consists of the leftmost $n/8$ numbers, and the *right block* consists of the fourth set of $n/8$ numbers. The $n/4$ numbers between these blocks act as a buffer that force us to move at least $n/8$ steps each for the next $n/8$ numbers.

For the next $n/8$ (even) numbers to be sent, we consider the current position of the cursor. If the cursor is in the left half of the string, then pick an odd number i in the right block for which $i+1$ has not been sent yet. (There exists such a number since there are $n/8$ odd numbers in the right block.) Similarly, if the cursor is in the right half of the string, pick an odd number i in the left block for which $i+1$ has not been sent yet. Send the number $i+1$ next.

By choice of i, the distance between the cursor and i is at least $n/8$. In response to sending number $i+1$, the cursor will move some number of positions, k, and then insert the number $i+1$. Thus, the distance between i and $i+1$ is at least $n/8-k$. In total the cursor moves k steps when inserting $i+1$, and will have to move at least $n/8-k$ steps when extracting $i+1$ after having extracted i. Thus, in both insertion and extraction phase together, the cursor must move $n/8$ moves to get $i+1$. Since this applies to $n/8$ numbers, the total number of moves is at least $n^2/64$. □

Lastly we consider the case of $(Q, D+, D+)$ and show that it can sort in $\Theta(n^{3/2})$.

Note we use the term online algorithm in the sense of an algorithm which has access to the input one item at time and does not know the length of the input sequence until it reaches its end.

Theorem 8 *Any online $(Q, D+, D+)$ algorithm takes $\Theta(n^{3/2})$ moves in the worst case.*

Proof: For the lower bound an argument similar to the proof of Theorem 4 will work. In this case we will need an additional bit to record the sign of the moves in the second $D+$.

Now for the upper bound, assume for ease of computation that n is a perfect square. Split the input into $\sqrt{n}$ blocks of size $\sqrt{n}$. The elements are output in fixed order from the queue Q. For each block, as the ith element of each block is output, the cursor on the middle deque $D+$ is moved so that this element is inserted in the ith block of $D+$. Moreover, the elements are inserted so that they are in order, e.g. if the permutation is $16, 10, 7, 3, |12, 9, 4, 1, |15, 11, 6, 2, |14, 13, 8, 5$ then the third block of $D+$ is $13, 11, 10, 9$. Moving the deque cursor in this fashion costs $O(\sqrt{n})$ for each block of output for a total of $O(n^{3/2})$.

Now merge these blocks in the output deque $D+$ in a manner similar to mergesort. The first block is inserted at no cost and without changing the order of its elements. Then the cursor is moved to the beginning of the output string at a cost of at most $O(n)$. The second (sorted) block is merged with the current (sorted) string T with a cost of at most $O(n)$ right cursor movements, and then the cursor is moved to the beginning again ($O(n)$ cost again). A similar situation occurs with third sorted block and so on. This takes $O(n)$ steps per block and $O(n^{3/2})$ total. $\square$

6 Conclusions and Open Problems

We have established a classification of the power and efficiency of sorting networks. We obtained a tradeoff between number of nodes and efficiency. An number of open problems remain. Most obviously there is the problem of obtaining matching upper and lower bounds for the sorting networks in Theorems 3 and 4 that can sort in time $O(n^2)$ but whose lower bound is only $\Omega(n^{3/2})$.

Acknowledgements: We wish to thank the participants of the Algorithmic Problem Session at the University of Waterloo and J.D. Horton for many helpful discussions.

References

[1] M. H. Albert, M. D. Atkinson, "Sorting with a forklift," *Electronic J. Combinatorics*, 9 (2), (2003), Paper R9.

[2] N. M. Amato, M. Blum, S. Irani, R. Rubinfeld, "Reversing Trains: A Turn of the Century Sorting Problem", *J. Algorithms*, 10 (1989), 413–428.

[3] M.D. Atkinson, "Sorting permutations with networks of stacks," Technical Report TR-210, School of Computer Science, Carleton University, Ottawa, Canada, August 1992.

[4] M. D. Atkinson, "Generalised stack permutations," *Combinatorics, Probability and Computing*, 7 (1998), 239–246.

[5] M. D. Atkinson, M. J. Livesey, D. Tulley, "Permutations generated by token passing in graphs," *Theoretical Computer Science*, 178 (1997), 103–118.

[6] M.D. Atkinson, N. Ruskuc, M.M. Murphy, "Sorting with Two Ordered Stacks in Series" *Theoretical Computer Science*, 289 (2002), 205–223.

[7] M. D. Atkinson, J.-R. Sack, "Sorting with parallel pop-stacks," *Information Processing Letters*, 70 (1999), 63–67.

[8] M. D. Atkinson, T. Stitt, "Restricted permutations and the wreath product," *Discrete Math.*, 259 (2002), 19–36.

[9] M.D. Atkinson, D. Tulley, "Bounded capacity priority queues," *Theoretical Computer Science*, 182 (1997), 145–157.

[10] D. Avis, M. Newborn, "On pop–stacks in series," *Utilitas Mathematica*, 19 (1981), 129–140.

[11] M. Bóna, "Exact enumeration of 1342–avoiding permutations: a close link with labeled trees and planar graphs," *J. Combin. Theory Ser. A*, 80 (1997), 257–272.

[12] M. Bóna, "The solution of a conjecture of Stanley and Wilf for all layered patterns," *J. Combin. Theory Ser. A*, 85 (1999), 96–104.

[13] M. Bóna, "Symmetry and Unimodality in Stack sortable permutations," *J. Combin. Theory Ser. A*, 98, (2002), 201–209.

[14] M. Bóna, "A simplicial complex of 2–stack sortable permutations." *Advances in Applied Mathematics*, 29 (2002), 499–508.

[15] M. Bóna, Corrigendum to "Symmetry and Unimodality in Stack sortable permutations," *J. Combin. Theory Ser. A*, 99 (2002), 191–194.

[16] M. Bońa, A survey of stack-sorting disciplines, *Electronic J. Combin.* 9 (2002/03), no. 2, Article 1, 16pp.

[17] P. Bose, J.F. Buss, A. Lubiw, "Pattern matching for permutations," *Information Processing Letters*, 65 (1998), 227–283.

[18] M. Bousquet-Mélou, "Multi-statistic enumeration of two-stack sortable permutations," *Electronic J. Combinatorics*, 5 (1998), Paper R21.

[19] M. Bousquet-Mélou, "Sorted and/or sortable permutations," *Discrete Math.* 225 (2000), 25–50.

[20] M. Bousquet-Mélou, "Counting Walks in the Quarter plane," in *Mathematics and Computer Science: Algorithms, trees, combinatorics and probabilities*, Trends in Mathematics, Birkhauser, 2002, pp. 49–67.

[21] P. Branden, "The generating function of two-stack sortable permutations by descents is real–rooted," arXiv:math.CO/0303149.

[22] F. Brenti, "Unimodal, log-concave and Pólya frequency sequences in combinatorics" *Mem. Amer. Math. Soc.*, 81 (1989), no. 413.

[23] F. Brenti, "Hilbert polynomials in combinatorics," *J. Algebraic Combin.*, 7 (1998), 127–156.

[24] S. Dulucq. S. Gire, J. West, "Permutations with forbidden subsequences and non-separable planar maps." *Proceedings of the 5th Conference on Formal Power Series and Algebraic Combinatorics (FPSAC)* (Florence, 1993), *Discrete Math.*, 153 (1996), 85–103.

[25] S. Even, A. Itai, "Queues, and graphs," in Z. Kohavi and A. Paz, eds., *Theory of Machines and Computations, Proc. Internat. Symp. on the Theory of Machines and Computations*, Academic Press, New York, 1971, 71–86.

[26] I. P. Goulden, J. West, "Raney paths and a combinatorial relationship between rooted nonseparable planar maps and two–stack–sortable permutations," *J. Combin. Theory Ser. A*, 75 (1996), 220–242.

[27] O. Guibert, "A combinatorial proof of J. Wests conjecture," *Discrete Math.*, 187 (1998), 71–96.

[28] O. Guibert, "Stack words, standard Young tableaux, permutations with forbidden subsequences and planar maps," *Formal power series and algebraic combinatorics (FPSAC)* (Minneapolis, MN, 1996). *Discrete Math.* 210 (2000), no. 1–3, 71–85.

[29] T. Harju, L. Ilie, "Forbidden subsequences and permutations sortable on two parallel stacks," in: C. Martin-Vide, V. Mitrana, eds., *Where Mathematics, Computer Science, Linguistics, and Biology Meet*, Kluwer, Dordrecht, 2001, 267–275.

[30] T. Jiang, M. Li, P. Vitanyi, "Average-Case Complexity of Shellsort (preliminary version)," arXiv:sc.CC/9906008.

[31] A.E. Kézdy, H.S. Snevily, C. Wang, "Partitioning permutations into increasing and decreasing subsequences," *Journal of Combinatorial Theory A*, 74 (1996), 353–359.

[32] D. E. Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms* and *Volume 3, Sorting and Searching*, Addison-Wesley, 1973. 2nd ed. 1997 and 1998.

[33] G. Kreweras, "Sur une classe des problèmes liés au treillis des partitions dentiers," *Cahiers du B.U.R.O*, 6 (1965), 5–105.

[34] M. M. Murphy, *Restricted permutations, antichains, atomic classes and stack sorting*, PhD thesis, University of St Andrews, 2002.

[35] V.R. Pratt, “Computing permutations with double-ended queues, parallel stacks and parallel queues,” *Proc. of 5th. ACM Symp. Theory of Computing*, 5 (1973), 268–277.

[36] R. Simion, F.W. Schmidt, “Restricted permutations,” *Europ. J. Combinatorics* 6 (1985), 383-406.

[37] Z. E. Stankova, “Forbidden subsequences,” *Discrete Math.* 132 (1994), 291-316.

[38] Z. Stankova, J. West, “A new class of Wilf-equivalent permutations,” *J. Algebraic Combin.*, 15 (2002), 271–290.

[39] R. P. Stanley, “Log-concave and unimodal sequences in algebra, combinatorics, and geometry,” in *Graph Theory and Its Applications: East and West,* Ann. NY Acad. Sci. , 576 (1989), 500–535.

[40] R.E. Tarjan, “Sorting Using Networks of Queues and Stacks” *Journal of the ACM (JACM)*, 19 (2) (April 1972), 341–346.

[41] W. Unger, “On the k–colouring of circle graphs,” *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* Number 294 (1988), 61–72.

[42] W. Unger, “The Complexity of Colouring Circle Graphs,” *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science*, Number 577 (1992), 389–400.

[43] J. West, *Permutations with forbidden subsequences and Stack sortable permutations*, PhD-thesis, Massachusetts Institute of Technology, 1990.

[44] J. West, “Sorting Twice Through a Stack,” *Theoretical Computer Science*, 117 (1993), 303–313.

[45] J. West, “Generating trees and the Catalan and Schröder numbers,” *Discrete Math.* 146 (1995), 247-262.

[46] D. Zeilberger, “A proof of Julian Wests conjecture that the number of two–stack–sortable permutations of length n is $2(3n)!/((n+1)!(2n+1)!)$,” *Discrete Math.*, 102 (1992), 85–93.