HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-2001-03

Fun-Sort

Therese Biedl* Timothy Chan* Erik D. Demaine*
Rudolf Fleischer† Mordecai Golin† J. Ian Munro*

February 20, 2001

Abstract

In this paper we study greedy in-place sorting algorithms which miraculously happen to work in reasonable time. Dumb-Sort which repeatedly compares all possible pairs of array cells sorts n elements in $n-1$ cycles, or time $O(n^3)$. Not-So-Dumb-Sort, which only tests adjacent cells, also sorts in $n-1$ cycles, or in time $O(n^2)$. Guess-Sort, a randomized version of Dumb-Sort, runs in expected time $O(n^2 \log n)$. And Fun-Sort, an in-place variant of Insertion-Sort that performs repeated insertions by binary search into an initially unsorted array, sorts in time $O(n^2 \log n)$.

1 Introduction

Comparison based sorting algorithms tend to be based on a few basic paradigms: insertion, exchanging, selection, and merging [8, page 73]. Most methods operate by maintaining an invariant (of sortedness) on an increasingly longer segment of the array. The exchange sorts (e.g., Bubblesort and Quicksort) are notable exceptions to this approach. But even they maintain strong invariants on the progress made thus far. Shellsort, from one point of view, maintains sorted subarrays of increasingly greater length as the process continues. Perhaps greater "density" is a better term for the notion of the interleaved sorted segments of Shellsort. From another point of view, Shellsort

*Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada. Email: {biedl, tmchan, eddemaine, imunro}@uwaterloo.ca.

†Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Email: {rudolf, golin}@cs.ust.hk.
R. Fleischer was partially supported by HKUST grant DAG00/01.EG03.

is delightfully chaotic. At any stage, one is presented with a subregion which one hopes is reasonably close to sorted order, and then is to complete the task by using linear Insertion-Sort. Magically, it works pretty well.

Our starting point is the notion of sorting by repeatedly moving values from current to "more likely" locations by performing binary searches. If an array were in sorted order, except for the location of one value, then this approach would minimize both the number of comparisons and the number of moves necessary to complete the task. The approach would certainly place the set maximum and minimum in their proper spots, and never move them again. If the array is close to being in sorted order, the process will generally make further progress. Our question is whether the general approach of binary insertion into an initially unordered array can be harnassed to yield a viable sorting algorithm.

We are able to adapt the method to correctly perform a sort and bound the worstcase runtime reasonably accurately. However, we feel the greater contribution of this paper is to lay open the approach, both for purposes of algorithmic improvements and for improvements in the expected runtime of our approach. In particular, we observe that the expected position of an element, binary inserted into an array, is approximately its rank.

Several variations on this theme are explored. In a more greedy approach we move values to "more likely" locations by comparing all pairs of array cells and swapping their contents if they are not in the right order. We show that $n-1$ iterations (or *cycles*) of this brute-force approach suffice to sort any input, independent of the order of the comparisons in a cycle. Interestingly, it turns out that not all $\Theta(n^2)$ pairs have to be considered in each cycle. It is sufficient to have $n-1$ cycles that only compare the $n-1$ pairs of adjacent cells in any arbitrary order (Bubblesort is a special case of such an algorithm). Previously, it was only known that n cycles will sort correctly [3, 12, 13].

Our results are also of interest because all of our sorting algorithms are purely in-place. And with the exception of the unordered binary searches algorithm they are also oblivious, i.e., they could be realized by a sorting network. But in contrast to other work on sorting networks, we are not interested in exploiting the parallelism of our sequential algorithms to reduce the number of rounds or the depth of the network, for example [2, 6, 7, 12, 13].

This paper is organized as follows. In Section 2 we introduce our model of inversion-fixing algorithms and show that all inversion-fixing algorithms will eventually terminate. In Section 3 we give a simple proof, based on the 0-1 Principle, that Dumb-Sort which performs $n-1$ cycles of comparisons of all possible cell pairs sorts correctly. We think that this is a more natural $\Theta(n^3)$-time "slow-sort" algorithm than Julstrom's Slow Sort [5] which is somehow artificially bloated. We then show that Not-So-Dumb-Sort, a streamlined version of Dumb-Sort which only considers adjacent cells, also sorts correctly in $n-1$ cycles, reducing the runtime to $O(n^2)$.

In Section 4 we introduce Guess-Sort, a randomized variant of Dumb-Sort, which sorts in expected time $O(n^2 \log n)$. In Section 5 we show that Fun-Sort which is Insertion-Sort with binary-search insertions into an initially unordered array sorts in time $O(n^2 \log n)$. And we conclude with a few open problems in Section 6.

2 To Swap or not to Swap — That's our Question

Assume we are given n elements from a totally ordered set, stored in an array of size n which we think of being arranged from left to right in increasing order. For simplicity we assume that all elements are different, but the results in this paper also hold in the general case. At any time, we denote the content of cell i by a_i, for $i = 1, \ldots, n$. Our goal is to sort these elements in non-decreasing order. Intuitively, the simplest way to sort an array is to concentrate on just two elements at a time, from which we can extract only one bit of information. A pair (i, j) of cells i and j, where $i \neq j$, is *good* if the cell contents are correctly ordered, i.e., $a_i < a_j$ if $i < j$, or $a_i > a_j$ if $i > j$. Otherwise the pair is *bad*. Bad pairs are also called *inversions*. The number of inversions in the array can be between zero if the array is correctly sorted, and $\binom{n}{2}$ if the array is sorted in reverse order.

Following these ideas, the most basic method of sorting is to search for and swap bad pairs. De Bruijn's [3] called the action of comparing the contents of a pair of cells and leaving the smaller value in the left cell (i.e., the cell with lower index) a *swap*, or a *mini-swap* if the two cells are adjacent. Note that swapping a good pair has no effect, whereas in a swap of a bad pair (an *exchange swap*) the cell contents are exchanged.

The following simple observation seems to be folklore but we could not find it stated properly in the literature (although it is used in [3]). It implies that we can sort by repeatedly doing exchange swaps.

Lemma 1. *An exchange swap strictly decreases the number of inversions.*

Proof. Consider an exchange swap (i, j), where $i < j$ and $a_i > a_j$ (see Fig. 1). Let (k, ℓ) be a good pair before the swap, where $k < \ell$. Since the content of cell i is decreasing and the content of cell j is increasing, (k, ℓ) can only become a bad pair if $k = j$ or $\ell = i$.

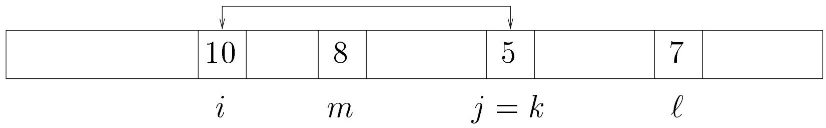

Figure 1. If we swap the bad pair (i, j) then the good pair (k, ℓ) becomes a bad pair. However, the bad pair (i, ℓ) becomes a good pair. We also accidentally turn the bad pairs (i, m) and (m, j) into good pairs but that is ok.

In the former case, (i, ℓ) was a bad pair before the swap (because a_i moves to cell j) but it is good afterwards; and in the latter case, (k, j) was a bad pair before the swap (because a_j moves to cell i) but it is good afterwards. In any case, the number of bad pairs decreases because (i, j) becomes a good pair. $\square$

We only consider sorting algorithms based on swaps, called *exchange sorting algorithms* by Knuth [8, page 73]. We prefer to call them *inversion-fixing algorithms*. Knuth also calls sorting algorithms purely based on mini-swaps *primitive* [8, Exercise 5.3.4.36 on page 240]. Lemma 1 implies that inversion-fixing algorithms cannot make an unbounded number of exchange swaps.

Corollary 2. *Any inversion-fixing algorithm makes at most* $\binom{n}{2} = O(n^2)$ *exchange swaps.*

Proof. Initially, the array can have at most $\binom{n}{2}$ inversions, and each exchange swap strictly decreases the number of inversions by Lemma 1. $\square$

So if we knew the indices of the bad pairs we could easily sort in $O(n^2)$ time. Unfortunately, we cannot expect to get this information for free. In the next three sections we will show several simple but nevertheless correct and even semi-efficient ways to find bad pairs. We only consider *oblivious* algorithms, where the sequence of comparisons is fixed in advance. In particular, it is independent of the input sequence or the outcome of individual comparisons. That makes the algorithms similar to sorting networks, except that we do not try to exploit any parallelism.

3 Dumb-Sort — Test All, Hit One

A not too clever brute force way of finding an exchange swap is to try all swaps. This approach is similar to the Bellman-Ford algorithm for computing shortest paths in a weighted graph where in each step we know that there is one node of optimal value which we should relax next, but since we cannot identify that node we simply relax all nodes [10, Section 7.5.7] (therefore the bad runtime of that algorithm).

A *cycle* is a sequence of all $\binom{n}{2}$ swaps in some arbitrary order, the *swap sequence*. A *mini-cycle* is a sequence of all $n-1$ mini-swaps in some arbitrary order, the *mini-swap sequence*. Note that no mini-swap sequence can sort the input sequence $3, 2, 1$. On the other hand, there is a swap sequence that sorts all sequences, namely the swap sequence

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4),\ldots,(2,n),(3,4),\ldots .$$

However, not all swap sequences can sort any input sequence. For example, the sequence $3, 2, 1$ will not be sorted by the swap sequence

$$(1,2),(2,3),(1,3)$$

(the result would be $2, 1, 3$).

Since any swap sequence contains at least one exchange swap if the input is not sorted, $\binom{n}{2}$ cycles of arbitrary swap sequences suffice to sort any input by Corollary 2, resulting in a runtime of $O(n^4)$. The same is true for mini-cycles, resulting in a runtime of $O(n^3)$. But this is overly pessimistic. We will show that any input sequence must already be sorted after $n-1$ (mini-)cycles.

We use the generic name Dumb-Sort for any algorithm which consists of $n-1$ cycles of (possibly different) swap sequences. Similarly, Not-So-Dumb-Sort denotes any algorithm doing $n-1$ arbitrary mini-cycles. Since we can view the sequence of comparisons made by Dumb-Sort or Not-So-Dumb-Sort as an oblivious sorting network, we can use the well-known 0-1 Principle (see [8, Theorem Z on page 223], for example) to prove the correctness of the algorithm.

Theorem 3 (0-1 Principle). *If an oblivious sorting network sorts all sequences of '0's and '1's then it also sorts all sequences of any numbers.* □

Theorem 4. Dumb-Sort *sorts n elements in $n-1$ cycles, or time $O(n^3)$.*

Proof. Assume the input is a sequence of '0's and '1's. Note that a '0' can only move to the left and a '1' can only move to the right. In particular, if the leftmost '1' (rightmost '0') in the array is replaced by a '0' ('1') the content of this cell can never change again, i.e., the cell contains the final correctly sorted number.

If the input consists of n '0's or n '1's the theorem is clearly true. Otherwise, assume that at the beginning of a cycle cell i contains the leftmost

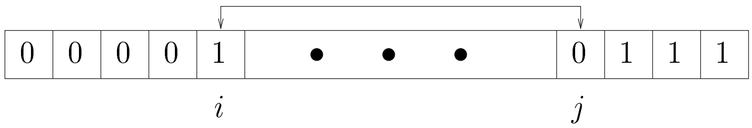

Figure 2. i is the leftmost '1'-cell and j is the rightmost '0'-cell.

'1' and cell j the rightmost '0' (see Fig. 2). If the array is unsorted, we have $i < j$. During the cycle none of the '0's to the left of cell i or '1's to the right of cell j are swapped. Also, once a '0' is placed in cell i or a '1' in cell j it is never moved out again. Consider the point in the cycle at which we swap (i, j). If $a_i = 1$ and $a_j = 0$ then they are exchange swapped. Thus, by the end of the cycle, either $a_i = 0$, $a_j = 1$, or both. □

The following example shows that the bound of Theorem 4 is tight. Assume the largest element is initially stored in cell 1 and the swap sequence ends $\ldots, (1, 2), (1, 3), \ldots, (1, n)$. Then the largest element only proceeds to cell 2 after the first cycle. Iterating this construction, we obtain a swap-sequence that can move the largest element in any cell only one cell to the right, so we need $n - 1$ of these cycles to move it from cell 1 to cell n.

The proof of Theorem 4 actually suggests that mini-cycles instead of full cycles might sometimes be sufficient. We can, for example, similarly prove that the mini-swap sequence $(1, 2), (2, 3) \ldots, (n - 1, n)$ (or the reverse sequence) sorts in $n - 1$ mini-cycles (the rightmost '1' left of the rightmost '0' will replace that '0' in any mini-cycle). And the runtime of this algorithm, which an attentive reader might have recognized as Bubblesort [8, p. 106], is only $O(n^2)$. Since *any* oblivious sorting network must contain all mini-swaps [8, Exercise 5.3.4.35 on page 240], Bubblesort is minimal among all sorting algorithms based on iterating cycles. Our proof also shows that we could arbitrarily augment the Bubblesort swap sequence by other swaps and still sort in $n - 1$ cycles (in general, adding comparisons to an oblivious sorting network may prevent the network from sorting correctly [8, Exercise 5.3.4.21 on page 238]). However, this result is not new. De Bruijn actually showed the following stronger theorem [3, Theorem 6.2].

Theorem 5. *Any sequence of mini-swaps that sorts correctly can safely be augmented by arbitrary swaps.* □

Some authors [13, Fact 1][12, Prop. 1.9] also attribute the following theorem to de Bruijn [3], even though his paper does not mention it. However, it is an easy corollary of Theorem 5.

Theorem 6. *Any sequence of n mini-cycles sorts n elements in time $O(n^2)$.*

Proof. It is known that n phases of the *odd-even transposition sorting network* sort n numbers [8, Exercise 5.3.4.36 on page 240]. Since the swap sequence of the i-th mini-cycle contains the i-th phase of Transposition Sort (either $(1,2),(3,4),\ldots$, or $(2,3),(4,5),\ldots$) as a subsequence, Theorem 5 implies that the n mini-cycles sort correctly. $\square$

Since no primitive sorting network (like Transposition Sort) can have less than $\binom{n}{2}$ swaps (otherwise, $n, n-1, \ldots, 1$ could not be sorted; see [8, Exercise 5.3.4.36 on page 240]), $n-1$ phases of Transposition Sort do not suffice to sort correctly, so this proof cannot be adapted to prove that Not-So-Dumb-Sort, which performs only $n-1$ mini-cycles, sorts correctly. We also note that de Bruijn proved Theorem 5 by the not universally accepted method of "proof by convincing example", so purists might object that the proof of Theorem 6 actually still contains a "gap". To put things straight we give a more direct proof which, how nice, even shows a stronger statement.

Theorem 7. Not-So-Dumb-Sort *sorts n elements in $n-1$ mini-cycles, or time $O(n^2)$.*

Proof. We make again use of the 0-1 Principle. Assume the input sequence contains k '1's and $n-k$ '0's, for some $0 < k < n$. We denote the '1's from left to right by $T_1, T_2, \ldots, T_k$. We prove by induction on the mini-cycles that after the i-th mini-cycle T_j has reached at least cell $\min\{i+2j-k, n+j-k\}$, for $j = 1, \ldots, k$. Thus, T_j reaches its final destination $n+j-k$ after at most $n-j \leq n-1$ mini-cycles.

Since $2j-k \leq j$ for $j \leq k$, the claim is true initially, i.e., for $i = 0$. Now consider the i-th mini-cycle, for some $i \geq 1$. We use induction on j to prove the claim. Since T_k is the rightmost '1' it will move at least one cell to the right during the mini-cycle if it has not yet reached its final cell n. This proves the inductive step for T_k.

Now consider T_j for some $j < k$. By the inductive hypothesis, T_{j+1} was at least in cell $\min\{(i-1)+2(j+1)-k, n+(j+1)-k\} = \min\{i+2j+1-k, n+j+1-k\}$ before the i-th mini-cycle. If T_j was at that time in cell $\min\{i+2j-k, n+j-k\}$ or further to the right then the claim is proved. Otherwise, it must have been in cell $\min\{(i-1)+2j-k, n+j-k\}$ by the inductive hypothesis. If $(i-1)+2j-k < n+j-k$ then it started the i-th mini-cycle left of a '0' and therefore moved at least one cell to the right during the mini-cycle, which proves the claim. $\square$

We note that the worst case in the proof above occurs if the input is $1, \dots, 1, 0, \dots, 0$, and then the bound can be tight. These inputs are the maximal elements of de Bruijn's partial order on input permutations (see [3] for details), and we implicitly proved an analogon to his Theorem 6.1 which states that it is sufficient to prove correctness for maximal elements of the partial order (in his case there is exactly one maximum element).

In view of Theorem 5, Theorem 4 is actually a corollary of Theorem 7 because each full cycle contains a mini-swap subsequence. But we think our simple direct proof has its own merits.

4 Guess-Sort — Even a Blind Pig Finds an Acorn

As argued before, we cannot build cycles from swap sequences with less than $n-1$ swaps, so there is no hope of getting a "dumb" (or greedy) $O(n \log n)$-size sorting network like the AKS sorting network [1]. But there is a simple randomized variant of Dumb-Sort which does better on the average. Guess-Sort chooses random pairs of cells to swap until the array is sorted.

Theorem 8. Guess-Sort *sorts n elements in expected time $O(n^2 \log n)$.*

Proof. If there are m bad pairs in the array we hit one with probability $\Theta(\frac{m}{n^2})$. Since the number of bad pairs decreases by Lemma 1 after each exchange swap, the expected number T_n of swaps until the array is sorted is bounded by

$$T_n = O\left(\sum_{m=1}^{n^2} \frac{n^2}{m}\right) = O(n^2 \log n) .$$

□

There is just a minor problem with this algorithm. How do we know when we can stop, i.e., when the array is sorted? One solution is to check after every $n-1$ tests whether the elements are already sorted correctly. This can be done with $n-1$ comparisons, so we at most double the runtime.

The bound of the theorem is tight. Guess-Sort needs expected time $\Theta(n^2 \log n)$ to sort the sequence $2, 1, 4, 3, \dots, n, n-1$. This is an application of the coupon collector's problem [11, page 57]: We need $\Theta(n)$ steps to guess a pair of adjacent cells, and this must happen $\Theta(n \log n)$ times until we have found all the pairs $(1,2), (3,4), \dots$. Note that it does not help us to choose only random adjacent pairs; we find bad pairs faster, but we make less progress by swapping them than by swapping some far-apart bad pair.

5 Fun-Sort — Smart Search in Bad Arrays

Another way of finding bad pairs is by binary search. While we are normally taught to apply binary search only when an array is sorted, we could rebelingly apply the same procedure to an unsorted array. To avoid special treatment of the array boundaries we assume $a_0 = -\infty$ and $a_{n+1} = \infty$. We keep two indices ℓ and h such that at any time $a_\ell < x < a_h$. Initially, $\ell = 0$ and $h = n + 1$. In each step we compare x to a_m, where $m = \lfloor \frac{\ell+h}{2} \rfloor$, and set $h = m$ if $x \leq a_m$, or $\ell = m$ if $x > a_m$. We stop if $h = \ell + 1$ and return the index h as the result of the search. We say the search was *successful* if $x = a_h$ (note that $x = a_l$ is impossible), otherwise it *failed*. Note that a search for x can fail even though x is contained in the array.

Fun-Sort is an in-place variant of Insertion-Sort. It performs repeated insertions by binary search into an initially unsorted array, and surprisingly sorts correctly after only $O(n^2)$ of these "unordered" insertions.

Fun-Sort runs in n phases. In phase i, for $i = 1, \ldots, n$, we repeatedly binary search for the current content a_i of cell i until the search is successful and "finds" the element in cell i. If the search fails we find an index h such that $a_{h-1} < a_i < a_h$. We then either swap the bad pair $(i, h-1)$ if $i < h-1$, or the bad pair (h, i) if $h < i$. The phase continues with a search for the new content of cell i. Note that Fun-Sort is not an inversion-fixing algorithm in the strict sense (because we do not repair wrongly ordered pairs during a binary search), but a failed search is always followed by an exchange swap.

Theorem 9. Fun-Sort *sorts n elements in time $O(n^2 \log n)$.*

Proof. We show by induction that after phase i, for $i = 1, \ldots, n$, the elements stored in cells $1, \ldots, i$ are ordered correctly (although they are not necessarily the smallest i elements).

The claim is clearly true for $i = 1$. If in phase $i > 1$ a search fails and returns index h and $h < i$ then we swap (h, i). But $a_{h-1} < a_i < a_h$ so this does not destroy the sortedness of the first $i - 1$ cells. And if the search for a_i is successful we have $a_{i-1} < a_i$, so the first i cells are sorted at the end of phase i.

Each binary search takes time $O(\log n)$. There are n successful searches, one per phase. And there are at most $\binom{n}{2}$ unsuccessful searches by Lemma 1. Thus, the total time is $O(n^2 \log n)$. □

The following example shows that the bound of Theorem 9 is tight. Assume $n = 2k + 1$ for some $k \geq 1$ and the initial sequence is $k + 2, k +$

$3, \dots, 2k, 2k+1, 1, 2, \dots, k, k+1$. In the first phase, we first try to locate $k+2$. The binary search ends with $h = n+1$, so we swap $(1, n)$. Then cell 1 contains $k+1$, and we search for $k+1$. The search ends with $h = n$, etc. After $k+1$ unsuccessful searches, the sequence of numbers is $1, k+3, \dots, 2k, 2k+1, 2, 3, 4, \dots, k+1, k+2$, and the phase ends with a successful search for 1. Analogously, the next phases will have $k, k-1, \dots$ unsuccessful searches.

Note that we actually have some liberty in the definition of binary search regarding how to break ties. If $x = a_m$ we continue the search in the left half, but we could also continue in the right half or even stop immediately and return index m. These three variants always produce the same result on sorted arrays, but not on unsorted arrays. Fun-Sort works fine with all variants, just the correctness proof becomes slightly more involved for the other two variants.

6 Conclusion

Quite a few interesting questions still wait for answers:

- There is one swap sequence that can sort in one cycle, whereas arbitrary swap sequences may need up to $n-1$ cycles. Is there a characterization of swap sequences that need at most k cycles, for $1 \le k \le n$? And which mini-cycle needs the fewest repetitions to sort any input?

- Is there an inversion-fixing algorithm in the strict sense based on unsorted binary searches? In particular, how efficient is the following algorithm (if it works at all): binary search on a fixed search path and swap elements which are in the wrong order (i.e., on each level of the path have lower bound, search key, and upper bound ordered such that the binary search follows the fixed path)?

- Is it true that Fun-Sort searches at most n times for each element of the sequence? This would also imply the $O(n^2 \log n)$ runtime of Fun-Sort.

- We implemented unordered binary search and observed the at the first view somewhat surprising fact that binary searches for $i = 1, \dots, n$ in a random permutation of $1, \dots, n$ end on the average very close to cell i, i.e., the correct location for element i (it seems to end exactly at i if $n = 2^k - 1$, i.e., if the binary search always divides the search interval into equal-size halves). We only have the following quick, dirty, and wrong proof (but this can sometimes be a reasonably convincing approach, as for example demonstrated in [4]). Let $Pos(i,n)$ denote the expected index found by a binary search for i. In the first step of the binary search, we go left with probability $\frac{n-i}{n}$, and we go right with probability $\frac{i}{n}$. In each half of the input, i has expected rank $\frac{i}{2}$. Thus (and this 'thus' is really not correct, even though it looks convincing at the first glance),

$$Pos(i,n) = \frac{n-i}{n} \cdot Pos(\frac{i}{2}, \frac{n}{2}) + \frac{i}{n} \cdot (Pos(\frac{i}{2}, \frac{n}{2}) + \frac{n}{2}) = Pos(\frac{i}{2}, \frac{n}{2}) + \frac{i}{2}\,.$$

Since $Pos(1,n) = 1$, we have $Pos(i,n) = i$.

We ask: Where does the binary search for i end on the average, and what would be a correct proof?

Acknowledgements

We thank Otfried Cheong for providing a slightly smaller example of a swap-sequence that cannot sort all inputs. We also thank the unknown referees whose valuable comments will undoubtedly benefit this paper. And last not least, we thank the Second Cup Coffee Shop at the University of Waterloo for providing a stimulating problem solving environment over the past year.

References

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

A preliminary version was published with a different title: An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on Theory of Computation (STOC'83)*, pages 1–9, 1983.

[2] E. A. Bender and S. G. Williamson. Periodic sorting using minimum delay, recursively constructed merging networks. *The Electronic Journal of Combinatorics*, 5(1), 1998. #R5.

[3] N. G. de Bruijn. Sorting by means of swapping. *Discrete Mathematics*, 9:333–339, 1974.

[4] R. Fleischer and P. Sanders. Asymptotic complexity from experiments? A case study for randomized algorithms. In *Proceedings of the 4th Workshop of Algorithms and Engineering (WAE'00)*, 2000. To appear. A full version of this paper will be submitted as book chapter [9].

[5] B. A. Julstrom. Slow sorting: A whimsical inquiry. *ACM SIGCSE Bulletin*, 24(3):11–13, 1992.

[6] M. Kik. *Efficient comparator networks.* Ph.D. thesis, Institute of Computer Science, University of Wrocław, Poland, 2000.

[7] M. Kik, M. Kutyłowski, and G. Stachowiak. Periodic constant depth sorting networks. In *Proceedings of the 11th Annual Symposium on the Theoretical Aspects of Computer Science (STACS'94).* Springer Lecture Notes in Computer Science 775, pages 201–212, 1994.

[8] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 2. edition, 1998.

[9] C. McGeogh, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. Searching for Big-Oh in the data: Inferring asymptotic complexity from experiments. In preparation.

[10] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, 1999.

[11] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, Cambridge, England, 1995.

[12] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, and R. Wanka. Periodification scheme: Constructing sorting networks with constant period. *Journal of the ACM*, 47(5):944–967, 2000.

[13] Y. Rabani, A. Sinclair, and R. Wanka. Local divergence of Markov chains and the analysis of iterative load-balancing schemes. In *Proceedings of the 39th Symposium on Foundations of Computer Science (FOCS'98)*, pages 694–705, 1998.