Fun-Sort — Or the Chaos of Unordered Binary Search

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

April 25, 2003

Abstract

Usually, binary search only makes sense in sorted arrays. We show that insertion sort based on repeated "binary searches" in an initially unsorted array also sorts n elements in time $\Theta(n^2 \log n)$. If n is a power of two then the expected termination point of a binary search in a random permutation of n elements is exactly the cell where the element should be if the array was sorted. We further show that we can sort in expected time $\Theta(n^2 \log n)$ by always picking two random cells and swapping their contents if they are not ordered correctly.

1 Introduction

Comparison based sorting algorithms apply four paradigms: selection, insertion, exchanging, and merging [3, page 73]. Most methods operate by maintaining an invariant (of sortedness) on an increasingly longer segment of the array, or, as in the case of Quicksort, at least strong invariants on the progress made thus far. Shellsort, from one point of view, maintains sorted

*Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada. Email: {biedl, tmchan, ja3king, imunro}@uwaterloo.ca.

†MIT Laboratory for Computer Science, Cambridge, MA 02139, USA. Email: edemaine@mit.edu.

‡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.

subarrays of increasingly greater length as the process continues. From another point of view, Shellsort is delightfully chaotic. At any stage, one is presented with a subsequence which one hopes is reasonably close to sorted order, and then one is to complete the task by using linear Insertion-Sort.

We study *exchange sorting algorithms*, i.e., algorithms that compare pairs of array elements and swap them if they are out of order. In particular, we consider the algorithm Fun-Sort[1] that repeatedly moves values from current to "more likely" locations by performing a "binary search". Since the array is initially unsorted, a "binary search" will most likely end up at the wrong array position, resulting in rather chaotic behavior of the insertion sort algorithm. Nevertheless, we were able to adapt the method to correctly perform a sort in the (rather poor) worst-case runtime by $\Theta(n^2 \log n)$. 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 in a randomly permuted array a "binary search" for an element of rank i will on the average end at position i. We also ask whether the paradigm can be further developed to give substantially better behavior in the worst case.

This paper is organized as follows. In Section 2 we define the model of exchange sorting algorithms. In Section 3 we introduce Guess-Sort, a simple randomized exchange sorting algorithm which runs in expected time $O(n^2 \log n)$. We also give a new direct proof of an old theorem that $n-1$ rounds of comparing all adjacent cells, in any order, suffice to sort. In Section 4 we show that Fun-Sort, which is Insertion-Sort where insertions are guided by "binary search" in the initially unsorted array, sorts in time $O(n^2 \log n)$. We conclude with a few open problems in Section 5.

2 Basics

Let an array a hold n elements from a totally ordered set. When convenient, we will assume that these elements are distinct, all results also hold in general. Denote the contents of cell i by a_i, for $i = 1, \ldots, n$. Our goal is to sort these elements in non-decreasing order. In an exchange sorting algorithm we 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 is between zero if the array is correctly sorted, and

[1]after the conference series *Fun with Algorithms*

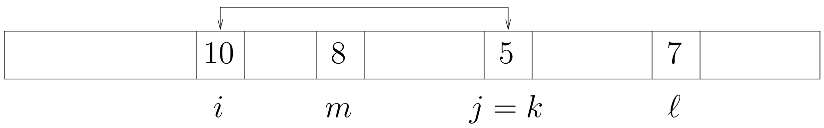

Figure 1. If we swap the bad pair (i, j) then the good pair (k, ℓ) becomes a bad pair. However, the bad pair (i, ℓ) becomes a good pair. As an unintended (but pleasant) side effect, we also turn the bad pairs (i, m) and (m, j) into good pairs.

$\binom{n}{2}$ if the array is sorted in reverse order.

Sorting algorithms based on finding and swapping bad pairs are called *exchange sorting algorithms* [3, page 73]. A *primitive comparison* compares two adjacent cells. A *primitive exchange sorting* algorithm only does primitive comparisons. In an *oblivious* sorting algorithm the sequence of comparisons is fixed in advance, i.e., it is independent of the input sequence or the outcome of individual comparisons. Bubblesort is an example of an oblivious primitive exchange sorting algorithm.

De Bruijn showed that adding arbitrary comparisons to an oblivious primitive sorting algorithm yields another sorting algorithm. As a consequence (also attributed to de Bruijn by some authors [7, Fact 1][6, Prop. 1.9], although his paper does not mention this corollary) we see that doing n rounds of all $n-1$ primitive comparisons, in arbitrary order, is already an oblivious primitive sorting algorithm (because the n phases of Odd-Even Transposition Sort [3, page 240] can be found in the augmented sorting algorithm). Later, de Bruijn's theorem was rediscovered [8, 4, 1] and the corollary was slightly improved; it was shown that $n-1$ rounds of all primitive comparisons are sufficient to sort. In Section 3 we give a new direct proof without reducing to Odd-Even Transposition Sort. Note that the bound of $n-1$ rounds is tight. If we want to sort the input sequence $1, 0, \ldots, 0$ by repeatedly comparing the pairs $(n-1, n), (n-2, n-1), \ldots, (1, 2)$ in this order then we need $n-1$ rounds to move the '1' to the rightmost position.

The following simple observation is folklore and implies that we can sort by repeatedly swapping bad pairs.

Lemma 1. *Swapping a bad pair strictly decreases the number of inversions.*

Proof. Consider swapping a bad pair (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 cell i becomes smaller and cell j becomes larger, (k, ℓ) can only turn bad if $k = j$ or $\ell = i$.

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; 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. $\square$

Corollary 2. *Any exchange sorting algorithm terminates after swapping at most* $\binom{n}{2} = O(n^2)$ *bad pairs.* $\square$

If we could somehow identify the indices of bad pairs we could sort in $O(n^2)$ time. Unfortunately, we cannot expect to get this information for free. In Section 3 we show that choosing the comparisons randomly yields a sorting algorithm with expected running time $O(n^2 \log n)$. In Section 4 we use "binary search" (in an initially unordered array) to find bad pairs in logarithmic time. The resulting algorithm can be seen as Insertion-Sort based on "unordered binary searches" to locate the position for an insertion. It runs in time $O((n + F) \log n) = O(n^2 \log n)$, where F is the number of inversions in the input array.

3 Two Simple Exchange Sorting Algorithms

We first show that we can sort by performing, $n-1$ times, all $n-1$ primitive comparisons, in any order. Bubblesort, for example, performs $n-1$ iterations, each of which compares the pairs $(1, 2), (2, 3), \ldots, (n-1, n)$. A slightly weaker theorem, that $n-1$ repetitions of the same sequence of all primitive comparisons suffice to sort an array, was shown in [8, 4, 1], but their proofs can easily be adopted to also show our stronger Theorem 4. We give a direct proof which is not based on Odd-Even Transposition Sort. It becomes simpler when using the well-known 0-1 Principle (see [3, Theorem Z on page 223], for example).

Theorem 3 (0-1 Principle). *If an oblivious sorting algorithm sorts all sequences of '0's and '1's then it sorts all sequences of arbitrary values.* $\square$

Theorem 4. $n-1$ *rounds of all* $n-1$ *primitive comparisons, in any order, suffice to sort an array of* n *elements.*

Proof. We assume an input of '0's and '1's. Let k be the number of '1's in the input sequence. Since the claim is trivially true if the input only consists of '0's or '1's, we may assume that $0 < k < n$. Denote the '1's from left to right by $T_1, T_2, \ldots, T_k$. We claim that after round i every entry T_j, for $j = 1, \ldots, k$, is in cell $\min\{i + 2j - k, n + j - k\}$ or further to the right, and prove this by induction on i. In particular, T_j reaches its final destination $n + j - k$ after at most $n - j \le n - 1$ rounds.

Since $2j-k \le j$ for $j \le k$, the claim is true for $i = 0$. Now consider round i, for some $i \ge 1$. 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 current round 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 round i. If T_j was at that time directly to the left of T_{j+1} then it was in cell $\min\{i+2j-k, n+j-k\}$ or further to the right, and the claim holds. Otherwise, it started round i left of a '0' and moved at least one cell to the right during the round. Since it was at least in cell $\min\{(i-1)+2j-k, n+j-k\}$ after round $i-1$, it moved at least to cell $\min\{i+2j-k, n+j-k\}$ during round i, which proves the claim. □

We note that the worst-case inputs are exactly the maximal elements of de Bruijn's partial order on input permutations (see [2] for details), and we implicitly proved an analogue to his Theorem 6.1. According to that theorem, it is sufficient to prove correctness for maximal elements of the partial order (in his case there was exactly one maximum element).

Instead of comparing all adjacent cells, which is a rather tedious way of finding bad pairs, we could try a randomized approach. Guess-Sort, in each step, chooses a random pair of cells, not necessarily adjacent, to compare until the array is sorted. How do we know that the array is sorted? We could, for example, check after every $n-1$ steps whether the array is sorted. This needs $n-1$ comparisons, so we at most double the runtime.

Theorem 5. 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 $p = \Theta(\frac{m}{n^2})$, so the expected number of tries until we hit a bad pair is $\frac{1}{p} = \Theta(\frac{n^2}{m})$. By Lemma 1, the number of bad pairs decreases each time we swap a bad pair, so the expected number T_n of swaps until the array is sorted is bounded by

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

□

The proof above is similar to the proof of the coupon collector's problem [5, page 57]. Similarly to [5], we can find an upper bound of $\lim_{n\to\infty} \frac{V_n}{n^4} = \frac{\pi}{6}$ for the variance V_n. The bound would be tight if the number of bad pairs would decrease by 1 in each swap, which is not case.

The bound of Theorem 5 is tight. Guess-Sort needs expected time $\Theta(n^2 \log n)$ to sort the sequence $2, 1, 4, 3, \ldots, n, n-1$. This is an application of the coupon collector's problem: 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 pairs $(1, 2), (3, 4), \ldots$

Consider Primitive Guess-Sort, a variant of Guess-Sort that always chooses a random pair of adjacent cells. Primitive Guess-Sort would sort the sequence above in expected time $O(n \log n)$. It is easy to see that any input sequence will be sorted in expected time $O(n^2 \log n)$, i.e., Primitive Guess-Sort is not worse than Guess-Sort. To prove the claim, observe that we need an expected $\Theta(n \log n)$ steps until we have done each primitive comparison at least once. By Theorem 4, $n-1$ such phases suffice to sort. But is this bound tight? Experiments suggest that Primitive Guess-Sort needs an expected $\Theta(n^2 \log n)$ steps to sort the input $\frac{n}{2}+1, \frac{n}{2}+2, \ldots, n, 1, 2, \ldots, \frac{n}{2}$ (assuming n is even). Note that the coupon collector's problem only guarantees a lower bound of $\Omega(n^2 \log\log n)$ steps. This follows from the observation that we must hit each pair of adjacent cells in the middle half of the array at least $\frac{n}{4}$ times each to move the leftmost $\frac{n}{4}$ elements to their correct positions at the right end of the array.

4 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 rebel and apply the same procedure to an unsorted array. Note that a standard implementation of binary search can also "search" in an unsorted array; the sortedness is only needed to show correctness of the binary search procedure. Therefore, we also speak of binary search when we apply the procedure to an unsorted array.

We give a short description of *binary search*. Let x be the search element. 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 $x \le a_m$, 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 the search for x can fail even though x is contained in the array, and that encountering x during the search (i.e., $a_m = x$ at some time) does not cause the search to halt.

Fun-Sort is an in-place variant of Insertion-Sort. It performs repeated insertions guided by binary search into an initially unsorted array. We show that $O(n^2)$ of these insertions suffice to sort the array.

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 exchange algorithm in the strict sense (because we do not repair wrongly ordered pairs during a binary search), but a failed search always identifies a bad pair which is subsequently swapped.

To illustrate Fun-Sort, we demonstrate phase 1 when the input array is $(12, 14, 8, 1, 20)$. We first binary search for $a_1 = 12$. The binary search starts with $(l, m, h) = (0, 3, 6)$. Since $a_3 = 8 < 12$, we set $l = 3$ and $m = 4$. Since $a_4 = 1 < 12$, we set $l = 4$ and $m = 5$. Since $a_5 = 20 \geq 12$, we set $h = 5$. Now $l+1 = h$, so the binary search stops and returns position $h = 5$. Since we are in phase 1, we swap the pair $(1, 4)$. The array now contains $(1, 14, 8, 12, 20)$, and we continue binary searching for $a_1 = 1$. This search is successful (it ends in position 1), ending phase 1.

Note that, incidentally, 12 ended up near its correct final position (position 3) in the first step of phase 1. Indeed, we can show that binary searching for the i-th item (in increasing order) returns exactly array position i on the average if all input permutations are equally likely and if $n = 2^k - 1$, so Fun-Sort just does the right thing when it moves an item to cell $h-1$ (or h) after an unsuccessful search.

Proposition 6. *Let $n = 2^k - 1$ for some $k \geq 1$. Then the expected termination point of a binary search for i, $i = 1, \ldots, n$, in a random permutation of $\{1, \ldots, n\}$ is cell i.*

Proof. Let d_i be the expected value of the difference of the array positions where the searches for i and $i+1$ end. Instead of binary searching in a random permutation of $\{1, \ldots, n\}$, we think of searching in a complete binary tree with k levels where the elements $\{1, \ldots, n = 2^k - 1\}$ are randomly placed on the nodes. At an interior node containing element z, the search for x proceeds to the left child if $x \leq z$ and to the right child otherwise. Note that we do not stop the search at the interior node if $x = z$; a search always ends at some leaf.

If the search path for i passes through the node containing i we say the search *hits* i. If the search does not hit i then the search paths for i and $i+1$ are identical. Otherwise, if the search for i hits i in node v at level r, for some $r = 0, \dots, k-1$, then the search for i ends at some expected position x in the left subtree of v. Since the set of all possible left subtrees is identical to the set of all possible right subtrees, the search for $i+1$ ends at the same expected position x in the right subtree of v (where the expectation is over all placements such that we hit i at level r).

So the difference in the positions between the search for i and $i+1$ depends only on the level r where the search for i hits i. We could now write down the probabilities that we hit i at a particular level r and what that would mean for the offset, but we can also observe that the argument above shows that d_i is actually independent of i, so we have $d_1 = d_2 = \cdots = d_n$. Since the search for '1' always ends in cell 1 and the search for '$n+1$' always ends in cell $n+1$, we have $d_1 + \cdots + d_n = n$ and thus $d_1 = \cdots = d_n = 1$, proving the claim. $\square$

Theorem 7. Fun-Sort *sorts* n *elements in time* $\Theta((n+F)\log n) = \Theta(n^2 \log n)$, *where* F *is the number of inversions in the input array.*

Proof. First we prove that the runtime is $O(n^2 \log n)$. This is done by induction on the number of phases such that after phase i, for $i = 1, \dots, n$, the elements stored in cells $1, \dots, i$ are ordered correctly (although they are not necessarily the smallest i elements).

The claim is clearly true for $i = 1$, so assume $i > 1$. By induction, the first $i-1$ cells are ordered correctly at the beginning of phase i. If now a search fails (i.e., we are not done with phase i yet) then it returns an index $h \neq i$. If $h > i$ the subsequent swap of $(i, h-1)$ does not affect the sortedness of the first i cells. If $h < i$ then we swap (h, i). Since binary search guarantees $a_{h-1} < a_i < a_h$ this swap also maintains the sortedness of the first $i-1$ cells. If the search for a_i is successful (and phase i ends) then we have $a_{i-1} < a_i$, and the first i cells are sorted as claimed.

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

To demonstrate that this bound is tight in the worst case, we give a family of examples. If n is even, we add another item ∞ to the input to get an odd number of items. Thus, we can assume $n = 2k+1$ for some $k \geq 1$. Consider the input $k+2, k+3, \ldots, 2k, 2k+1, 1, 2, \ldots, k, k+1$. 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 $1, k+3, \ldots, 2k, 2k+1, 2, 3, 4, \ldots, k+1, k+2$, and the phase ends with a successful search for 1. Analogously, the next phases will have $k, k-1, \ldots$ unsuccessful searches. □

An alternate proof of the upper bound can be obtained by demonstrating that no two searches in the same phase will terminate at the same location. This follows from a few observations: First, in phase i, if an unsuccessful search terminates to the right of i at position p, the next search will end to the left of p. Next, in phase i, if an unsuccessful search terminates in the range $[0, i-1]$ at position p, the next search will terminate at position $p+1$. Finally, in phase i, all unsuccessful searches terminating to the right of i occur before all unsuccessful searches terminating to the left of i.

Note that we actually have some liberty in the definition of binary search regarding how to break ties. If $x = a_m$ we continued the search in the left half, but instead we could also continue the search in the right half (and at the end return the last index ℓ as the result of the search), or even stop immediately and return index m as the result of the search. In the former case, Fun-Sort would swap the bad pair $(\ell+1, i)$ if $\ell < i$ and the bad pair (i, ℓ) if $i < \ell$. These variants always produce the same result on sorted arrays, but not on unsorted arrays. It is easy to adapt the proof of Theorem 7 to these Fun-Sort variants, so the theorem still holds.

5 Conclusions

A few interesting questions still await answers:

- Prove a lower bound of $\Omega(n^2 \log n)$ for the expected running time of Primitive Guess-Sort.

- Is there an exchange sorting 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)?

- Proposition 6 only holds for values of n of the form $n = 2^k - 1$. For other values of n, the search for i in a random permutation of $\{1, \ldots, n\}$

still ends approximately in cell i on the average, but not exactly cell i. However, an exact analysis seems to be non-trivial.

The main questions, though, are:

- Can Fun-Sort be adapted to run in $o(n^2)$ (or even $O(n^2)$) worst-case time?
- What is the average-case runtime of Fun-Sort? Proposition 6 seems to indicate that it might be rather fast on the average.

Acknowledgements

We thank Otfried Cheong and two unknown referees for their valuable comments. We also thank the Second Cup Coffee Shop next to the University of Waterloo for providing a stimulating problem solving environment.

References

[1] Biedl, T., Chan, T., Demaine, E. D., Fleischer, R., Golin, M., and Munro, J. I. Fun-Sort. In *Proceedings of the 2nd International Conference* FUN with Algorithms 2*, Island of Elba, Italy (FUN'01)* (2001), E. Lodi, L. Pagli, and N. Santoro, Eds., Carleton Scientific, Proceedings in Informatics 10, pp. 15–26.

[2] DeBruijn, N. Sorting by means of swapping. *Discrete Mathematics 9* (1974), 333–339.

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

[4] Krammer, J. G., Bernard, E. G., Sauer, M., and Nossek, J. A. Sorting on defective VLSI-arrays. *INTEGRATION, the VLSI Journal 12* (1991), 33–48.

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

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

[7] Rabani, Y., Sinclair, A., and Wanka, R. 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)* (1998), pp. 694–705.

[8] Schröder, H. Partition sorts for VLSI. In *Proceedings of the 13th GI-Jahrestagung, Informatikfachberichte 73*, I. Kupka, Ed. Springer-Verlag, Heidelberg, 1983, pp. 101–116.