

RESEARCH ON THE AMOUNT OF INFORMATION
NEEDED TO RESTORE THE ORIGINAL ORDER OF FOUR
OR EIGHT ELEMENTS LISTS, WHEN USING DIFFERENT
SORTING ALGORITHMS

Metodi L. Dimitrov[✉], Tzvetomir I. Vassilev

Received on June 21, 2024

Presented by Ch. Roumenin, Member of BAS, on July 30, 2024

Abstract

Data sorting is essential in most software applications, but sometimes elements need to be restored to their original order after processing. If this restoration happens long after sorting or on a different computer, additional information is needed to restore the order. This work explores the information needed to restore sequences of 4 and 8 elements. The following sorting algorithms were studied: parallel neighbour (odd even) sort, insertion sort, bubble sort, shell sort, merge-insertion sort, and merge sort. For each algorithm, the amount of information required in bytes to restore the original order of the elements was determined.

Key words: sorting, algorithm, restore sequence order, sorting reversal, parallel neighbour sort, odd even sort, brick sort, insertion sort, bubble sort, shell sort, merge sort, merge-insertion sort, Ford-Johnson

Introduction. Sorting algorithms are widely used in software development. Over the years, many such algorithms have been developed and studied in terms of speed, stability, used additional space, etc. Some comparison and applications of different sorting algorithms can be seen in [1–4].

This study is financed by the European Union – NextGenerationEU, through the National Recovery and Resilience Plan of the Republic of Bulgaria, project No BG-RRP-2.013-0001-C01.
<https://doi.org/10.7546/CRABS.2024.09.07>

In practice, sorting algorithms can be classified according to a variety of criteria. One way of classification is according to the way they process the elements in the sequences, as discussed in [5]:

- In-place algorithms – these algorithms process the elements in the same address space in which they are located, by swapping their positions;
- Algorithms for sorting read-only collections – In some cases, list elements can only be read and cannot be moved, swapped, or overwritten. In these situations, algorithms use additional address space for sorting.

In practice, preserving the original order of elements is often important. The second type of algorithms can be used for this, but they work only on the same computer with access to the memory where the elements are stored. If the sorted list needs to be transferred to another machine, we cannot use these algorithms. Instead, we must send the sorted list along with additional information to restore the original order of the elements.

As an example, to illustrate the need for this functionality, consider compressing data by storing the first element of a list and the differences between subsequent elements. In sorted lists, these differences are smaller, thus the compression is improved. We then send the compressed data and additional information to restore the original order to another computer.

Different sorting algorithms require varying amounts and types of operations to sort a sequence. These operations include moving, swapping, inserting, comparing elements, etc. Each operation can be recorded during sorting and later executed in reverse order, to restore the original order of the sequence.

This paper compares and analyzes the information needed to restore the original order of elements in lists of 4 and 8 integers sorted using different algorithms.

Lists of 4 and 8 numbers were chosen because if we take a list of 8 integers, we could say that they are located at positions from 0 to 7, which we can assume as the positions in a byte. These positions (and the numbers), we can easily swap, move, etc.

Some sorting algorithms and their applications when restoring the original order of elements is required. *Parallel neighbour (odd even) sort.* Parallel neighbour sort [6] is a comparison algorithm that compares all elements. To restore the original order of a list of 4 elements it requires 6 bits (1 bit per comparison). For a list of 8 elements, the algorithm generates 28 bits (28 comparisons). The algorithm first compares all odd elements, then all even elements, repeating until the list is sorted. For a list of 4 elements, the worst case requires two rounds of comparisons; for 8 elements, four rounds.

This method allows early termination if the list is already sorted. That is why, for a list of 4 elements 3 consecutive zeros from comparisons indicate sorted list. For an 8-element list, seven consecutive zeros suffice. This reduces the bits needed to restore the original order.

Insertion sort – version 1. Insertion sort assumes the first element is

already sorted. Therefore, sorting starts from the second element, indicating where each unsorted number should be placed to keep the list sorted to the left of the current number [7].

The data needed to restore the original order may contain the positions (in binary) where each element was inserted. To reduce bit usage, sorting up to the second element uses 1 bit (0 or 1) to indicate the position to be inserted at. Sorting the third and fourth elements uses 2 bits each for their positions, and so on.

Following this algorithm, restoring the original order of a 4-element list requires (generates) 5 bits, and an 8-element list requires 17 bits. Unlike the previous method, this approach cannot reduce bits for already sorted or semi-sorted lists, as every position must be recorded, even if the element remains in place.

Insertion sort – version 2. Let us consider another version of insertion sort, which instead of remembering the position to which each element should move, stores the number of swaps a given element needs to make to move up to its position. The current implementation of the algorithm can be summarized as follows:

1. Whenever an element starts moving up to its correct position, we write a 1 for each swap it performs with a preceding element;
2. If the element reaches its final position and no further swaps are needed, we write a 0;
3. However, if an element reaches the first position in the sequence, we do not write a 0. In this case, the number of swaps indicates that the element has reached its final destination.

We can calculate that to restore the original order of the list of 4 elements, in the worst case, we will need a maximum of 6 bits of information (6 swaps), and for a list of 8 elements, in the worst case, we will need a maximum of 28 bits of information (28 swaps). This method allows reducing the number of bits, if an element does not need to climb to the top of the list.

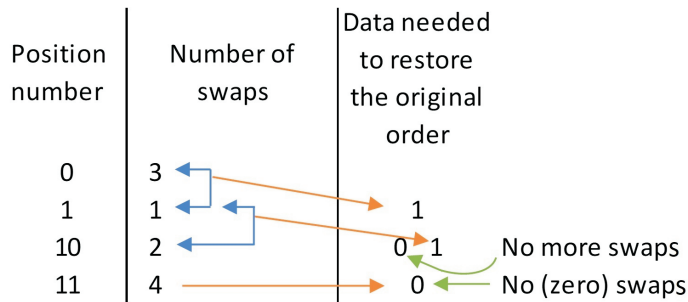


Fig. 1. List sorting and recovery data extraction with version 2 of insertion sort

Figure 1 shows how we can sort an example sequence consisting of the elements (3, 1, 2, 4). After sorting, we get bits 1010, which contain the information

needed to restore the original order of the sequence. From the example, we can see that for this particular sequence, instead of the maximum 6 bits, we will obtain 4 bits for restoring the original order of the sequence.

Insertion sort – version 3. As we know, binary search is one of the methods to find the exact position in a sorted list with fewer comparisons [8–10]. We can apply this search in insertion sort. The result of each comparison we will write down as 0 or 1.

Using binary search, we will need fewer bits in certain situations. This is possible because when the array is divided for searching into two, it may turn out that one subarray has fewer elements. Therefore, we will need fewer comparisons in some cases.

We can calculate that to restore the original order of the list of 4 elements, in the worst case, we will need a maximum of 5 bits of information (5 comparisons), and for a list of 8 elements, in the worst case, we will need a maximum of 17 bits of information (17 comparisons).

Bubble sort. As we know, bubble sort performs consecutive passes and comparisons through the list, with one of the elements going to its place at each pass (the smallest or the largest) [7]. After each comparison we can write down 1 (the elements are swapped) or 0 (the elements are not swapped). Therefore, we will need 6 bits to restore the original sequence of 4 elements. If our sequence has 8 elements, then we need a total of 28 bits (28 comparisons).

With bubble sort, we can reduce the number of bits, if all comparisons in a pass show that no further swaps are needed.

Shell sort. Shell sort is a sorting algorithm that sorts in several passes, where elements separated by a gap are swapped (sorted) at each step [7]. After swapping all possible elements is made in a given pass, the gap is reduced by a given step. This process is repeated until the gap finally reaches 1. Then, the algorithm performs insertion sort to find the right place for each element. We can think of this algorithm as an insertion sort that performs several preliminary steps. The purpose of this pre-sorting is to reduce the number of comparisons performed by the insertion sort.

The essence of this algorithm is the gap and the step by which it decreases (we can call the whole sequence a sequence of gaps). Over the years, many such gap sequences have been proposed [11–14]. In [13] lists containing between 2 and 16 numbers were sorted with different gap sequences. As a result, the optimal gap sequences were determined. Namely, for sequence of 4 elements, the best discovered gap sequence consist of only 1 element and this element is 1. This means that for a sequence of 4 elements, no pre-sorting is done, but instead, insertion sort is applied directly.

For a sequence of 8 elements, according to [13], the best sequence for the gap is 7, 5 and 1. This means that we will need to sort the elements by comparing elements with numbers 1 and 8 (at gap 7) and elements with numbers 1 and 6, 2

and 7, and 3 and 8 (at gap 5). After these comparisons, we can switch to insertion sort.

The result of the comparisons at gap 7 and 5 can be recorded as 1 or 0 (1 bit), indicating whether the two compared elements have been swapped or not.

When performing the subsequent insertion sort, we can use the variants 1, 2, and 3 discussed above. This is why there are 3 variants for shell sort in Tables 1 and 2. Shell sort version 1 uses insertion sort version 1, Shell sort version 2 uses insertion sort version 2 and sort version 3 uses insertion sort version 3.

Merge sort. With merge sort, we merge two sorted list into one sorted list [7]. To achieve that, we can divide a list into two smaller lists, then each of them into other 2 lists and so on until we reach several lists of 1 element.

When merging two lists, we can determine which list each element came from by writing 0 (first list) or 1 (second list). For example, if we merge the lists (2,3) and (4,5), we can write 001, which will tell us that to obtain the sorted sequence (2,3,4,5), we took the first two elements from the first list, and the third element from the second list. Since we always have an equal number of elements in the two lists we are merging, we do not record where the last element came from, as this is implied.

Following this sorting algorithm (mere sort) we can calculate that with 4 elements we will need 5 bits of information to restore the original ordering of the elements and with 8 elements we will need 17 bits. This algorithm does not allow using fewer bits on already sorted lists.

Merge-insertion sort (Ford–Johnson algorithm). Merge-insertion sort [15] is one of the algorithms that adheres to the minimal theoretical number of comparisons for sorting a given sequence (for small sequences) or approaches it [8–10, 16–18]. Since this algorithm is based on comparisons and rearrangement of elements after them, we can store the result of each comparison (as 0 or 1), and based on this data, we can restore the original sequence.

Because the algorithm uses binary search, the number of comparisons can be reduced in some cases, as discussed in the insertion sort v3.

For 4 elements, in the worst case, this sorting algorithm will need 5 bits (5 comparisons) of information to restore the original ordering of the elements and for 8 elements it will need 16 bits (16 comparisons).

Merge-insertion sort (Ford–Johnson algorithm) – version 2. When using Merge-insertion sort and we have sequences of 4 elements, we can reduce the number of bits required to reconstruct the original order by using the number of bits to encode additional information.

For example, if we need to insert b2 into b1, a1, a2, instead of writing the result of each comparison as 0 or 1, we could reduce the number of bits as follows:

- zero bits mean that after the 1-st comparison, b2 is placed before b1 and no further comparisons are needed;

- 1 bit with value 0 indicates that b2 is greater than b1 and b2 is less than a1;
- 1 bit with value 1 indicates that b2 is greater than b1 and b2 is greater than a1.

The disadvantage of this approach is that it does not allow chaining the result obtained of several lists of 4 elements.

For 4 elements, in the worst case, this sorting algorithm will need 4 bits for 5 comparisons – compare el. 1 (a1) and el. 2 (b1), el. 3 (a2) and el. 4 (b2), a1 and a2, b2 and a1, b2 and b1.

Results comparison and analysis. Table 1 compares the worst-case max amount of information required to restore the original order of a given list of 4 or 8 elements, using different algorithms.

T a b l e 1

Comparison of the information required to restore the original order of lists of 4 and 8 numbers, using different sorting algorithms

Algorithm	Worst case max number of bits to restore the original order of a sequence of 4 integer elements	Worst case max number of bits to restore the original order of a sequence of 8 integer elements	The algorithm can be chained (4 elements / 8 elements)
Odd even sort	6	28	Yes / Yes
Insertion sort v1	5	17	Yes / Yes
Insertion sort v2	6	28	Yes / Yes
Insertion sort v3	5	17	Yes / Yes
Bubble sort	6	28	No / No
Shell sort v1	5	21	Yes / Yes
Shell sort v2	6	32	Yes / Yes
Shell sort v3	5	21	Yes / Yes
Merge sort	5	17	Yes / Yes
Merge-insertion sort	5	16	Yes / Yes
Merge-insertion sort v2	4	–	No / –

Table 2 and Fig. 2 show the performance of the algorithms with real data. This data was obtained by averaging the resulting values from 10 million unsorted lists, filled with random values between 0 and 1 million. According to the data in Table 2 and Fig. 2, we see that for a list of 4 numbers, if we do not have to chain several lists, the most effective algorithm for restoring the original order of a list is the merge-insertion sort v2. The average number of bits it generates is 3.652.

If we want to chain the data (link and put them one after another), generated from several lists of 4 elements, then we have two algorithms with almost identical

T a b l e 2

Average number of bits required to restore the original order of lists of 4 and 8 numbers
(calculated on 10 million unsorted lists)

Algorithm	Bits required to restore the original order of a sequence of 4 integer elements	Bits required to restore the original order of a sequence of 8 integer elements	The algorithm can be chained (4 elements/ 8 elements)	Percentage of the total amount of information in four 8-bit numbers, containing 32 bits	Percentage of the total amount of information in eight 8-bit numbers, containing 64 bits
Odd even sort	6.000	28.000	Yes / Yes	18.75	43.75
Insertion sort v1	5.000	17.000	Yes / Yes	15.63	26.56
Insertion sort v2	5.000	19.283	Yes / Yes	15.63	30.13
Insertion sort v3	4.6524	15.590	Yes / Yes	14.54	24.36
Bubble sort	5.695	25.996	No / No	17.80	40.62
Shell sort v1	5.000	21.000	Yes / Yes	15.63	32.81
Shell sort v2	5.000	17.765	Yes / Yes	15.63	27.76
Shell sort v3	4.6524	20.185	Yes / Yes	14.54	31.54
Merge sort	5.000	17.000	Yes / Yes	15.63	26.56
Merge-insertion sort	4.6522	15.457	Yes / Yes	14.54	24.15
Merge-insertion sort v2	3.652	—	No / —	11.41	—

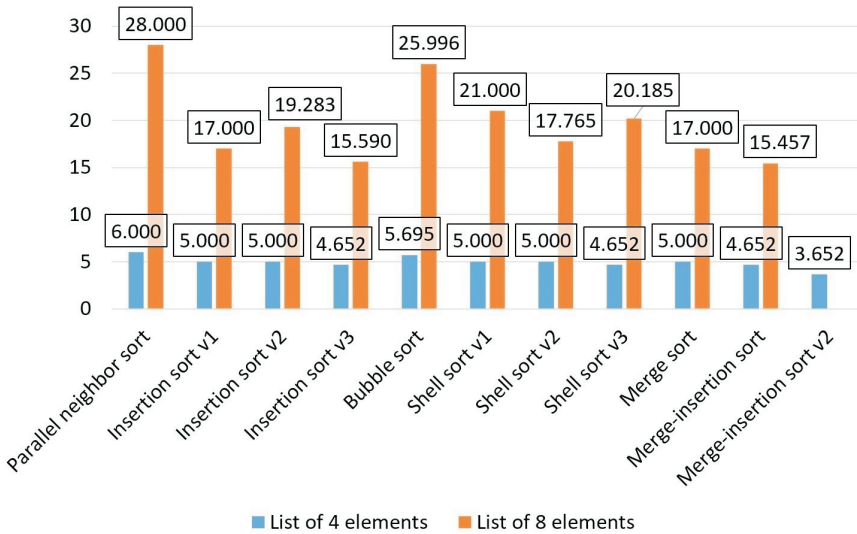


Fig. 2. Average number of bits required to restore the original order of lists of 4 and 8 numbers (calculated on 10 million unsorted lists)

results. They are insertion sort v3 (using binary search) and merge-insertion sort. Chaining the data is possible, because each sequence contains enough data, so we can determine where each of them ends.

When the sequence contains 8 elements, again we have very close results for insertion sort v3 (4.6524 bits) and merge-insertion sort (4.6522 bits), with merge-insertion sort being only slightly more efficient and generating fewer bits needed to restore the original order of the sequence.

From Table 2 we can see that shell sort generates the same number of bits as underlying insertion sort version. That is because for 4 elements, the shell sort turns into an insertion sort.

Conclusions and future work. The current work examines implementations of some of the popular sorting algorithms by evaluating the amount of information (generated after applying each algorithm) needed to restore the original order of lists of 4 and 8 integer elements. From the obtained results, it became clear that for lists of 4 elements, the least amount of information was generated by insertion sort v3 (4.6524 bits) and by merge-insertion sort (4.6522 bits), which is about 14.54% of the total amount of data (assuming each element contains 8 bits). If we do not need to chain data, generated from multiple lists in merge-insertion sort, we can reduce the amount of generated data per list to 3.652 bits.

For a list of 8 elements, the same algorithms have the best performance. Merge-insertion sort generates 15.457 bits on average, which is 24.15% of the total amount of data (assuming each element consists of 8 bits). The insertion sort generates 15.590 bits on average, which is 24.36%.

REFERENCES

- [1] BURADAGUNTA S., J. D. BODAPATI, N. B. MUNDUKUR, S. SALMA (2020) Performance comparison of sorting algorithms with random numbers as inputs, *Ingénierie des Systèmes d'Information*, **25**(1), 113–117, <https://doi.org/10.18280/isi.250115>.
- [2] DUBEY S., S. GUPTA, A. KUMAR, G. KAUR (2023) Visualizing Algorithms: A Comprehensive Exploration of Sorting and Computational Problem-Solving, *IEEE 2023 International Conference on Integrated Intelligence and Communication Systems (ICIIICS)*, 1–6.
- [3] MISHRA A. D., D. GARG (2008) Selection of best sorting algorithm, *Int. J. Intell. Inf. Process.*, **2**(2), 363–368.
- [4] YANG Y., P. YU, Y. GAN (2011) Experimental study on the five sort algorithms, *International Conference on Mechanic Automation and Control Engineering*, Hohhot, China, 1314–1317, <https://doi.org/10.1109/MACE.2011.5987184>.
- [5] CHAN T. M., J. I. MUNRO, V. RAMAN (2018) Selection and sorting in the “restore” model, *ACM Transactions on Algorithms (TALG)*, **14**(2), 1–18, <https://doi.org/10.1145/3168005>.

- [6] HABERMANN A. N. (1972) Parallel neighbor-sort (or the glory of the induction principle).
- [7] NAKOV P., P. DOBRIKOV (2015) Programming = ++Algorithms;, ISBN 954890506X, (in Bulgarian).
- [8] AYALA-RINCON M., B. T. DE ABREU, J. DE SEQUEIRA (2007) A variant of the Ford–Johnson algorithm that is more space efficient, *Inf. Proc. Letters*, **102**(5), 201–207.
- [9] KASAI T., S. SAWATO, S. IWATA (1994) Thirty-Four Comparisons Are Required to Sort 13 Items, *LNCS*, **792**, 260–269.
- [10] PECZARSKI M. (2004) New Results in Minimum-Comparison Sorting, *Algorithmica*, **40**(2), 133–145.
- [11] KNUTH D. E. (2008) *Art of Computer Programming, Volume 3, Sorting and Searching*, Pearson, ISBN 978-8131709832.
- [12] LEE Y. W. (2021) Empirically Improved Tokuda Gap Sequence in Shellsort, arXiv preprint, <https://arxiv.org/pdf/2112.11112>.
- [13] Lee Y. W. (2021) Optimal Gap Sequences in Shellsort for $n \leq 16$ Elements, arXiv preprint, <https://arxiv.org/pdf/2112.11127>.
- [14] PRATT V. R. (1980) *Shellsort and Sorting Networks*, Outstanding Dissertations in the Computer Sciences, Garland Pub., ISBN 9780824044060.
- [15] FORD L. R., S. M. JOHNSON (1959) A Tournament Problem, *American Mathematical Monthly*, **66**, 387–389.
- [16] MANACHER G. K., T. D. BUI, T. MAI (1989) Optimal combinations of sorting and merging, *J. ACM*, **36**, 290–334.
- [17] PECZARSKI M. (2007) The Ford–Johnson algorithm still unbeaten for less than 47 elements, *Inf. Proc. Letters*, **101**(3), 126–128.
- [18] STOBER F., A. WEIS (2023) Lower Bounds for Sorting 16, 17, and 18 Elements, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 201–213.

“Angel Kanchev” University of Ruse,
 8 Studentska St, Building 1, 7017 Ruse, Bulgaria
 e-mails: mdimitrov@uni-ruse.bg, tvassilev@uni-ruse.bg