Best Sorting Algorithm for Nearly Sorted Lists

Curtis R. Cook
Oregon State University

Do Jin Kim
National Semi-Conductor

SUMMARY: Straight Insertion Sort, Shellsort, Straight Merge Sort, Quickersort, and Heapsort are compared on nearly sorted lists. The ratio of the minimum number of list elements which must be removed so that the remaining portion of the list is in order to the size of the list is the authors' measure of sortedness. Tests on randomly generated lists of various combinations of list length and small sortedness ratios indicate that Straight Insertion Sort is best for small or very nearly sorted lists and that Quickersort is best otherwise. Cook and Kim also show that a combination of the Straight Insertion Sort and Quickersort with merging yields a sorting method that performs as well as or better than either Straight Insertion Sort or Quickersort on nearly sorted lists.

1. Introduction

There is no one sorting method that is best for every situation. Some of the factors to be considered in choosing a sorting algorithm include the size of the list to be sorted, the programming effort, the number of words of main memory available, the size of disk or tape units, the extent to which the list is already ordered, and the distribution of values [10].

Key words and phrases: sorting, Quickersort, Shellsort, Merge Sort, Insertion Sort, Heapsort, sorting effort measures
CR Category: 5.31

Author's address: C.R. Cook, Computer Science Department, Oregon State University, Corvallis, OR 97331; D.J. Kim, National Semi-Conductor, Sunnyvale, CA 94086.

Here we report on an investigation to determine the best internal sorting algorithm for nearly sorted lists. Most studies of sorting algorithms derive an expression for the average or expected sorting effort for an arbitrary unsorted list. Rather than considering all possible orderings, our investigation concentrated on lists that were nearly in order. For our measure of "sortedness," we selected the minimum number of elements that can be removed to leave the remaining list sorted. If we call this number k, then the *sortedness ratio* is defined by k/N, where N is the size of the original list. Our empirical study compared the following sorting algorithms on various sized (50, 100, 200, 500, 1,000, 2,000) lists with sortedness ratios 0.00, 0.02, 0.04, . . . , 0.20: Straight Insertion Sort [7], Straight Merge Sort [7], Shellsort [1], Quickersort [12], and Heapsort [14]. We chose these algorithms because they are representative of the various families of sorting techniques.

Loeser [8] in a related study reported the empirical results from a comparison of Quicksort [6] and two of its descendants (Quickersort [12] and qsort [13]), Shellsort [1], stringsort [2], and Treesort [4] on random as well as some specially ordered lists. His special orderings ranged from reverse order to completely in order. In his almost-in-order case, which is nearly identical to ours, he began with an ordered list and randomly selected m elements and set them equal to a different random number. Our algorithm for generating random permutations with m elements out of order randomly selects and deletes m elements from the identity permutation and then randomly inserts them into the list preserving the sortedness ratio. Loeser did not systematically consider various sortedness ratio and list size combinations as we did in our study.

Melhorn [11] developed a new sorting algorithm for nearly sorted lists. He chose the number of inversions as his measure of sortedness. The *number of inversions* of a list of elements is the number of times a larger element appears to the left of a smaller one. His sorting algorithm used a straight insertion sort in conjunction with an AVL-treelike data structure for quick insertion. He showed that his algorithm had a running time of the order of $n(1 + \log (F/n)$ where n is the size of the list and F is the number of inversions.

For lists that are nearly in order, [7, 9] suggest using Straight Insertion Sort; however, no justification or definition of nearly-in-order is given, leading one to suspect that this is folklore. Our test results indicate that Straight Insertion Sort is the best sorting algorithm for small or very nearly sorted lists and that Quickersort is best otherwise. By combining Straight Insertion Sort and Quickersort with merging we obtained a sorting algorithm that, in almost all cases, out-performed either Straight Insertion Sort or Quickersort and, in some cases, its sorting effort was half that of Quickersort.

2. Best Sorting Algorithm and Nearly Sorted

In comparing internal sorting algorithms to determine the best one for nearly sorted lists, we are immediately confronted with the problems of defining what we mean by best sorting algorithm and nearly sorted list. First, we will define best sorting algorithm. Although there is no "best" definition of a best sorting algorithm, we will, nevertheless, define the best sorting algorithm as the one whose weighted sum of the number of comparisons, moves, and exchanges (where an *exchange* is equal to two comparisons or two moves) is minimal. We assume that a *comparison* is a binary comparison, a *move* is typically a transfer from a word in memory to a register or vice versa, and an *exchange* is the interchange of the contents of two words in memory. We feel that equivalencing two comparisons, two moves, and one exchange is realistic for typical computer systems. We will not consider any effort spent in bookkeeping (such as keeping track of pointers or doing simple arithmetic) in the sorting process. We realize the shortcomings of our definition, but feel that its simplicity provides both an honest basis of comparison and insight into the nature of the algorithm. We feel that our measure is a better reflection of the sorting effort than the commonly used number of comparisons.

Defining when a list is nearly sorted is a more difficult problem. Intuitively, a list is nearly sorted if only a few of its elements are out of order. The problem is to define an easily computed measure that coincides with intuition. For the sake of simplicity, we will assume that the list has N records and each record has a distinct key. We will define the *sortedness ratio* of a list with N records by k/N, where k is the minimum number of records that must be removed so that the remaining portion of the list is sorted. For a sorted list, this ratio is zero, and for a completely out of order list, the ratio approaches one. Since the keys of the records in the list are distinct, if the size of the list is n, there is a natural correspondence between the list elements and a permutation of the first n positive integers. Fredman [5] gives an algorithm with an order of $n \log_2 n$ for finding the length of the longest increasing subsequence in a permutation of n integers. Hence, the sortedness ratio of a list is an easily computed measure that coincides with our intuitive notion.

Note that this measure does not take into account the distance of a record from its sorted position, and that the elements in the ordered subsequence are most likely not in their final sorted positions. For example, both lists (1, 2, 3, 4, 5, 7, 6) and (7, 1, 2, 3, 4, 5, 6) have the ratio 1/7; in the first list, all that is required to sort the list is to interchange the 6 and 7, while in the second the 7 must be moved to the other end and the elements 1 to 6 shifted one position. However, this shifting is necessary only if the list has a sequential representation. If the list has a linked representation, then only two links need to be changed to sort both lists. Also notice that in the list (1, 3, 5, 7, 2, 4, 6) the elements in the longest ordered subsequence (1, 3, 5, 7) are only in relative order.

We will not define a nearly sorted list to be one whose sortedness ratio is some arbitrary fraction such as .20. We could just as well have chosen .10, .05, or .15. In this paper, we compare the various algorithms on lists of varying size and a whole range of sortedness ratios less than 0.20.

3. Empirical Test Results

In this section, we give the test results for the five sorting algorithms on nearly sorted lists.

There is no common nomenclature for sorting algorithms and often there are slight variations in the descriptions of the algorithms. The Straight Insertion Sort in Knuth [7] is called an improved version of the Bubble Sort in Martin [10], the sifting-or-sinking method in MacLaren [9], and the Linear Insertion Sort in Elson [3]. In this paper we will use the names and versions of the sorting algorithms as given in Knuth [7] which should be consulted for descriptions of the algorithms.

The five algorithms that are compared on nearly sorted lists are:

(1) Straight Insertion Sort
(2) Quickersort
(3) Shellsort
(4) Straight Merge Sort
(5) Heapsort

It would be virtually impossible to compare all sorting algorithms and their variations on nearly sorted lists. We have attempted to select algorithms that are representative of the various categories of sorting algorithms. The Bubble Sort was *not* selected as one of the algorithms to be compared because the Straight Insertion Sort can be considered an improved version of it. We chose Quickersort rather than Quicksort because Quicksort partitions the original list into three sublists using the first element as an estimate of the median. All elements in the first sublist are less than the median estimate, the median estimate is the second sublist, and all elements in the third sublist are greater than the median estimate. Quickersort does the same partitioning, but uses the middle list element as an estimate of the median. Quicksort and Quickersort work best for good estimates of the median or when the two sublists are nearly the same size. Hence, for nearly sorted lists the middle list element is more likely to be a good estimate of the median than the first list element. Interestingly, the worst case of Quicksort is when the list is already in order.

The test data for the experiment consisted of lists of sizes 50, 100, 200, 500, 1,000, and 2,000 and sortedness ratios of 0.00, 0.02, 0.04, 0.06, . . . , 0.20. Ten lists of each size and sortedness ratio combination were used in the empirical testing. Each list of size N with sortedness ratio k/N was created from the identity permutation as follows: First, k randomly chosen elements were removed from the permutation and randomly inserted into another list of size N. Then the remaining $N - k$ elements of the permutation were inserted in order in the vacant list slots. If in doing so, one of the random k elements lies between and has a value between two of the inserted elements, then the random element and one of the inserted elements are exchanged. Thus, the k elements of the permutation is the smallest subset whose removal leaves the list sorted, and hence the sortedness ratio of the list is k/N. Computer programs were written in Fortran for each sorting algorithm. Besides sorting the input lists, the programs computed the weighted sum of the number of comparisons, moves, and exchanges for each list. Entries in Table I are the average of the weighted sums divided by the size of the list. These allow easy comparison of lists of different sizes.

What can we conclude from our experiment? Table I indicates that either Quickersort or Straight Insertion Sort gave the minimum value for each combination of list size and sortedness ratio. Generally speaking, Straight Insertion performed best on the smallest and very nearly sorted lists and Quickersort was best for the other combinations of list size and sortedness ratios. To be more explicit, the crossover values where Quickersort performed better than Straight Insertion Sort were: 12–14 percent for 50 elements, 6–8 percent for 100, 4–6 percent for 200, 2–4 percent for 500, and 0–2 percent for 1,000 and 2,000 elements. Also note that Straight Insertion Sort was by far the worst on the larger sized lists and sortedness ratios.

Table I also indicates that the Straight Merge Sort and Heapsort do not take into account the sortedness of the list, as they spent almost the same effort sorting the nearly sorted lists for each of the sortedness ratios. Inspection of these algorithms verifies that this is indeed the case. In addition to the five sorting algorithms, tests were performed on modified versions of these algorithms, such as a Revised Heapsort, with no noticeable improvement. We also considered the Natural Merge Sort [7], which works by successively merging the sorted sublists at each end of the list, working toward the middle, and would seem to take advantage of the sortedness of the list. Surprisingly, it performed worse than the Straight Merge Sort for all but the very smallest sortedness ratios. It made slightly fewer moves than the Straight Merge Sort, but many more comparisons.

Many authors, including Knuth and MacLaren, mention that Straight Insertion Sort is the best sorting algorithm when the list is very nearly in order. However, they only give a general justification and an intuitive definition of very nearly sorted. The results of our experiment do verify that they are correct.

4. New Sorting Algorithm

As a consequence of this experiment, a New Sorting Algorithm was developed for nearly sorted lists. It is a novel combination of Straight Insertion Sort and Quickersort with merging. The algorithm is as follows: First the original list is scanned and pairs of unordered elements are removed and placed in another array. After a pair of unordered elements has been removed, the next pair compared are the elements immediately preceding and immediately following the pair just removed. After repeating this for all of the elements in the original list, the array of unordered pairs of elements are sorted by Straight Insertion if there are no more than 30 elements and by Quickersort otherwise. Finally, the

Table I. Comparison of Sorting Algorithms.

	Percentage out of Order										
	0	2	4	6	8	10	12	14	16	18	20
50 elements											
Straight Insertion	2.94	3.81	4.60	4.76	5.78	5.76	6.75	7.71	8.04	8.60	9.85
Quickersort	5.66	6.44	6.73	6.72	6.88	6.78	7.10	7.40	7.51	7.53	7.52
Shellsort	4.06	5.35	6.10	6.17	7.23	6.56	7.31	8.19	8.58	9.58	8.81
Merge Sort	10.50	10.54	10.55	10.54	10.55	10.60	10.57	10.59	10.63	10.62	10.64
Heapsort	19.48	19.41	19.24	19.33	19.26	19.25	19.29	19.09	19.08	18.92	18.80
100 elements											
Straight Insertion	2.97	4.88	5.58	6.65	8.71	9.16	10.08	12.14	13.20	15.18	15.48
Quickersort	6.60	7.50	7.78	8.15	8.20	8.42	8.45	8.65	8.45	8.83	8.95
Shellsort	5.03	7.52	7.94	8.61	10.00	9.82	11.02	10.97	11.21	11.92	12.10
Merge Sort	13.25	13.34	13.21	13.35	13.37	13.39	13.41	13.48	13.48	13.51	13.48
Heapsort	23.61	23.41	23.42	23.28	23.38	23.21	23.28	22.77	23.26	22.99	23.07
200 elements											
Straight Insertion	2.99	5.52	7.99	10.42	13.16	15.42	18.94	20.35	23.14	26.12	27.37
Quickersort	7.57	8.64	8.93	9.20	9.59	9.66	10.02	9.87	10.22	10.32	10.31
Shellsort	6.02	9.01	10.79	12.17	13.28	13.23	14.15	14.78	15.23	15.82	15.86
Merge Sort	14.00	14.10	14.09	14.18	14.29	14.30	14.29	14.40	14.35	14.45	14.44
Heapsort	27.82	27.61	27.56	27.59	27.48	27.30	27.37	27.27	26.94	26.91	27.15
500 elements											
Straight Insertion	2.99	9.63	14.92	21.90	29.17	34.65	41.87	49.50	54.54	59.57	65.69
Quickersort	9.44	10.44	11.07	11.06	11.16	11.58	11.80	12.01	12.23	12.05	12.13
Shellsort	7.01	12.32	14.16	15.57	16.83	17.15	17.59	18.60	19.07	19.38	19.58
Merge Sort	16.95	17.06	17.18	17.29	17.34	17.32	17.42	17.44	17.41	17.48	17.51
Heapsort	32.93	32.85	32.43	32.55	32.55	32.53	31.95	32.46	32.28	32.04	32.16
1,000 elements											
Straight Insertion	3.00	15.67	28.46	41.07	53.59	67.75	—	—	—	—	—
Quickersort	10.43	12.06	12.24	12.53	12.70	13.15	13.30	13.50	13.68	13.70	13.93
Shellsort	8.01	15.31	17.20	19.02	20.13	21.48	21.95	22.75	23.98	24.08	24.98
Merge Sort	17.04	17.94	18.08	18.19	18.22	18.36	18.37	18.45	18.47	18.50	18.52
Heapsort	37.00	36.53	36.89	36.34	36.69	35.70	36.57	35.53	36.10	36.03	35.97
2,000 elements											
Straight Insertion	3.00	29.51	55.76	79.81	109.73	131.98	—	—	—	—	—
Quickersort	11.42	13.00	13.69	13.82	14.53	14.01	14.78	14.72	14.93	15.15	15.04
Shellsort	9.00	18.58	21.77	23.60	24.54	25.24	26.60	28.19	28.65	29.02	29.43
Merge Sort	20.45	20.82	21.00	21.17	21.24	21.30	21.37	21.42	21.46	21.50	21.50
Heapsort	40.88	40.86	40.47	40.42	40.60	40.71	39.79	39.93	40.18	39.50	40.14

Table II. Rates for the New Sorting Algorithm.

	Percentage out of Order										
Number of elements	0	2	4	6	8	10	12	14	16	18	20
50	3.02	3.81	4.15	4.32	4.63	4.84	5.22	5.70	6.06	6.49	7.12
100	3.01	3.98	4.25	4.55	5.11	5.40	6.08	6.73	7.41	6.49	6.79
200	3.01	4.09	4.49	5.02	5.39	5.49	5.95	6.26	6.63	7.12	7.71
500	3.00	4.27	4.59	5.00	5.46	5.88	6.28	6.78	7.21	7.68	8.32
1,000	3.00	4.28	4.71	5.20	5.63	6.18	6.65	7.18	7.68	8.26	8.80
2,000	3.00	4.36	4.84	5.35	5.93	6.42	7.04	7.67	8.24	8.85	9.34

two sorted arrays are merged into one array.

Table II gives the test results for New Sorting Algorithms for the same set of data as used in Table I. Again, each entry is the average of the weighted sums of the number of comparisons, the moves and exchanges for ten lists of each size, and the sortedness ratio divided by the size of the list. The values in Table II show that the New Sorting Algorithm compares favorably with the Straight Insertion Sort for small and very nearly sorted lists and some values are one-half of the value for many of those for the corresponding Quickersort. The New Sorting Algorithm appears to take more advantage of the sortedness of the list than either Straight Insertion or Quickersort. Also, note that in the worst case the New Sorting Algorithm reduces to Quickersort plus the size of the list number of moves for merging. The New Sorting Algorithm also suggests that other new and faster sorting algorithms for nearly sorted lists may be developed from combinations of sorting algorithms.

5. Summary

We have compared several internal sorting algorithms that are representative of the various categories of sorting algorithms on nearly sorted lists. Our test results showed that Straight Insertion Sort is best for small or very nearly sorted lists and that Quickersort is better otherwise. A combination of Straight Insertion Sort and Quickersort with merging yielded a new sorting algorithm that performed as well as or better than either Straight Insertion or Quickersort on nearly sorted lists. Our investigation was by no means complete and suggests the following for further investigation.

(1) Here a measure of sortedness has been defined in terms of the minimum number of elements that must be removed in order for the remaining elements to be in order. Equivalently, a measure is the length of the longest increasing subsequence in the original list. Another measure of the disorder of the list and the sorting effort required would be an indication of how far out of order elements are from their proper position; e.g., in the lists (1, 2, 3, 4, 5, 7, 6) and (7, 1, 2, 3, 4, 5, 6) only one element (7) is out of order, but in the first list it is very close to its proper position and in the second list it is as far away from its proper position as possible. Intuitively, the first list would seem to require less sorting effort. We should define a measure of "how far" and compare various sorting algorithms on nearly sorted lists with different "how far" values.

(2) After defining a "how far" measure, we could compare sorting algorithms on nearly sorted lists with different combinations of the "how far" and the sortedness ratio defined in this paper.

(3) An algorithm that performed better than both the Straight Insertion and Quickersort on nearly sorted lists was given in Section 4. It is very unlikely that this is the best possible algorithm and that better sorting algorithms could be developed for nearly sorted lists.

(4) Our study compared several sorting algorithms on lists with the special property of being nearly sorted. It suggests comparing sorting algorithms on lists with other special properties such as almost completely out of order.

(5) Here the sorting effort was measured by the weighted sum of the number of comparisons, moves, and exchanges. Most prior studies just used the number of comparisons. Using only the number of comparisons as our measure of sorting effort, the same results hold for nearly sorted lists with the exception that Straight Merge Sort replaces Quickersort. Hence, Straight Insertion Sort is best for small and very nearly sorted lists and Straight Merge Sort is best otherwise; however, the New Sorting Algorithm is still almost always better than either. This suggests investigating the sensitivity of our results to other changes in the measure of the sorting effort.

Acknowledgments

We would like to thank the Oregon State University Computer Center for providing the computing time for this study and the referees for their helpful comments.

References

1. Boothroyd, J. Algorithm 201: Shellsort. *Comm. ACM 6*, 8 (Aug. 1963), 445.

2. Boothroyd, J. Algorithm 207: Stringsort. *Comm. ACM 6*, 10 (Oct. 1963), 615.

3. Elson, M. *Data Structures.* Sci. Res. Associates, Chicago, Ill., 1975. An excellent data structures textbook with a chapter on sorting.

4. Floyd, R.W. Algorithm 245: TREESORT3. *Comm. ACM 7*, 12 (Dec. 1964), 701.

5. Fredman, M.L. On computing the length of longest increasing subsequences. *Discrete Math. 11* (1975), 29–35. Describes a simple algorithm using order $n \log n$ comparisons to find the length of a longest increasing subsequence in a sequence of n distinct elements. The algorithm is also shown to be the best possible.

6. Hoare, C.A.R. Algorithm 64: Quicksort. *Comm. ACM 4*, 7 (July 1961), 321.

7. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973. The complete reference book on sorting and searching.

8. Loeser, R. Some performance tests of "quicksort" and descendants. *Comm. ACM 17*, 3 (March 1974), 143–152. Reports results of an empirical study that compared Quicksort, two of its descendants (Quickersort and qsort), Shellsort, stringsort, and Treesort.

9. MacLaren, M.D. Internal sorting by radix plus sifting. *J. ACM 13*, 3 (July 1966), 404–411. Describes a sorting algorithm that is a combination of the radix and Straight Insertion Sorts. Author chose the Straight Insertion Sort because it works well "when the records are almost in order, as they will be after the radix sort."

10. Martin, W.A. Sorting. *Comptng. Surveys 3*, 4 (Dec. 1971), 147–174. A highly recommended, readable survey of sorting algorithms.

11. Melhorn, K. Sorting presorted files. In Theoretical Computer Science 4th GI Conf., K. Weihrauch, Ed., *Lecture Notes in Computer Science*, Vol. 67, Springer-Verlag, Berlin, 1979, pp. 199–212. This article assumes a knowledge of AVL-trees. It presents a new sorting algorithm for nearly sorted lists that uses a Straight Insertion Sort for rapid insertion in an AVL-treelike data structure.

12. Scowen, R.S. Algorithm 271: Quickersort. *Comm. ACM 8*, 11 (Nov. 1965), 669–670.

13. vanEmden, M.H. Algorithm 402: qsort. *Comm. ACM 13*, 11 (Nov. 1970), 693–694.

14. Williams, J.W.J. Algorithm 232: Heapsort. *Comm. ACM 7*, 6 (June 1964), 347–348.