Information Processing Letters 26 (1987/88) 37-43
North-Holland

15 September 1987

PRACTICAL VARIATIONS OF SHELLSORT

Janet INCERPI

Institut National de Recherche en Informatique et en Automatique (INRIA), Centre de Sophia Antipolis, Avenue Emile Hugues, 06565 Valbonne Cédex, France

Robert SEDGEWICK

Department of Computer Science, Princeton University, Princeton, NJ 08540, U.S.A.

Communicated by David Gries
Received 11 March 1986
Revised 5 November 1986

Keywords: Shellsort, sorting

1. Introduction

Shellsort is a well-known sorting algorithm that uses an increment sequence $h_t, h_{t-1}, \ldots, h_1$ and works by performing insertion sort on subfiles consisting of every h_i*th* element for i from t down to 1. Sorting the elements in the h_i subfiles constitutes a 'pass', after which the file is said to be h_i-*sorted*. To guarantee that the file is sorted, a final pass using 1 as the increment completes the sort. This amounts to running insertion sort on the file. We examine variants of Shellsort that perform limited work per pass, i.e., not necessarily sorting the subfiles. One variant developed is shown to perform very well empirically and has potential as a practical sorting algorithm and as a possible sorting network.

Knuth [5] suggests a variant of Shellsort that does one comparison for each element rather than inserting it in place within its subfile. That is, during the h-sort we compare each element only to the element that is h away, exchanging the elements if necessary. Dobosiewicz [2] ran some empirical tests on what he calls a 'variant of bubblesort'; this is Knuth's variant except that Dobosiewicz finishes with bubblesort rather than insertion sort. (Dobosiewicz also compares each element with the element that is h away to the right while Knuth compares it to the element h away to the left. However, this is a cosmetic difference of the variants and does not affect overall performance.) Recall that bubblesort works by repeatedly looping through the file (from left to right), comparing adjacent elements, and exchanging as needed. Dobosiewicz's variant does not use bubblesort as the inner loop of Shellsort; rather, he uses one pass of bubblesort.

The decision to finish the algorithm with bubblesort is unfortunate. To complete a final insertion sort pass in linear time, we need only know that the number of inversions prior to this pass is linear. This is not the case for bubblesort. For example, if the smallest element is in the rightmost position of the file, then each bubblesort pass moves it just one position to the left. In this case, $N-1$ passes are needed; the time to complete the sort is $O(N^2)$, although there are only $N-1$ inversions. In general, the number of passes needed is found by taking the maximum, over all elements in the file, of the number of larger elements to the left.

Dobosiewicz compares this variant to Shellsort, using increments of the form $2^j - 1$, and quicksort. He uses $\frac{1}{2}N$ as the first increment and then

$h_i = \frac{3}{4}h_{i+1}$ for subsequent increments. The results of his empirical tests show that this variant performs favorably for small files, less than a thousand elements. We discuss these results in greater detail below.

The variation we consider is similar to those of Knuth and Dobosiewicz. Let a[1..N] represent the file of N elements stored in an array. Rather than h-sorting the file, we first make a pass, going left to right, that compares each element a[x] with a[x + h] and then make another pass, going right to left, through the file that compares a[x] with a[x − h]. Thus, while Dobosiewicz uses a bubblesort pass from left to right and Knuth uses a bubblesort pass going from right to left, we use a two-way bubblesort pass as the inner loop. (The two-way bubblesort pass is a refinement suggested in [5]. The algorithm that repeatedly makes such passes is called 'cocktail shaker sort' by Knuth.)

2. Shaker sort

We begin by looking at what happens to a file when we perform the two passes: (i) starting with the first element in the file, each element is compared to the element to its right (exchanging if necessary); (ii) starting with the last element, each element is compared to the element to its left (exchanging if necessary). Table 1 gives the starting file, as well as the file after each pass is complete. In pass (i), we see that the 11 is exchanged with elements until it is compared with the 12. The 12 moves until it runs into the 13, which is moved to the rightmost position in the file. This is precisely a bubblesort pass and the largest element is put in place. Pass (ii) goes from right to left through the file. Thus, the 7 exchanges with the 10, the 2 with the 8, 12, 4, 11, and 9. The 1 then moves past the remaining elements. This pass puts the smallest element in place.

We call pass (i) an *up-shake* and pass (ii) a *down-shake*. The up-shake puts the largest element in place while a down-shake moves the smallest element into position. Of course, if we perform another up-shake on the above file, the 12 would move into its final position. We call an up-shake and down-shake pair a *1-shake*. Thus, cocktail shaker sort repeats 1-shakes until the file is sorted. We use this naming scheme since in our Shellsort variant we talk about an h-shake, performing 1-shakes on the subfiles of elements that are h apart.

What elements are moved during a 1-shake? Define a *left-to-right maximum* of a permutation, denoted by $a_1a_2 \ldots a_N$, as any a_i that is greater than all a_j, where $j < i$. That is, in scanning the permutation from left to right, those elements that are the largest value seen when scanned are left-to-right maxima. We can similarly define the *right-to-left minimum* of a permutation. It is easy to show that the only elements whose positions change after a 1-shake are those that are either left-to-right maxima or right-to-left minima in the file (for details, see [3]).

We now describe our Shellsort variant, which we call *shaker sort*. The algorithm is based on a sequence of integer increments, $h_t, \ldots, h_1$. For the subfiles consisting of elements that are h_i apart, we perform one 1-shake; there are h_i such subfiles. After one such pass we say that the file is h_i-shaken. Notice that each pass results in every element getting compared to two elements, so in an N element file there are $2(N - h_i)$ comparisons (and at most $2(N - h_i)$ exchanges). Thus, the work done in each pass (comparisons and exchanges) is linear in the size of the file. To complete the sort we could either run insertion sort or finish with repeated 1-shakes.

We now look at a small example. The file shown in Table 2 is first 7-shaken; all the elements that are 7 apart get 1-shaken. Next, we 3-shake the file; there are three subfiles each of which is

Table 1

	5	11	3	6	1	9	12	4	13	8	2	10	7
(i)	5	3	6	1	9	11	4	12	8	2	10	7	13
(ii)	1	5	3	6	2	9	11	4	12	8	7	10	13

Table 2

	6	11	3	12	1	9	5	8	13	4	10	2	7
7-shaken	6	11	3	10	1	7	5	8	13	4	12	2	9
3-shaken	4	1	2	6	8	3	5	11	7	9	12	13	10

1-shaken. Note that the 7-shake sorts the subfiles because there are only two elements per subfile and a 1-shake puts two elements in place. In the case of the 3-shake, two of the subfiles are sorted while the third has only two elements out of place, the 6 and 5.

How sorted are the subfiles in general? This depends on the increments used. Ideally, we want to find increments for which the subfiles are almost sorted and to prove that the last pass requires linear time. If the increment sequence has O(log N) increments, then up until the insertion sort (or repeated 1-shakes) we perform a number of O(N log N) comparisons and exchanges. Therefore, we restrict our attention to such sequences.

Figs. 1, 2, and 3 give a graphic representation of the essential differences between shaker sort, Dobosiewicz's variant, and Shellsort, Each shows a series of plots of i vs. a[i], for i = 1, 2,...,256, taken before the sort and after each stage for the increments 121, 40, 13, 4, 1. The figures do not show the relative costs of the algorithms, nor are they intended to expose the best increment sequences for Dobosiewicz's variant and shaker sort.

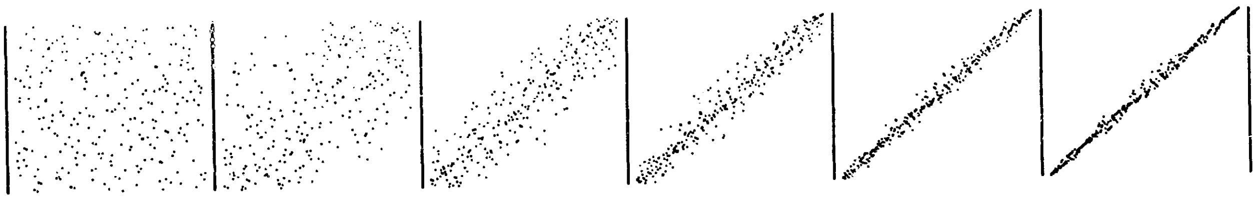

Fig. 1. Shaker sort.

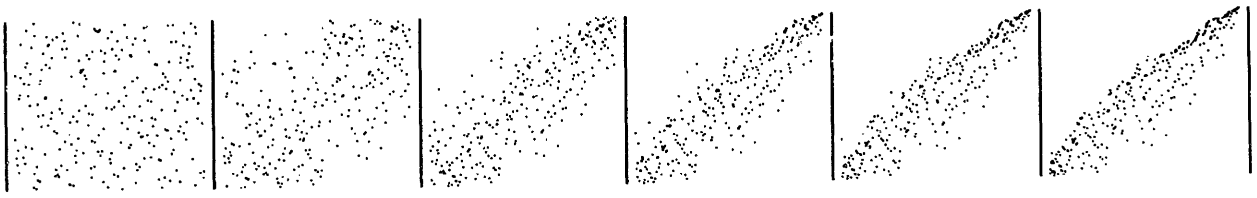

Fig. 2. Dobosiewicz's variant.

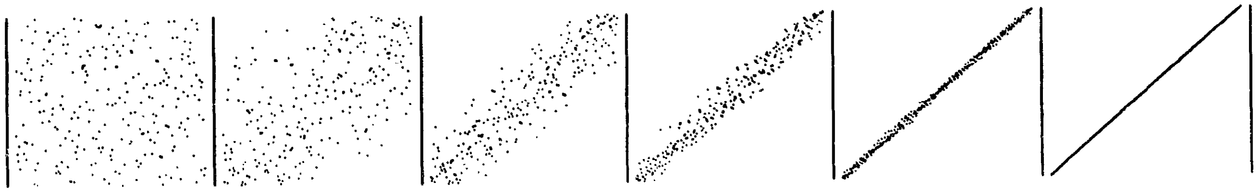

Fig. 3. Shellsort.

They do, however, strikingly illustrate two facts. First, Dobosiewicz's method exhibits some asymmetry in that some elements below the diagonal may not move as close to their final place (on the diagonal) as those above the diagonal. Completing the sort can be quite expensive, especially using bubblesort. Second, shaker sort comes quite close to producing the same degree of sortedness as Shellsort does, even though each pass is much less expensive.

To use shaker sort as an internal sorting method, we finish the algorithm with insertion sort. This guarantees that the file is sorted. To get an optimal sorting method we must show that the insertion sort pass takes O(N log N) time. That is, we must show that the file contains O(N log N) inversions prior to the insertion sort.

To use shaker sort to build a sorting netwc.k, we must finish the algorithm with repeated 1-shakes. Since each 1-shake requires 2N comparators, for an optimal network we must show that at most O(log N) 1-shakes are needed. Ajtai, Komlos and Szemeredi [1] presented an O(N log N) sorting network; however, their construction is not practical. Shaker sort, finished with repeated 1-shakes, easily translates into a sorting network since the comparisons made are completely determined by the increments used and the number of 1-shakes made at the end.

Shaker sort is of interest because it performs well in practice. It is also an attractive approach towards a structurally simple solution to the optimal sorting network problem. However, we have been unable to either bound the number of inversions remaining in the file or bound the number of 1-shakes needed to complete the sort.

3. Empirical study results

We ran extensive tests trying to get a better understanding of the algorithm. We wanted to see how different increments performed—if there were sequences that almost sorted the subfiles at each stage. We now examine the results of the empirical tests. The tests were run on a VAX 11/780 and the programs were written in the C programming language.

Initially, we ran shaker sort trying various increment sequences. We stopped the algorithm after the h-shakes and counted the number of inversions remaining. (Recall that an inversion in a permutation is a pair of indices i, j such that $a_i < a_j$ and $j < i$.) For those increments that performed well we counted the number of 1-shakes needed to complete the sort. This reveals a number of increment sequences that perform much better than others. However, it does not give us any indication why they perform so well. Thus, we ran more tests for one of the better sequences to measure how sorted the file was after each pass. That is, we looked at the number of inversions in the whole file as well as the number of inversions along the subfiles of each pass. Finally, we ran tests timing shaker sort against Shellsort, Dobosiewicz's variant, and quicksort. The results are summarized below. (For more details on the tests reported here, see [3].)

3.1. Counting inversions

We begin our search for good shaker sort increments by trying familiar sequences for Shellsort. We tested a wide range of increments, many of which are used in practice as well as some that are known to have good asymptotic bounds. Here, we report on those that perform well, along with a few standard O(log N) sequences. The sequences include a number of the form $\lceil \alpha^j \rceil$ for $\alpha < 2$, a sequence suggested by Knuth with increments $\frac{1}{2}(3^j - 1)$, and a sequence suggested by Hibbard with increments $2^j - 1$. Finally, we tested a few sequences that merge two O(log N) sequences, for example, the merging of 2^j and 3^j as well as a sequence that combines 2^j and $2^j + 2^{j-1} + 1$.

We tested files of size 100k and 1000k for k = 1, 2,..., 10. For each file size we ran shaker sort on five random permutations and counted the remaining inversions after all the passes were complete. We ran still larger files on those that did well. Table 3 contains a representative sample for files of size 10000, 20000, 40000, 80000, and 160000. The numbers in the table represent the average number of inversions rounded to the nearest integer (prior to any 1-shakes or the insertion sort pass). The $\{2^i \cup 3^j\}$ sequence and

Table 3

	10000	20000	40000	80000	160000
Knuth's	1445416	5618162	21537191	88171268	341897550
Hibbard's	127119	499477	1758469	7969322	33584639
$2^i \cup 2^j + 2^{j-1} + 1$	2611	5432	12622	90470	1072773
$2^i \cup 3^j$	2508	5053	10179	20369	41834
α^j sequences					
1.41	2486	5002	10047	20014	40128
1.5	2633	5226	10503	20938	41962
1.6	2586	5127	10217	20474	40924
1.7	2491	5015	10015	20013	40159
1.8	6703	13537	26768	54164	107560
1.9	6013	12646	25206	51209	103699

many of the α^j sequences perform very well. The number of inversions remaining seems to be about $\frac{1}{4}N$.

3.2. *Counting 1-shakes*

For those increments that had the fewest inversions remaining, we ran tests that performed repeated 1-shakes to finish the sort. This time we ran five trials of size 1000k where k = 1, 2,...,50, as well as some larger sizes. These tests were run on the α^j sequences and the merged sequences.

In Table 4 we show the average number of 1-shakes needed to sort five files of sizes 5000, 10000, 20000, 40000, 80000, and 160000. Note that most require a small number of 1-shakes to finish. This is because many of the inversions are due to elements being close to their final position prior to the 1-shakes. The merged sequence $2^i \cup 2^j + 2^{j-1} + 1$ performed well for files of size less than 40000; for larger files, the number of inversions and 1-shakes varied widely. The $\{2^i \cup 3^j\}$ sequence performed very well, although it occasionally had a file that required five or six 1-shakes to complete the sort. However, the α^j sequences were very consistent. In the case of the 1.7 sequence we ran ten trials for N = 100k (k = 1, 2,...,10), as well as for N = 1000k (k = 1, 2,...,130) and only one 1-shake was required to sort the file. The increments of this sequence are

1 2 3 5 9 15 25 42 70 119 202 343
583 991 1684 2863 4867 8273 ....

The tests above try a small number of trials for a wide variety of file sizes. Another approach is to try many trials for a fixed file size. We ran an experiment of this type with the 1.7 sequence. The algorithm successively sorted over 6000000 files of size 100 using one 1-shake. Recently, we learned from Weiss [6] that he tried similar experiments for files of size 500000. After sorting 309 files with one 1-shake he discovered a file that required

Table 4

	5000	10000	20000	40000	80000	160000
$2^i \cup 2^j + 2^{j-1} + 1$	1.0	1.2	2.0	5.0	26.0	61.8
$2^i \cup 3^j$	1.4	1.2	1.8	1.8	2.0	2.4
α^j sequences						
1.41	1.4	1.6	1.8	1.6	2.4	2.2
1.5	2.2	2.4	2.8	3.0	3.4	3.4
1.6	1.0	1.0	1.0	1.0	1.0	1.0
1.7	1.0	1.0	1.0	1.0	1.0	1.0
1.8	3.2	3.2	3.2	3.4	3.6	4.0
1.9	3.2	3.4	3.8	3.4	4.2	4.4

two 1-shakes to finish the sort. Clearly, 1.7 is not a magic number. Many of the α^j sequences finished after a small number of 1-shakes. Also, there may be other increment sequences that perform as well. As with Shellsort, the large space of possible increment sequences makes finding the 'best' one (or even one with some provable properties) a daunting challenge.

Above, we mentioned that the small number of 1-shakes reflects the fact that many elements were close to their final positions. How close to sorted are the subfiles of shaker sort when using a good increment sequence? We examined the subfiles after each pass using the 1.7 sequence. Once again the results were consistent for the different file sizes. No element was ever more than 5 places from its sorted position along its subfile. For example, after 9-shaking a file of 130000 elements, no element was more than 2 from its correct position in subfiles of 14444 elements. Also, the most unsorted such subfile had only 137 inversions. To gain a better understanding of the algorithm we need to study the interaction of passes more closely.

3.3. Shaker sort vs. other methods

We examine how shaker sort performs against Shellsort, Dobosiewicz's variant, and quicksort. We tried to optimize the algorithms equally for a fair test (using register variables and running a nonrecursive quicksort). For shaker sort we use the 1.7 sequence and end with insertion sort.

For Shellsort, we use the increments 1, 2, 5, 10, 22, 55, 110, 170, 374, 935,.... These cannot be described with a simple form such as Hibbard's sequence; the increments share large common divisors and are known to perform better in practice than standard increments (see [4]). As well, we ran Shellsort using Hibbard's increments. The quicksort we use is a nonrecursive version that does median-of-3 partitioning and uses 16 as a cutoff for small files. To finish the sort we make an insertion-sort pass. Finally, for Dobosiewicz's variant we use his increments:

$$h_t = \tfrac{1}{2}N, \qquad h_i = \tfrac{3}{4}h_{i+1}.$$

We ran tests for files of size 100k and 1000k for k = 1, 2,..., 10 as well as 5000k for k = 1, 2,..., 15. We timed the sorts (in seconds, including the generation of the permutations) as well as counted comparisons and exchanges. The latter is used to see where the algorithms spend their time. For example, shaker sort has twice as many comparisons as Shellsort because of the pass up and down in the inner loop.

Table 5 looks at the time of the algorithms in seconds for files of size 1000, 5000, 10000, 35000, and 75000.

Shaker sort and Shellsort performed about the same for files with less than 5000 elements. The ratio of the running times, Shellsort with the 'divisor' increments to shaker sort, for larger files is approximately 0.85. Shaker sort is doing many more comparisons, due to having the up and down passes as well as having more passes. To guarantee a quick insertion sort, more passes are necessary.

Shaker sort is also doing a number of unnecessary exchanges. All elements that are not left-to-right maxima or right-to-left minima are moved during the up-shake and moved back during the down-shake. There may be a way to improve shaker by avoiding unnecessary moves. That is, just move the maxima and minima. However, it is not clear that the savings are great enough to warrant a more complicated algorithm with a longer inner loop. Note that as the file becomes more nearly sorted, many more elements are maxima or minima.

In [2], Dobosiewicz compares his algorithm against quicksort and Shellsort (using Hibbard's increments). Most of the tests are on small files, although there is a test for a file of size 10000. These tests show that the variant is twice as fast as Shellsort for small files and it beats quicksort for files with less than 1000 elements (although the

Table 5

	1000	5000	10000	35000	75000
Dobosiewicz's	1.5	3.0	5.1	18.2	142.4
Shaker sort	1.5	3.0	5.2	17.2	37.9
Shellsort					
'divisors'	1.5	2.9	4.6	14.7	32.4
Hibbard's	1.5	2.9	4.9	17.0	40.6
Quicksort	1.7	2.6	3.8	10.5	20.8

ratio of the running times was 0.97). Table 5 shows that better increments make Shellsort win more consistently. Shaker sort performs similar to Dobosiewicz's variant for files of less than 10 000 elements but does better for larger files. Dobosiewicz's increments change depending on the file size, so the running time of the algorithm varies. The running time for 75 000 shows the type of hidden problems that arise from having increments dependent on the file size. (These problems can occur for smaller files as well.) Ignoring such obvious 'bad spots', after running the variant the files are not nearly as sorted as after running shaker sort. Although doing twice as much work per pass and having more passes, shaker sort still wins. The reason is the asymmetrical treatment of subfile elements we mentioned earlier (see Figs. 1–3).

Shaker sort (or Shellsort) are best for files of less than 5000 elements. Both algorithms are extremely easy to code. For larger files, quicksort is the method of choice. However, quicksort has an $O(N^2)$ worst-case performance which we know that Shellsort has not. As yet, we do not know about shaker sort, because we have been unable to find bad cases or to prove that none exist for the increment sequences of interest. Also, there may exist better increments for which both algorithms win over quicksort for still larger files.

4. Summary

We have examined a variant of Shellsort, called shaker sort, that works by comparing each subfile element with just two other elements. Empirical tests have revealed increment sequences for which the algorithm performs very well. In comparison with other methods, shaker sort, like Shellsort, is the best method for files of a few thousand elements. However, other increment sequences may lead to even better performance for larger files. We believe the results of our experiments make shaker sort a good candidate for further study as a fast internal sorting method and as a possible sorting network.

References

[1] M. Ajtai, J. Komlos and E. Szemeredi, An $O(n \log n)$ sorting network, Proc. 15th Ann. ACM Symp. on Theory of Computing, Boston, MA (1983) 1–9.

[2] W. Dobosiewicz, An efficient variation of bubble sort, Inform. Process. Lett. 11 (1) (1980) 5–6.

[3] J. Incerpi, A study of the worst-case of Shellsort, Tech. Rept. CS-85-15, Ph.D. Thesis, Dept. of Computer Science, Brown Univ., Providence, RI, August 1985.

[4] J. Incerpi and R. Sedgewick, Improved upper bounds on Shellsort, Proc. 24th Ann. Symp. on Foundations of Computer Science, Tucson, AZ (1983) 48–55.

[5] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[6] M. Weiss, Personal communication, February 1986.