AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

Theory and Practice in the Construction of a Working Sort Routine

By J. P. LINDERMAN*

(Manuscript received August 15, 1983)

Because comparison in the standard *UNIX*™ operating system sort routine, `/bin/sort`, is interpretive, it is generally more time-consuming than the standard paradigm of comparing two integers. When a colleague and I modified `sort` to improve reliability and efficiency, we found that techniques that improved performance for other sorting applications sometimes degraded the performance of `sort`. Input and output are important when comparisons are simple, but as comparisons become more complex, the number of comparisons quickly dominates the performance of `sort`.

I. INTRODUCTION

1.1 Background

In 1981, Terry Crowley and I modified the standard *UNIX*™ operating system sort routine, `/bin/sort`, hereinafter referred to as `sort`, to relax the 512-byte limit on record size and to make it more robust and efficient. The main modifications were to use more memory in the sort phase and to merge more files on each pass in the merge

* AT&T Bell Laboratories.

Table I—Performance of original and modified sort

Version	Time (Seconds)		
	Elapsed	User	System
Original	9334	2995	913
Modified	5142	3036	322

phase. We also incorporated several ideas from the scientific literature in an attempt to improve performance. Our first test results from a large sort run by the AT&T Bell Laboratories library located in Murray Hill, New Jersey, are shown in Table I.

The reduction in elapsed and system times was gratifying, but the observed increase in user time was puzzling. Although the original `sort` had to make an extra pass over the data, it had consumed less processor time. This paper explains the differences in times between the old and new `sort` routines and describes additional changes that have improved the performance of the new version.

1.2 Related research

Sorting is a well-studied area of computer science. Knuth's Volume III is both a fine introduction to sorting and a thorough analysis of techniques.[1] `Sort` uses a modified version of Quicksort, an algorithm introduced by C. A. R. Hoare.[2] Hoare suggested several optimizations on the original algorithm.[3] Most of our algorithmic changes to `sort` were inspired by Sedgewick's study of Quicksort implementations.[4] Kernighan and Plauger present a sort routine that is structurally similar to `sort`.[5,6]

1.3 Overview

After giving a brief description of `sort`, we show why comparison of two records can be computationally expensive. The general operation of `sort` is sketched.

We consider the sort phase in greater detail. After a review of Quicksort and insertion sort, the techniques of changing to insertion sort for small partitions and median-of-three selection for the Quicksort partition element are considered. The first technique reduces administrative overhead at the expense of additional comparisons, a poor trade-off in `sort`, while the second technique reduces comparisons. Artificially partitioning the records, sorting the partitions, and merging the sorted results is also shown to reduce comparisons.

The merge phase is the topic of the next section. We look at the effect of merging more files on each pass and show that the use of a heap generally makes things worse for the number of files `sort` will

be merging. A merging routine based on binary search is shown to be better than either a heap or an insertion sort. We look at the special case of long runs of records coming from a single merge input and introduce a simple adaptive technique to improve the behavior of the binary search method.

We close with performance comparisons of the original `sort` and the new version, and we mention new directions for even greater improvements.

II. COMPARISON IN SORT

I will assume that most readers are familiar with the externals of `sort` as presented in *UNIX* system user manuals. The input to `sort` is a stream of characters broken into lines by the occurrence of newline characters. Blanks and tab characters break each line into fields of characters. `Sort` can operate either on the line as a whole or on one or more of the fields in a line. Dynamic delimiting of fields distinguishes `sort` from many other sort procedures whose fields are defined by their length and their offset from the start of fixed format records. (The free format also discourages optimizations such as generating an executable comparison routine tailored to the sort arguments. We considered the portability of `sort` to be more important than its performance, so we generally avoided modifications that were machine-dependent. Bentley describes machine-dependent as well as machine-independent methods for improving sort programs.)[7] `Sort` can operate on fixed positions within fields and lines, but fixed format data are the exception rather than the rule on most *UNIX* systems.

In its simplest form, `sort` compares lines or fields left to right, byte by byte. `Sort` supports options to ignore the distinction between uppercase and lowercase letters; to ignore leading blanks; and to consider only letters, digits, and blanks. `Sort` can be instructed to perform numerical rather than lexicographical comparison, so that, for example, 5 would precede 42. Even if we ignore the complexity of comparison, simply isolating the fields to be compared can require considerable computation. For example, if the major sort field is the tenth field in each line, `sort` must skip over the first nine fields and the white space separating them. If comparison based on the major sort field results in a tie, `sort` starts over from the beginning of the line to isolate the next sort field. Comparison in `sort` therefore tends to be much more costly than the standard paradigm of comparing two integers. As we shall see, techniques that are attractive when comparison is efficient may not apply when comparison is expensive. Conversely, the techniques that improved the performance of `sort` may make other sort procedures run more slowly.

2.1 General operation of sort

`Sort` operates in two phases. In the sort phase, lines are read into main memory until no more will fit. The lines are then sorted and written to a temporary file, and the process is repeated until the input is exhausted. In the merge phase, collections of the sorted temporary files are merged together to form larger sorted temporary files. Eventually, all remaining temporary files can be merged to produce the sorted output.

Each line is read and written exactly once in the sort phase. If main memory is large enough to accommodate all the lines, then no merge phase is necessary. If a merge phase is necessary, then each line will be read and written at least one more time, as the final collection of temporary files are merged to produce the sorted output. `Sort` can merge approximately 20 files at one time, limited only by the number of files that a process may have open simultaneously. (If very long lines are being sorted, main memory could, in principle, impose an even more stringent limit. In practice, lines are not that large nor memory that small.) In the merge phase, therefore, lines may be read and written several times until the number of temporary files is adequately reduced.

To the extent that input and output dominate the time it takes to sort, a reduction in the number of merge passes is the best hope for improved times. This can be achieved by writing larger, and hence fewer, temporary files in the sort phase and by merging more files at each step in the merge phase. In the sections that follow, we will look at the sort and merge phases of `sort` in more detail and see how it was possible to increase the size of `sort` temporary files and reduce the number of merge passes. We will see how these changes can increase processor time as they reduce the time spent on input and output, and we will describe some additional changes to help reduce the processor time as well.

III. THE SORT PHASE

3.1 Introduction

In the sort phase, available memory was originally divided into two areas of fixed size. Four-fifths of memory was reserved for storing the lines to be sorted. Because lines differ in length, it is not practical to exchange two lines in memory. Instead, the remaining one-fifth of memory was dedicated to hold pointers to the stored lines, and it is the pointers to the lines, not the lines themselves, that are reordered in the sort phase. It is simpler to talk about "swapping two lines" than "swapping pointers to two lines," so we will drop the distinction.

With this fixed partitioning of main memory, memory was consid-

ered full when there were no more pointers or when there were fewer than 512 bytes left in the line storage area. If both a pointer and 512 bytes of line storage were available, `sort` would set the pointer to the start of the remaining free space and read another line into the area. (If the line was longer than the remaining line storage, pointers were overwritten with data and a core dump usually ensued. Otherwise, lines longer than 512 bytes could survive the sort phase only to be silently truncated in the merge phase.) Unless lines were quite short, less than four times the size of a pointer, the algorithm would exhaust line storage before running out of pointers, meaning that sorted temporary files were roughly four-fifths of the size of available memory.

We added an option to `sort` to allow it to allocate more memory in the sort phase. Increased work space would obviously increase the size of the temporary files. However, that would not remove the line size restriction and a purist would still be dissatisfied with running out of line space while there were unused pointers, or vice versa. We therefore eliminated the fixed partitioning of allocated memory, reading lines into the top of the work space, and assigning pointers from the other end. When lines met pointers, we sorted the complete lines, wrote the temporary file, copied the incomplete line to the start of the work space, and continued. The size of the largest line was recorded so adequate buffers could be allocated in the merge phase. Although detecting that lines had reached the pointers added time to an already expensive read routine, it virtually eliminated any limit on line length, made the best possible use of available memory, and removed a cause of core dumps from a command that should be user-proof.

3.2 Quicksort and insertion sort

The changes to memory management did not require any changes to the basic sort or merge algorithms. However, while we were changing the program, we took the opportunity to implement some proposed improvements, primarily those from Sedgewick's study of Quicksort implementations.[4] Detailed analysis of the algorithms can be found there or in Sedgewick's thesis[8] or in Knuth.[1] For convenience, we review the basic Quicksort and insertion sort algorithms. Quicksort sorts an array of lines as follows:

- If an array contains no lines or one line, do nothing. This correctly sorts arrays of size zero and one, and establishes the inductive base for the correctness of the overall method.
- If an array contains two or more lines, pick any line and compare it to all the others. Put all the lines that compare low or equal to its left, put all the lines that compare high to its right, and recursively "quicksort" the arrays to the left and to the right. This puts the

selected line where it belongs in the array and creates two strictly smaller arrays that sort correctly by induction. To be precise, Quicksort is less constrained, allowing equal lines in either the left array or the right array.

Insertion sort is also easy to understand and implement.

- If the first j lines in an array are already correctly ordered, a j + 1st line can be put where it belongs by swapping it with each previous line to which it compares low.

This procedure can be used to sort an array of N lines by invoking it to put line 2 into place relative to line 1, then invoking it to put line 3 into place relative to the first two lines, and so on until line N is put into place. This is similar to the way many people arrange a card hand, putting each new card in its correct place as it is dealt.

3.3 Small subarrays

Although Quicksort is extraordinarily elegant, the recursive approach incurs substantial bookkeeping overhead for small arrays. Hoare observed that a more efficient technique, such as an insertion sort, should be used when array size falls below some threshold, M.[3] Sedgewick took Hoare's suggestion a step further and noted that instead of doing an insertion sort on each small interval, one could leave them unsorted, and invoke a single insertion sort on the entire array of lines when all quicksorting was complete.

Sedgewick was able to improve performance by 10 to 15 percent because, in his model, comparison and exchange were simple operations, comparable in complexity to pushing an argument onto a stack. He made large reductions in administrative overhead at the cost of a small increase in comparisons and exchanges. `Sort` does not fit Sedgewick's model. The processing required to compare two lines in `sort` can easily exceed the total processing that Sedgewick measured for sorting a small array. When averaged over all permutations of N distinct elements, Quicksort never does more comparisons than insertion sort, and for four elements or more, it does fewer. Table II shows calculated values for the average number of comparisons performed when sorting small arrays.

The ill-advised implementation of this technique helps to account for the excessive user time we first observed. When I removed the insertion sort on small arrays and restored the original Quicksort algorithm, `sort` ran faster.

3.4 Median of three selection

Like many divide-and-conquer algorithms, Quicksort works best when it divides the remaining work into nearly equal pieces. If a line is chosen at random, it is unlikely that half the remaining lines will

Table II—Average number of comparisons on small arrays

N	2	3	4	5	6	7	8	9
Insertion sort	1.000	2.667	4.917	7.717	11.050	14.907	19.282	24.171
Quicksort	1.000	2.667	4.833	7.400	10.300	13.485	16.921	20.579

compare low to it and half high. We are as likely to pick the minimum or maximum line as the median.

Hoare suggested partitioning around the median of several randomly selected lines. Sedgewick, following Singleton,[9] recommended choosing the median of the first line in the array, the last line in the array, and the line in the middle of the array. This approach leads to several positive effects. (Our first rewrite of `sort` did not include this technique. Just as the suggestion we implemented was worth leaving out, the one we left out was worth implementing.) Using the median increases the probability of a favorable partitioning. Suppose we have N distinct values, 1 through N, and a particular value i in that range. By definition, i is the median of three values if one value is less than i and one value is greater than i. There are $i - 1$ values less than i and $N -$ i values greater than i, so of all choices of three values, $(i - 1)^*$ $(N - i)$ have median i. The effect therefore is to scale up the probability of selecting a value in the middle of the range and reduce the chances of selecting values near the extremes.

The Quicksort algorithm eventually compares the first and last lines in the array to the partitioning element to determine where they belong. The two or three comparisons that determine the median of three lines also establish the relative order of the three lines. As a side effect of the median selection, we therefore can move some of the work out of the main loop. Of course, the Quicksort subroutine becomes more complex. Arrays of size 3 or less become special cases, but we can terminate the recursion for these arrays where Quicksort formerly terminated for arrays of size 1 or 0. This produces some of the administrative savings that Sedgewick[4] observed from using a simpler technique on smaller arrays.

Table III shows calculated values for the average number of comparisons performed while quicksorting arrays of various sizes, with and without the median-of-three modification.

3.5 Sorting by merging

Table III shows that sorting a 4000-line array with the median-of-three feature requires an average of 47,868 comparisons. One can sort two arrays of 2000 lines each for 21,564 comparisons per array, then merge the two arrays for one more comparison per line, resulting in $2 * 21{,}564 + 4000 = 47{,}128$ comparisons, 740 fewer than the straightforward sort on 4000 lines.

Table III—Number of comparisons with and without median-of-three selection

	Number of Lines in Array							
	N		$10 * N$		$100 * N$		$1000 * N$	
N	With	Without	With	Without	With	Without	With	Without
1	0.000	0.000	22.59	24.44	573.5	647.9	9,600	10,986
2	1.000	1.000	64.42	71.11	1376.2	1563.0	21,564	24,730
3	2.667	2.667	114.76	127.69	2268.2	2582.2	34,425	39,520
4	4.667	4.833	170.73	190.84	3218.2	3669.1	47,868	54,989
5	7.067	7.400	230.92	258.92	4211.4	4806.4	61,744	70,963
6	9.733	10.300	294.49	330.94	5239.1	5983.9		
7	12.648	13.486	360.89	406.26	6295.4	7194.8		
8	15.776	16.921	429.70	484.41	7376.3	8434.5		
9	19.095	20.579	500.64	565.03	8478.6	9699.1		

Table IV—Effect of partitioning and then merging 4000 lines

Number of groups	1	2	4	8	16	32
Comparisons sorting	47,868	43,128	38,400	33,691	29,018	24,405
Comparisons merging	0	4,000	8,000	12,000	16,000	20,000
Comparisons total	47,868	47,128	46,400	45,691	45,018	44,405
Comparisons saved	0	740	1,468	2,177	2,850	3,463
Lines moved	0	0	6,000	14,000	30,000	62,000

The idea of artificially partitioning a memory load into groups, sorting the groups, and then merging the sorted results can be used for any number of groups. The merging can be done using a sorted array of the minimum lines from each of the groups. A binary search can be used to determine the proper place in this array for a new line. It takes only log N comparisons to establish the proper place for a line in such an array, but an average of $(N - 1)/2$ lines must be moved to allow the new line to be put in place. Table IV shows the savings in comparisons against the cost in lines moved for partitioning and merging a 4000-line array into a varying number of groups. (The number of lines exchanged while quicksorting also varies as $N\log N$, but it grows more slowly than the number of comparisons. Detailed analysis is quite complicated, but the number of exchanges saved by partitioning and merging is less than the number of comparisons saved. The number of exchanges saved while quicksorting does not compensate for the extra moves shown in the table.)

The optimum trade-off of comparisons against moves will depend on their relative complexity. Ideally, one might determine the complexity of comparisons dynamically, then pick a group size accordingly. In practice, the new `sort` always uses 32 groups.

IV. THE MERGE PHASE

4.1 Balancing the merge tree

In the merge phase, M sorted temporary files are merged to produce a single sorted output. If there are M or fewer files to be merged, this output becomes the output of the sort. Otherwise, the output is written to a new temporary file to be processed later. Since we replace M files with one file, the number of temporary files is constantly decreasing; thus, the merge eventually completes.

On the last merge pass, all the input records participate in the merge. If fewer than M files participate in the final merge pass, it would have been better to reduce the number of inputs to the previous merge step to leave exactly M files for the final pass, thereby saving an extra pass over some of the temporary files. But, since the size of temporary files is constantly increasing, the same argument can be made for the penultimate step. The previous step should leave it with just the right number of temporary files so that after merging M of them onto a new temporary, exactly M remain. The argument continues from step to step, leaving us with a goal of merging the right number of temporary files on the first step, when files are smallest, to ensure that all subsequent steps have exactly M inputs.

Knuth provides a formula for determining the number of files to merge at the first step.[1] The typical merge step reduces the number of temporary files by $M - 1$. The number of temporary files remaining, modulo $M - 1$, is therefore unchanging. If we can arrange to make one file remain, then the final merge step, instead of writing this one temporary file, can produce the sort output. If the merge phase begins with T files, then the first merge step merges T modulo $(M - 1)$ of them, establishing the right number of temporary files for all subsequent merge steps. If T modulo $(M - 1)$ is one, we do nothing; if it is zero, $M - 1$ files are merged.

4.2 Merge width

The number of times each byte must be read and written in the merge phase varies as the logarithm, base M, of the number of files to be merged. When `sort` was written, there was limit of ten file descriptors. The standard output and standard error descriptors were reserved. Standard input is always read first when it is among the input files, and it can be closed when it is no longer needed. This meant that seven files could always be merged onto an eighth, so M was originally seven. Most *UNIX* systems now provide 19 or 20 file descriptors, so we increased M to 16.

4.3 Use of a heap in the merge phase

To merge M sorted temporary files, the original `sort` maintained a

sorted array of M lines, one from each temporary file. The basic loop in the merge phase was to write the line in the first entry of the array, read another line from that input file, and then swap the line with adjacent entries to which it compared high. This left the array ready for another cycle. In view of our expanded number of input files, we decided that we could, to quote Kernighan:

> · · · do better with a better algorithm and data structure.
>
> One of the best is to arrange the lines as a *heap*. A heap has two desirable properties; its smallest element can be found immediately, and a new element can be put into the proper position in a heap in a time that grows only logarithmically with the heap size. You can imagine the heap as a binary tree (that is, each element has at most two descendants) in which each element is less than or equal to its children.[6]

Because each element is smaller than its children, the minimum element is at the root of the heap. The typical loop using a heap writes the line at the root and reads another line from that file. In general, this new element will not be less than both of its children, so it is necessary to sift it down in the heap and to sift up lesser elements. This is done by comparing the two children to establish the lesser and then comparing the new element to the lesser child. If the new element is low or equal, we can stop sifting and start another merge loop. If the new element is high, then we swap it with the lesser child and continue sifting the new element down in the heap.

Both the number of comparisons and number of swaps are logarithmic in the number of elements in the tree. Unfortunately, because of the comparison of the children to establish the lesser child, there are two comparisons at each of the upper levels of the tree. Figure 1 shows the number of comparisons and the number of swaps involved, depending on where the new line finally comes to rest.

Assuming that elements are equally likely to end up at any node,

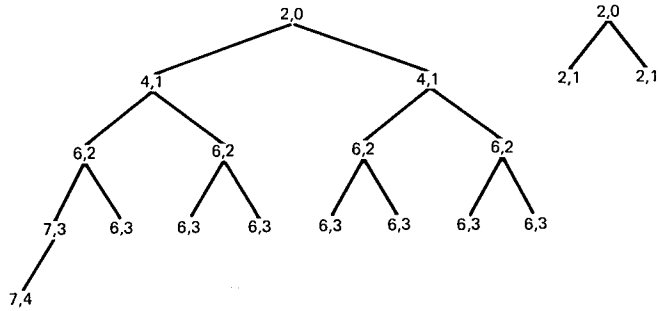

Fig. 1—Comparisons and swaps for heaps of 16 and 32 elements.

the average number of comparisons and swaps are 5.625 and 2.375 for the 16-element heap, and 2 and 0.667 for the 3-element heap. The original insertion sort would have averaged 8.438 comparisons and moved an average of 7.5 elements for a 16-way merge, but, interestingly enough, would have averaged 1.667 comparisons and one move for a 3-element array. The heap requires 20 percent more comparisons in the three-element case and also does worse for four- or five-element arrays. In short, the heap technique does not scale down nicely.

In principle, one might not worry about the scaled-down case, since it only happens at the start of the merge, when we balance the merge tree. In practice, with sort temporary files larger than half a million bytes, sorts that get into the merge phase at all probably will not have more than two or three inputs. (On many *UNIX* systems there is a limit of about a million bytes on individual files. Only users with special privileges can write larger files.) Fortunately, there is another alternative that works well for all the cases that we might encounter.

4.4 Binary insertion sort

With comparisons being our major concern, the problem with a simple insertion sort is that it averages too many comparisons when installing a new line into an array of more than a few lines. A binary search can hold the number of comparisons to the log of the number of inputs. Unlike the heap algorithm, we do not have to perform two comparisons at interior nodes of the binary tree, so the technique averages fewer comparisons for three lines or more. Having found the proper place in the array for the line, it is still necessary to move an average of half the lines to make room for the new line. Table V shows for arrays of various sizes the average number of comparisons required by a simple insertion sort, a binary insertion sort, and a heap algorithm. As the table shows, the binary insertion technique saves a significant number of comparisons over both of the other techniques.

V. SPECIAL CASES

Most of the analysis that I have included has measured performance averaged over all permutations of distinct lines. There are some special cases that deserve special emphasis.

5.1 Equal keys

It is not unreasonable to assume that it takes the same amount of time to compare any two integers, but this is certainly not valid when we consider the comparisons performed by `sort`. In particular, it is generally more expensive to detect equality than it is to detect inequality. If the sort key comprises several fields, we can stop comparing as soon as there is a difference, but we must isolate and process all

Table V—Average number of comparisons to add a new line

Number of Lines	Plain Insertion Sort	Binary Insertion Sort	Heap
2	1	1	1
3	1.66667	1.66667	2
4	2.25	2	2.5
5	2.8	2.4	3.2
6	3.33333	2.66667	3.33333
7	3.85714	2.85714	3.71429
8	4.375	3	4
9	4.88889	3.22222	4.44444
10	5.4	3.4	4.6
11	5.90909	3.54545	4.90909
12	6.41667	3.66667	5
13	6.92308	3.76923	5.23077
14	7.42857	3.85714	5.28571
15	7.93333	3.93333	5.46667
16	8.4375	4	5.625

the key fields to establish equality. Since `sort` is often used for the purpose of eliminating duplicate keys, its behavior in the presence of equal keys is worth noting.

The Quicksort algorithm in `sort` moves all the lines that compare equal to the partitioning line next to it. This is sometimes called a fat pivot. Quicksort would work correctly if the equal lines were simply left where they were found, since subsequent processing would cause them to sort where they belonged. There would be several adverse side effects, however. The partitions would be a little larger. If there were several equal keys, they would be compared again while processing the partitions. And, to eliminate duplicate keys, it would be necessary to compare adjacent keys again at output time, guaranteeing that all equal keys participated in another comparison. For these reasons, the fat pivot is worthwhile for `sort`.

5.2 Nearly ordered input

In the merge phase, suppose one of the inputs contains a series of lines that are less than the next line from any of the other inputs. This would be observed if the original input was in nearly sorted order. Using a simple insertion sort technique, a single comparison verifies that the new line is the minimum element. Using heaps, there are two comparisons, one to find the lesser child of the root, and one to verify that the root compares low or equal. Using the binary insertion technique, the number of comparisons will be the log of the number of input files.

Under these circumstances, the simple insertion sort makes fewer

comparisons than either a heap implementation or the binary insertion technique. Because I expect this behavior to be fairly common in practice, and because the overhead of doing the binary lookup is quite severe, I added an adaptive technique to the binary insertion merge algorithm. The algorithm keeps track of how often the new merge input remains in the first position. Each time this happens, the log of the number of merge inputs is added to a bonus counter. Each time it does not happen, one is added to a penalty counter. If the bonus counter exceeds the penalty counter, the new line is compared to the second line in the array. If it compares low or equal, we are done. Otherwise, we fall into the binary lookup technique, with the array shrunk by one to exclude the second line, which is now known to be less than the new line. If the penalty counter exceeds the bonus counter, we do a binary lookup on the entire array. In this way, we do only a single compare on input that demonstrates a significant amount of pre-ordering, and we do the standard binary lookup on random input.

VI. OBSERVED RESULTS

To measure the effect of the changes to `sort`, variants of the sort were timed on an AT&T 3B20 running in single-user mode. The input to all of the tests was employee data of the form

```
000876543␣0␣45138␣␣␣mh3c333␣␣␣mh  9999␣␣␣roe, richard r␣␣0␣␣␣roe, richard r␣
```

with fields delimited by the ␣ character. Counting from 0, as `sort` does, the only fields involved in the tests were

Field 0 (`000876543`) A nine-digit employee identification number.
Field 2 (`45138`) A department code of five or fewer digits.
Field 8 (`mh 9999`) A telephone extension, the first two characters of which identify an AT&T Bell Laboratories location, like "Murray Hill."
Field 16 (`roe, richard r`) The employee name, suitable for alphabetizing.

The input file was initially in order of employee surname (`roe`) with ties broken using employee identification number.

Two sets of comparison options were used on the tests. The simple option amounted to running `sort` with no arguments, so lines would be compared left to right with all bytes significant. No two employee identification numbers are the same, and all numbers have at least three leading zeros, so the sense of the comparison is determined by the fourth through ninth characters. The complex option ran

```
sort -t'ϕ' +8 -8.2 +2 -3 +16 -17
```

generating an alphabetical list of employees by department within location. The simple option therefore made comparisons about as easy as possible, and the complex option forced a fairly complicated form of comparison.

The effect of differing amounts of main-memory work space was measured by running some small tests with 32,768 bytes available, and some large tests with 500,000 bytes of available memory. The effect of differing file size was measured by running tests on the full file containing 29,157 lines and totaling 2,218,964 bytes, and a partial file of 380,609 bytes comprising the first 5000 lines of the full file. The smaller size was chosen so the entire file could fit in main memory when the large work areas were being tested. Tests were run on the old version of `/bin/sort` (with known bugs removed) and the new version I have been describing. In all cases, the final output was directed to `/dev/null`. The results for various combinations of these parameters are summarized in Table VI.

6.1 Analysis of the timings

By looking at temporary files on trial runs, I was able to determine that the original `sort` temporary files averaged around 25,435 bytes when there were 32,768 bytes of work space available, and around 399,185 bytes when there were 500,000 bytes of work space. Because of its dynamic use of work space, the new sort averaged 31,100-byte temporary files from the smaller space, and 475,000-byte files from

Table VI—Timing results

Times as Hours:Minutes:Seconds (Seconds)						Parameters			
Real		User		System		Compares	File	Memory	Sort
1:07	(67)	:44	(44)	:08	(8)	Simple	Partial	Small	Old
:42	(42)	:29	(29)	:05	(5)	Simple	Partial	Small	New
:22	(22)	:19	(19)	:02	(2)	Simple	Partial	Big	Old
:21	(21)	:17	(17)	:02	(2)	Simple	Partial	Big	New
8:24	(504)	5:28	(328)	1:02	(62)	Simple	Full	Small	Old
6:17	(377)	4:06	(246)	:44	(44)	Simple	Full	Small	New
4:20	(260)	3:17	(197)	:28	(28)	Simple	Full	Big	Old
4:06	(246)	3:00	(180)	:29	(29)	Simple	Full	Big	New
14:27	(867)	14:08	(848)	:08	(8)	Complex	Partial	Small	Old
5:11	(311)	4:59	(299)	:05	(5)	Complex	Partial	Small	New
14:52	(892)	14:49	(889)	:02	(2)	Complex	Partial	Big	Old
4:51	(291)	4:47	(287)	:02	(2)	Complex	Partial	Big	New
1:37:59	(5879)	1:35:35	(5735)	1:05	(65)	Complex	Full	Small	Old
38:09	(2289)	36:07	(2167)	:47	(47)	Complex	Full	Small	New
1:44:54	(6294)	1:43:54	(6234)	:29	(29)	Complex	Full	Big	Old
35:41	(2141)	34:39	(2079)	:29	(29)	Complex	Full	Big	New
34:41	(2081)	33:38	(2018)	:28	(28)	Complex	Full	Medium	New

Table VII—Number of sort temporary files

	Old Sort		New Sort	
	Work Space			
File	Large	Small	Large	Small
Full	6	88	5	72
Partial	1	15	1	13

the larger space. The number of sort temporary files for the various parameters is shown in Table VII.

There are no surprises in the timings when comparison is simple. The extra time required by the old sort running in a small work space reflects the extra merge passes. The improvement of the new sort over the old sort in a large work space, while modest, suggests that even when comparisons are very simple, our efforts to avoid them have been worthwhile.

The results for more complex comparisons are more dramatic. The old version of `sort` consistently runs two to three times longer than the new version. The relatively small effect of changes in working space is another manifestation of what originally surprised me when the library tested our first version of `sort`. With complex comparisons, input and output times are inconsequential. The smaller work spaces trade off input and output against comparisons in much the same way that the artificial partitioning technique trades moves for comparisons. It is for this reason that the old `sort` runs longer when it uses the large work space. The new `sort` is not immune to this phenomenon. When the work space was reduced from 500,000 bytes to 150,000 bytes to force exactly 16 temporary files from the sort phase, the last line in Table VI shows that about a minute was saved. Simply increasing the work space, one of the changes we initially thought would make the biggest improvement, may not improve performance at all, and may, in fact, make things worse.

I have no experience running the new `sort` on paged systems. In the sort phase, lines are accessed at random, so if the work space size exceeds the working set size, `sort` could suffer a page fault for every new line reference. It would be prudent to allocate a work space comfortably smaller than the expected working set size.

VII. FUTURE DIRECTIONS

The timings presented here are not comprehensive enough to justify sweeping generalizations about the performance of `sort`. Nevertheless, the following guidelines are hard to refute: (1) The complexity of comparison dominates the performance of `sort`. (2) Input and output are inconsequential by contrast.

Reducing the number of comparisons gave our most dramatic performance improvements. While it is possible to continue making improvements in this way, it will be much more fruitful to make comparison less expensive.

Profiling indicates that scanning lines for fields is a major contributor to the expense of comparison. This parsing currently takes place each time lines are compared, and it may be repeated several times if several fields participate in the comparison. It could be done once, when a line is read into main memory, if space for field pointers were associated with each line. This would reduce the effective capacity of main memory and increase the number of temporary files, but the guidelines above suggest that this is a favorable trade-off.

Another alternative is to remove most of the options from `sort` and put them in a separate key-manipulation command. The command would construct a suitable sort key for each input line, append the line to the key, pass the key and line to `sort`, and strip the keys from the output of `sort`. All the parsing of fields, mapping of upper- and lowercase, preparation of numeric fields and so on could be done, once per line, by the key manipulator, so `sort` could do simple comparisons. I wrote a simple `awk` script to add to the beginning of each line a sort key corresponding to the complex `sort` command, and another script to remove the key.

These scripts looked like

```
awk -F'\377' '{printf "%s:%s:%s:%s\n",
  substr($9,1,2),$3,$17,$0)}'
```

and

```
awk -F: '{printf("%s\n", $4)}'
```

respectively.

When I ran the scripts and the new `sort` on the full test file, they completed in about 635 seconds of elapsed time. This is less than one-third of the time it took the fastest running new `sort`, almost ten times as fast as the old. The first `awk` script consumed only two fewer seconds of user time than the `sort` (211 seconds versus 213 seconds), so a well-tuned command should do even better.

A separate key-building command has aesthetic appeal as well. Instead of further complicating a command that is already difficult to understand, `sort` could be simplified. The new command, which would also be much simpler than the current `sort`, would be more amenable to change. For example, it would be easy to add a time stamp or line counter to the sort keys so `sort` would appear to be stable, a change that would be difficult to make to `sort` itself. Options for sorting new types such as dates or times would be practical because the processing would only be done once per line. The timings give us reason to believe that we can provide greater flexibility at significantly reduced cost.

VIII. SUMMARY AND CONCLUSIONS

When we first set out to modify `/bin/sort`, we thought that performance was closely related to input and output, and we sought to reduce these by increasing the work space during the sort phase and by merging more files per pass in the merge phase. These changes reduced input/output (I/O) as expected but made it clear that comparison, not I/O, dominates the performance of `sort` when a comparison is nontrivial. Additional changes to reduce the number of comparisons dramatically improved the performance of complicated sorts and modestly improved even simple sorts.

The size limit on lines has been effectively eliminated. This is important for database applications and it paves the way for architectural changes to `sort` that trade line size for simplicity of comparison.

IX. ACKNOWLEDGMENTS

Terry Crowley made the initial changes to `/bin/sort`, and he did preliminary performance analysis while he was working with me as a summer student. Numerous readers made suggestions and comments on early drafts of this paper.

We would not have undertaken an investigation of `sort` on a system where commands were impenetrable or unportable. The *UNIX* system is responsible for providing us with both a worthwhile challenge and the measurement and development tools to carry it out.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming; Volume 3: Sorting and Searching*, Reading, MA: Addison-Wesley, 1973.
2. C. A. R. Hoare, "Partition: Algorithm 63; Quicksort: Algorithm 64; Find: Algorithm 65," CACM, *4*, No. 7 (July 1961), pp. 321–2.
3. C. A. R. Hoare, "Quicksort," Computer J., *5*, No. 1 (April 1962), pp. 10–15.
4. R. Sedgewick, "Implementing Quicksort Programs," CACM, *21*, No. 10 (October 1978), pp. 847–57.
5. B. W. Kernighan and P. J. Plauger, *Software Tools*, Reading, MA: Addison-Wesley, 1976.
6. B. W. Kernighan and P. J. Plauger, *Software Tools in Pascal*, Reading, MA: Addison-Wesley, 1981.
7. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1982.
8. R. Sedgewick, Quicksort, Stanford University Report STAN-CS-492, May 1975.
9. R. C. Singleton, "An Efficient Algorithm for Sorting With Minimal Storage: Algorithm 347," CACM, *12*, No. 3 (March 1969), pp. 185–7.

AUTHOR

John P. Linderman, S.B. (Mathematics), 1968; S.M. and Ph.D. (Computer Science), The Massachusetts Institute of Technology in 1970 and 1973, respectively; AT&T Bell Laboratories, 1973—. Mr. Linderman has participated in the design and development of several information retrieval systems, most recently, a collection of tools for implementing distributed databases. He is now in charge of computing facilities for the Computer Technology Research Laboratory. His interests are operating systems, algorithms, and software design. Member, AAAS and ACM.