Sorting n Objects with a k-Sorter

Richard Beigel*
Dept. of Computer Science
P.O. Box 2158, Yale Station
Yale University
New Haven, CT 06520-2158

John Gill
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Abstract

A k-sorter is a device that sorts k objects in unit time. We define the complexity of an algorithm that uses a k-sorter as the number of applications of the k-sorter. In this measure, the complexity of sorting n objects is between $n \log n/k \log k$ and $4n \log n/k \log k$, up to first order terms in n and k.

1 Introduction

Although there has been much work on sorting a fixed number of elements with special hardware, we have not seen any results on how to use special-purpose sorting devices when one wants to sort more data than the sorting device was designed for. Eventually, coprocessors that sort a fixed number of elements may become as common as coprocessors that perform floating point operations. Thus it becomes important to be able to write software that effectively takes advantage of a k-sorter.

*Supported in part by National Science Foundation grants CCR-8808949 and CCR-8958528 and by graduate fellowships from the NSF and the Fannie and John Hertz Foundation. This research was completed at Stanford University and the Johns Hopkins University.

2 Lower Bound

Since $\log_2 n!$ bits of information are required in order to sort n objects and since a k-sorter supplies at most $\log_2 k!$ bits per application, the cost of sorting with a k-sorter is at least

$$\frac{\log n!}{\log k!} = \frac{n \log n}{k \log k}(1 + o(1)),$$

where we write $f(n,k) = o(1)$ to denote that

$$\lim_{n,k\to\infty} f(n,k) = 0.$$

Note: in the remainder of this paper, we will implicitly ignore terms as small as $1 + O(1/n)$ or $1 + O(1/k)$. This will allow us to avoid the use of floors and ceilings in the analyses. Unless otherwise indicated, all logarithms will be to the base 2.

3 Simple Case

Using a 3-sorter, an obvious sorting method is 3-way merge sort. Since we can find the maximum of three elements in unit time, the algorithm runs in $n \log_3 n$ 3-sorter operations. Since we use the 3-sorter only to find the maximum of three elements, we are in effect discarding one bit of information per 3-sorter operation.

The cost of sorting by 3-way merge is much larger than our lower bound for $k = 3$, which is $n \log_6 n$. However, using a method suggested by R. W. Floyd [4], we can achieve $n \log_4 n$. We will perform a 4-way merge sort; if we can find the maximum of the elements at the head of the four lists with a single operation, then we will attain the desired bound.

We begin by wasting one comparison: we just compare the heads of two of the lists. This eliminates one element from being a candidate for the maximum; therefore, we need to compare only three elements to determine the maximum. Furthermore, the smallest of these three elements cannot be the maximum of the remaining elements, so we may continue in this way. As soon as one of the lists is exhausted we make back the one comparison that we wasted at the start. (Note that typically this method wastes some information per 3-sorter operation; we can expect to sort slightly faster on the average than in the worst case.)

We generalize this method in section 4.

4 Easy Upper Bound

The algorithms presented in this section are quite simple. For small values of k they are preferable to the nearly optimal algorithm to be presented later because they are easier to implement and they do not have the constant factor 4 and some lower-order terms in their running time.

Theorem 1 *We can sort n objects using*

$$\frac{n \log n}{k-1}$$

applications of a k-sorter.

Proof: We use a variation of **Tournament Sort** [5, pp. 142–43].
Algorithm:

(1) Play a knock-out tournament to determine the maximum of the n elements and store the results of all the matches as a binary tree.

(2) For $i = 1$ to n do

(3) Replace the maximum with $-\infty$ at its leaf and replay the the matches at its ancestor nodes.

Step (1) takes $O(n)$ operations. Since the winner of any match being replayed continues in the next match, $k-1$ consecutive matches involve only k elements. Thus we can determine the results of these matches with a single k-sorter operation. Therefore step (3) takes $\log n/(k-1)$ operations, and the algorithm runs in the time claimed. ∎

Tournament sort suggests a generalization of Floyd's 3-sorting method. We call this method **Tournament-Merge Sort**. We will perform a 2^{k-1}-way merge sort. We use a 2^{k-1}-player tournament in order to find the maximum element in the lists being merged; each time we delete the maximum element we replace it with the next element on the list that it was on. Since we can find the maximum of all the elements in a single operation, the cost of sorting with this algorithm is

$$\frac{n \log n}{\log 2^{k-1}} = \frac{n \log n}{k-1},$$

the same as for plain tournament sort. The advantages of this second method are the same as the advantages for merge sort: the lists to be merged can be generated in parallel at each stage, and consecutive stages can be pipelined.

5 Percentile Sort

The algorithm of the last section left a gap of a factor of $\log k$ between our lower and upper bounds. A generalization of quick sort, which we call **Percentile Sort**, will reduce that gap to a constant factor, for large k.

5.1 Selection Algorithm

We will need to know how to select the element of rank i from a list of m elements using $O(m/k)$ k-sorter operations. The following algorithm accomplishes this.

(1) Arbitrarily divide the m elements into m/k lists of k elements each.

(2) Select the median of each list (by sorting each list with a single k-sorter operation).

(3) Select the median, M, of the m/k medians by some linear-time selection algorithm, e.g., that of Blum, Floyd, Pratt, Rivest, and Tarjan [3].

(4) Divide the problem into two pieces, the elements greater than M and the elements less than M. Do this by comparing $k-1$ elements at a time to M.

(5) Invoke this algorithm recursively on the piece containing the element of rank i.

Step (1) takes no k-sorter operations. Step (2) takes m/k k-sorter operations. Step (3) takes $O(m/k)$ two-element comparisons; since we can use the k-sorter as a comparator, $O(m/k)$ k-sorter operations suffice. (In fact, the algorithm of [3] performs $\log(m/k)$ rounds of nonadaptive comparisons. Since the k-sorter can perform $k/2$ comparisons simultaneously, $O(2m/k^2+\log(m/k)) = O(m/k^2)$ k-sorter operations suffice.) Step (4) takes $m/(k-1)$ k-sorter operations. Thus steps (1), (2), (3), and (4) together take $O(m/k)$ k-sorter operations.

At least half the elements in each list are as large as that list's median. At least half of these medians are as large as M. Therefore at least one-fourth of all the elements are as large as M, and so at most three-fourths of them are less than M. Similarly, at most three-fourths of the elements are greater than M. Therefore the problem size is multiplied by a factor of at most $3/4$ after each iteration. Since the first recursive call takes $O(m/k)$ k-sorter operations, the whole algorithm takes $O(m/k)$ k-sorter operations.

We will use this selection algorithm in order to quickly find elements of ranks 1, $m/d, 2m/d, \ldots, (d-1)m/d, m$ for certain values of d. We will refer to this procedure as finding the d-tile points on the list. This takes $O(dm/k)$ k-sorter operations.

Now we are ready to present the sorting algorithm. For convenience, we define $a = b = \sqrt{k}/\log k$.

5.2 Percentile Sort Algorithm

Algorithm:

(1) Arbitrarily divide the n elements into n/k lists of k elements each.

(2) Find the a-tile points on each list. Call them the *a-points*.

(3) ¿From all the a-points select the b-tile points. Call them the *b-points*.

(4) Divide all n elements into b partitions depending on which pair of b-points the element is between.

(5) If $n > k$ then recursively sort on all partitions.

Step (1) takes no k-sorter operations. Step (2) takes one k-sorter operation per list, or n/k in all. Step (3) takes $O(nab/k^2) = O(n/k \log^2 k)$ operations. Since we can compare $k - b$ elements with b other elements with one k-sorter operation, step (4) takes $n/(k - b)$ k-sorter operations.

Therefore, it takes

$$n/k + n/k \log^2 k + n/(k - \sqrt{k}/\log k) = (2n/k)(1 + O(1/\log^2 k))$$

k-sorter operations to partition the problem. We claim that the size of each partition (from one b-point to the next b-point, inclusive) is at most $(2/b + 1/a)$ times the size of the original interval, that is, $n(2/b + 1/a)$.

Proof: We will count the number of elements between two consecutive b-points (inclusive). Consider any element that is between the two consecutive b-points. That element must lie between two a-points that are consecutive on one of the original lists (of k elements). We count two cases separately.

Case 1 (at least one of the two a-points lies between the two b-points): There are na/kb such a-points, since the b-points are the b-tile points on the list of a-points. Therefore there are at most $2na/kb$ possibilities for the pair of a-points. Because there are only k/a elements between two consecutive a-points, this case counts at most $(2na/kb)(k/a) = 2n/b$ elements between the two b-points.

Case 2 (both b-points lie between the two a-points): Since the two a-points must be consecutive on their list, there can be at most one such pair per list. Thus there are at most n/k possibilities for the pair of a-points. Because there are only k/a elements between consecutive a-points, this case counts at most $(n/k)(k/a) = n/a$ elements between the two b-points.

Combining the two cases, we see that there are at most $2n/b + n/a$ elements between the two b-points, as claimed.

Since $a = b = \sqrt{k}/\log k$, the number of iterations required is

$$\begin{aligned}
\frac{\log n}{\log \frac{1}{2/b+1/a}} &= \frac{\log n}{\log\left(\sqrt{k}/3\log k\right)} \\
&= \frac{\log n}{\frac{1}{2}\log k - \log\log k - \log 3} \\
&= \frac{2\log n}{\log k}(1 + O(\log\log k/\log k)).
\end{aligned}$$

So the total number of applications of the k-sorter is

$$\frac{2n}{k}(1+O(1/\log^2 k))\frac{2\log n}{\log k}(1+O(\log\log k/\log k)) = \frac{4n}{k}\frac{\log n}{\log k}(1+O(\log\log k/\log k)).$$

This can also be written as

$$\frac{4n}{k}\frac{\log n}{\log k}(1+o(1)).$$

6 A Faster Selection Algorithm

A slight modification to our percentile sort algorithm yields a selection algorithm that requires only $(2n/k)(1+o(1))$ applications of the k-sorter: Instead of recurring on all intervals, we recur only on the interval containing the element of rank i. As above, the first iteration takes $(2n/k)(1+O(1/\log^2 k)) = (2n/k)(1+o(1))$ k-sorter operations. Since each iteration multiplies the problem size by at most $3\log k/\sqrt{k}$, the additional work to find the sought-for element is negligible.

7 Average Time

A paper on sorting would not be complete without a discussion of an algorithm with good behavior in the average case. When using a k-sorter the average case is refreshingly easy to deal with for large k. We use a method akin to Quicksort. First we choose $k/\log k$ arbitrary elements and partition the elements into subintervals depending on how they compare to these $k/\log k$ elements. Then we proceed recursively in each subinterval.

Theorem 2 *The average cost of Quicksort using a k-sorter is $n\log n/k\log k$.*

Proof: Let $j = k/\log k$ and $m = k/(2\ln k\log k)$. We will show that it is very unlikely that partitioning an interval of length n produces any subinterval whose length is larger than $2n/m$. If such a long subinterval is produced, then one of the intervals $(tn/m, (t+1)n/m)$, where $0 \le t < m$, contains none of the j randomly selected division points. Let p be the probability of this latter event. Then

$$\begin{aligned} p &< m\frac{\binom{n-n/m}{j}}{\binom{n}{j}} < m\frac{(n-n/m)^j}{n^j} = m(1-1/m)^j \\ &< m(e^{-1/m})^j = me^{-j/m} = \frac{k}{2\ln k\log k}e^{-2\ln k} = \frac{1}{2k\ln k\log k} < 1/k\,. \end{aligned}$$

How many iterations of the partitioning operation on an interval of size n are needed before all subintervals have size less than $2n/m$? The expected number of iterations is less than

$$1 + p + p^2 + \cdots = \frac{1}{1-p} < \frac{1}{1-1/k},$$

and so the expected number of k-sorter operations required in order to partition an interval of length n into subintervals of length less than $2n/m$ is less than

$$\frac{n}{k}\left(\frac{1}{1-1/k}\right) = \frac{n}{k}(1+o(1)).$$

Therefore to partition a collection of intervals whose total length is n into subintervals whose lengths are all shorter by a factor of $m/2$ takes on average $\frac{n}{k}(1+o(1))$ k-sorter operations.

Once we reduce the interval sizes by a factor of n we are done, so the average number of k-sorter operations required in order to sort is less than

$$\begin{aligned}\frac{n}{k}(1+o(1))\log_{m/2} n &= \frac{n}{k}\frac{\log n}{\log k - 2\log\log k - \log(4\ln 2)}(1+o(1)) \\ &= \frac{n}{k}\frac{\log n}{\log k}(1+o(1)).\end{aligned}$$

∎

The above theorem yields very tight bounds on the average cost of sorting with a k-sorter. However, the upper bound is accurate only for large values of k.

8 Average Case for $k = 3$

In section 3 we presented a 3-sorter algorithm whose worst case running time is $\frac{1}{2}n\log_2 n$. It is an open question whether the constant $\frac{1}{2}$ is the best possible in the worst case. However, in this section we show that a modification of this algorithm will sort on average in $cn\log_2 n$ 3-sorter operations, for some constant $c < \frac{1}{2}$.

In section 3, we merged four lists by repeatedly finding the maximum of the heads of the lists. Because we always knew the relation between two of the list heads, we were always able to eliminate one of the heads from possibly being the maximum; thus we were able to find the maximum with a single operation that sorted the remaining three heads. Occasionally we actually knew more than just the relation between two of the heads; we knew the order of three of the heads.

Suppose, in particular, that in the course of merging four lists we find that the maximum of the heads is on the fourth list; then we know the relation between two of the first three heads. Suppose that we also find the next maximum on the fourth list; then we know the relation between another pair of the first three heads. In fact, we know the maximum of the first three heads. Suppose that the maximum of these first three heads is the head of the third list. At this point we need only compare the heads of the third and fourth lists in order to find the maximum of all four lists.

In this situation, let the algorithm always include the head of the first list in the 3-sorter operation. Suppose that the maximum is the head of the third list, and that the head of the first list is greater than the head of the fourth list. Then our next step compares the heads of the first three lists. Suppose that the maximum is the head of the third list again, and that the head of the second list is greater than the head of the first list.

Then we know the order of the heads of the first, second, and fourth lists. At this point we need only compare the heads of the second list and the third list. In this situation let the algorithm include the top two elements of the third list in the comparison. If both of them are greater than the head of the second list, then we save a comparison.

Every instance of four lists corresponds to a sequence of numbers $a_1, a_2, \ldots, a_n$, where the i-th element in the merged list is taken from list numbered a_i. We have just shown that we get lucky every time the sequence contains the subsequence $4, 4, 3, 3, 3, 3, 2, 1$, because it takes only one comparison to pull two heads from list 3.

The probability that a particular subsequence is $4, 4, 3, 3, 3, 3, 2, 1$ is

$$\binom{n-8}{n/4-2, n/4-4, n/4-1, n/4-1} \bigg/ \binom{n}{n/4, n/4, n/4, n/4} .$$

This probability is positive for $n \geq 16$ and in fact approaches 4^{-8} as $n \to \infty$. Therefore for $n \geq 16$ the probability is greater than some $\epsilon > 0$.

Thus for $n \geq 16$ the expected number of such subsequences is more than $\epsilon(n-7)$. In all but the last two merging operations, the expected number of 3-sorter operations is less than $(1-\epsilon)n+7\epsilon$. Thus the expected number of 3-sorter operations used in sorting is less than $\frac{1}{2}(1-\epsilon)n \log_2 n(1+o(1))$.

9 Open Questions

For large k, is it possible to sort with $c(n/k)\log_k n$ k-sorter operations in the worst case, for some $c < 4$? Is it possible to sort with $cn \log_2 n$ 3-sorter operations in the worst case, for some $c < 1/2$? What is the smallest value of c such that we can sort on average with $cn \log_2 n$ applications of a 3-sorter? How many applications of a k-sorter are necessary in the worst case or on average for other small values of k?

10 Conclusions

We have shown how to sort using asymptotically only $4(n/k)\log_k n$ applications of a k-sorter for large n and k, which is a factor of 4 slower than our lower bound. We have also seen how to select the element of rank i from a list of n elements using asymptotically only $2n/k$ applications of a k-sorter for large n and k. Thus it may be reasonable to design sorting coprocessors and software that take advantage of them.

Recently, Aggarwal and Vitter have independently considered the sorting problem in a more general context [1]. They have also shown that $\Theta(n \log n/k \log k)$ applications of a k-sorter are optimal for sorting n objects. More recently, Atallah, Frederickson, and Kosaraju [2] have found two other $O(n \log n/k \log k)$ algorithms for sorting n objects with a k-sorter. Neither of these papers has analyzed the constant factor involved.

Acknowledgments

We thank Rao Kosaraju and Greg Sullivan for helpful comments.

References

[1] A. Aggarwal and J. S. Vitter. The I/O complexity of sorting and related problems. In *Proceedings of the 14th ICALP*, July 1987.

[2] M. J. Atallah, G. N. Frederickson, and S. R. Kosaraju. Sorting with efficient use of special-purpose sorters. *IPL*, 27, Feb. 1988.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7:448–461, 1973.

[4] R. W. Floyd, 1985. Personal communication.

[5] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.