Sorting a Random Access File \textit{in Situ}\footnote{A preliminary version was presented at the 19th Annual Allerton Conference, 1981.}

H. W. Six and L. Wegner
Institut für Angewandte Informatik und Formale Beschreibungsverfahren, D-7500 Karlsruhe, Postfach 6380, W. Germany

To sort an external file on a random access device, the merge sort is the generally accepted method. We present a new algorithm based on Quicksort which allows sorting of external files \textit{in situ}. Analytical results and a comparison of test runs indicate that when the new algorithm is applied to a file with suitable key structure, it is competitive to the merge sort in terms of run-time behaviour.

1. INTRODUCTION

For external sorting with random access devices the merge sort method using internally presorted lists is a generally accepted standard.\footnote{If \( N \) records of data are to be sorted, additional space of order \( N \) on an external device is used in general, although from a theoretical point of view it is not a necessity, and large input/output sort areas are of advantage.} If internal sorting alone is considered, Quicksort and its variants\footnote{Even though Hoare suggested in his original paper\footnote{The use of Quicksort for external random access files, the idea seems not to have found its way into an actual implementation (we do not consider the internal Quicksort in a paging environment, as studied in Ref. 8, as a true external sort).} In his discussion of external sorting, Knuth comments [Ref. 9, p. 364]:}{\footnote{What sorting algorithm should be used? The method of merging is a fairly natural choice; other methods of internal sorting do not lend themselves so well to a disk implementation, \ldots\}} are on the average much better than the merge sort and need almost no additional space. Even though Hoare suggested in his original paper\footnote{The use of Quicksort for external random access files, the idea seems not to have found its way into an actual implementation (we do not consider the internal Quicksort in a paging environment, as studied in Ref. 8, as a true external sort).} the idea seems not to have found its way into an actual implementation (we do not consider the internal Quicksort in a paging environment, as studied in Ref. 8, as a true external sort). In his discussion of external sorting, Knuth comments [Ref. 9, p. 364]:}

In this paper we show that this view of Quicksort is deceptive, at least for files with not too large a key span. To support our claim, we present a sorting algorithm for external files based on Quicksort. The new algorithm, called EXQUISIT, needs no extra space on the external device (sorting in place), works rather well with relatively small core and is comparable to the merge sort in the average number of external read/write operations and in the movement of the disk arm. For the increasing number of mini- and microcomputers with one disc drive or two diskette drives, our method opens for the first time the perspective of sorting data sets consuming a complete disk, or stretching over two entire diskettes.

The order of presentation in the paper is as follows. Section 2 discusses the design of the algorithm and gives an example. Section 3 contains an analysis of the algorithm. Section 4 addresses the worst case and pivot selection techniques. Section 5 provides the results from letting EXQUISIT run against a merge sort program of comparable complexity.

2. DESIGN OF EXQUISIT

2.1 EXQUISIT in a nutshell

Consider the external random access file \( F \) to be sorted as a sequence of blocks each consisting of \( N \) records (in the sequel no distinction is made between records and keys). Assume that there is an internal storage area which may hold 2 blocks. Then apply Quicksort to \( F \) as follows. Read the first and the last blocks of \( F \) into core (unless the block resides already in core, see recursion). Compute a pivot element \( H \), e.g. from the keys of the first and of the last block. Then divide \( F \) into subfiles \( F_1 \) and \( F_2 \) s.t. all keys in \( F_1 \) are smaller than or equal to \( H \) and all keys in \( F_2 \) are greater than \( H \). This is done by letting a pointer \( lt \) run from 1 to \( N \) in the first (left end) block and a pointer \( rt \) from \( N \) to 1 in the last (right end) block. Whenever a pointer reaches the block border, the block is rewritten onto the external file and its adjacent neighbour block is read into the free space. Exchanges of keys occur only between blocks in core and no block is rewritten onto the external file unless its overwriting by another block is imminent.

Thus, when the pointers \( lt \) and \( rt \) meet, the rightmost block of \( F_1 \) is already in core and the recursion for \( F_1 \) may start with reading the first block of \( F_1 \). Moreover, when the recursion for \( F_2 \) starts, in most cases the first block of \( F_2 \) will again be available at no extra cost. Obviously recursion always ends with subfiles that lie entirely in one block.

As subfiles to be sorted become smaller, the shifts of locality become smaller, i.e. the distance which the arm of a disk drive has to travel to seek a block from the other end of the subfile decreases drastically. Figure 1 shows a possible history of core occupation for a file \( F \) consisting of 4 blocks.

2.2 Quicksort revisited

We assume the reader to be familiar with Quicksort.\footnote{In short this algorithm may be viewed to consist of two phases:}

Phase 1: choose at random a key \( H \) from the file \( F \) to be sorted and divide \( F \) into two subfiles \( F_1 \) and \( F_2 \) such that \( F_1 \) consists of all keys \( \leq H \) and \( F_2 \) consists of all keys \( > H \).

Phase 2: sort \( F_1 \), sort \( F_2 \) recursively; return \( F \) := sorted \( F_1 \), \( H \), sorted \( F_2 \).
SORTING A RANDOM ACCESS FILE IN SITU

-\[ F_1 \]
-\[ H_1 \]
-\[ F_{21} \]
-\[ F_2 \]
-\[ H_2 \]
-\[ F_{22} \]

| left half of | right half of | comment                |
| storage area | storage area  |                        |
| A            | B             | initial read           |
| C            | D             | \( l_t \) crossed border from A to B |
| A            | D             | \( l_t \) and \( r_t \) met in B, \( H_1 \) is pivot location, one read for \( F_1 \) |
| D            | B             | \( l_t \) crossed border from B to C |
| D            | B             | one read for \( F_{21} \) |

Figure 1. A history of core occupation for a file of 4 blocks.

The key \( H \) chosen from \( F \) is called the pivot element. Common strategies for selecting the pivot element are:

(i) taking the first (last) key of \( F \)
(ii) taking the key in the middle of \( F \)
(iii) taking the median of first, middle and last keys.

The partitioning of \( F \) is achieved by means of the previously mentioned pointers \( l_t \) and \( r_t \) scanning \( F \) from the left and right, respectively, for keys \( > H \) and keys \( \leq H \). Whenever \( l_t \) and \( r_t \) point to two such candidates, the keys are exchanged. Note that only those keys are exchanged (i.e. moved) which are out of order with respect to \( H \); if \( F \) consists of lengthy records, as is the normal case in a business environment, this fact leads to advantages over other sort algorithms, such as the merge sort, which move all records in core.

The scan of \( l_t \) and \( r_t \) comes to a halt when \( l_t \) and \( r_t \) cross within \( F \). The last exchange usually involves \( H \) whose final position is not known until the end of phase 1.

Adapting Quicksort to an environment where the data reside on external random access devices precludes some of the above techniques. Obviously, it is not desirable to exchange \( H \) at the end of phase 1, since the record belonging to key \( H \) will most likely not be in core by the time it is needed. As a result, the first design measure taken is to use a so-called fictitious pivot element \( H \) whose final placement may be thought to possibly lie on the border between two adjacent keys (the last key of \( F_1 \) and the first key of \( F_2 \)).

Secondly, only keys which are in core at the time the choice is made may be used to compute the fictitious pivot element \( H \). This excludes the middle key from \( F \) because only the left and right ends of \( F \) are on hand at the beginning of phase 1.

As a first alternative, we compute \( H \) from the first and last keys of \( F \). More refined pivot selection strategies are discussed in Section 4. Figure 2 shows our version of Quicksort but with external file handling not yet included.

Note that we have adopted Singleton’s method of stopping on keys equal to the pivot value, which prevents degeneration due to a multiset input (sorting a file with duplicate keys).

In that context, the reader should also be aware of the fact that Quicksort is an unstable sort, i.e. equal keys do not retain their relative order.

2.3 Quicksort with block handling

It is not difficult to extend the procedure \( \text{QUICKS} \) of Fig. 2 so that reading and writing of blocks is included. \( \text{EXQUIT} \) will view \( F \) as a sequence of \( M \geq 1 \) logical blocks, where a logical block is a multiple of physical blocks. Let \( N \geq 1 \) denote the size of a logical block in terms of records. A read or write request from \( \text{EXQUIT} \) will then always concern a logical block, i.e. it will result in the transfer of \( N \) records. These transfers have been localized in two procedures \( \text{READBLOCK} \) and \( \text{WRITEBLOCK} \) which will include the translation of a logical block transfer into the appropriate sequence of physical block transfers on a particular system.

As for the internal storage area, say \( S \), note that the left half of \( S \) may hold a logical block from the ‘left end’ of \( F \) or from the ‘right end’ of \( F \) (same for the right half). Figure 1 shows this situation when \( F_1 \) is about to be sorted. Thus the management of blocks uses indirect addressing which also handles the case where one and the same block in core is seen to stem from the left and from the right end of \( F \).

2.4 Selected reading and delayed writing

One of the key features of \( \text{EXQUIT} \) is a carefully designed reading and writing strategy for blocks. Remember that a block in \( F \) will not be rewritten unless the space in core must be used otherwise. Furthermore a block is rewritten only if an exchange of keys has occurred. This implies that when \( \text{EXQUIT} \) is applied to an already sorted file, no write-operations are needed.

The strategy for reading is similar. When a pointer runs out of a block, \( \text{EXQUIT} \) issues a request for the adjacent block. This is delegated to a procedure which reads the requested block (and rewrites the block whose space is needed) if and only if the requested block is not in core.

When two blocks are needed (beginning of phase 1 of the recursive sort procedure) another procedure is called which prevents mutual overwriting of needed blocks and handles the case where the two blocks are identical.

2.5 Sort left subfile first

At the start of phase 2 there is a choice of whether to sort \( F_1 \) or to sort \( F_2 \) first. Clearly, which subfile is sorted first

\[
\begin{align*}
\text{procedure } \text{QUICKS}(l, r; \text{integer}) \; & \text{;}
\text{var } l_t, r_t, H, \text{aux: integer}; \; & \\
\text{begin } & \text{;}
l_t := l - 1; \; & r_t := r + 1; \; & \\
H := (s[l] + s[r]) \text{ div } 2; \; & \{ \text{pivot selection} \} \; & \\
\text{while } l_t < r_t \text{ do } & \text{;}
\begin{align*}
\text{begin } & \text{;}
\text{repeat } l_t := l_t + 1 \text{ until } s[l_t] \geq H; \; & \\
\text{repeat } r_t := r_t - 1 \text{ until } s[r_t] \leq H; \; & \\
\text{if } l_t < r_t \text{ then } & \text{;}
\begin{align*}
\text{begin } & \text{ ;}
\text{[pointers have not crossed] } & \\
\text{aux := } s[l_t]; \; & s[l_t] := s[r_t]; \; & s[r_t] := aux \; & \\
\end{align*}
\end{align*}
\text{end; } & \\
\text{if } l < l_t - 1 \text{ then } \text{QUICKS}(l, l_t - 1); \; & \\
\text{if } r_t + 1 < r \text{ then } \text{QUICKS}(r_t + 1, r) \; & \text{;}
\end{align*}
\text{end of } \text{QUICKS} ;
\end{align*}
\]

Figure 2. Quicksort with fictitious pivot element.
has no influence on the running time. The usual strategy is to sort the shorter subtree first in order to limit the recursion stack to logarithmic size. In the case of EXQUISIT, it is preferable, not a must (see also Section 4), to adopt a left to right (preorder) strategy, provided that the pivot selection method is well enough chosen to prevent degeneration (see also Section 4). The left to right strategy guarantees that in almost all cases, at the end of the sort for $F_1$, the initial block of $F_2$ is at hand. The only exception to this preorder strategy is for subfiles $F_2$ of length less or equal to 2 blocks which are entirely in core (cf. Fig. 1, subfile $F_{22}$).

2.6 Sorting short subfiles

The recursive part of EXQUISIT is contained in a procedure QUEXS which has the same structure as procedure QUICKS of Fig. 2 except that it includes block handling. Since this block handling amounts to a certain overhead, sorting of a file $F$ switches from QUEXS to QUICKS if $F$ is contained within one block. Applying QUICKS to a file of two blocks is possible if they are in the right order, i.e. a left end block is in the left half of $S$, but it has not been implemented. Furthermore the usual improvements of QuickSort, namely to use an insertion type sort for files of length $\leq$ some constant, has not been implemented nor has the recursion been handled by use of a programmer controlled stack.

3. ANALYSIS OF EXQUISIT

The analysis of an internal sorting algorithm is usually based on the number of key comparisons. The analysis of an external sorting algorithm centers around the number of read and write operations, which generally dominates the execution time. Here, we derive an upper bound on the average number of block read operations needed to sort a file. The number of write operations is not considered because it is bounded by the number of read operations. The discussion of the movement of the disk head is deferred to Section 5.

As it turns out, the analysis is somewhat complicated because probabilities for the position of the files and subfiles with respect to block borders enter the calculation. However, we are able to prove the following: given a file $F$ which stretches over $k$ blocks and which has been split according to the fictitious pivot element $H$ into subfiles $F_1$ and $F_2$, it is possible to split $F_1$ and, after completely sorting $F_1$, to split $F_2$ with $k - 1$ block reads altogether regardless of the position of $H$ in $F$. Using this strongly twisted starting point ($F$ already split) and entering the probabilities, we can formulate a recurrence relation which yields about $2M \ln M + 3M$ as an upper bound for the average number of block reads to sort a file consisting of $M$ blocks.

For the course of this discussion, let $x$ be the size of the file $F$, i.e. the number of records (keys) in $F$ and let $L(F)$ denote the position of the first record of $F$ in the first block. As usual for this kind of analysis (see e.g. Refs 9 and 11) we shall make use of the following assumptions.

Assumptions

(i) The file $F$ is in random order and the keys to be sorted are distinct.
(ii) $L(F)$ is uniformly distributed over $\{1, \ldots, N\}$, i.e. all starting positions of $F$ are equally likely.
(iii) Let file $F$ of size $x$ be split according to a (fictitious) pivot element into two subfiles $F_1$ of size $s$ and $F_2$ of size $x - s$ ($1 \leq s \leq x - 1$). Then any value of $s$ occurs with probability $1/(x - 1)$.

The recurrence relation mentioned above averages over all starting positions of $F$, all sizes of the subfiles $F_1$ and $F_2$ after partitioning and all starting positions of the subfiles. As it turns out, the main problem is that we may not assume that, at the beginning of a partitioning phase, one of the blocks of the subfile under consideration is always in core. This anomaly occurs when the split happens to fall onto a block border on the previous level of recursion. Fortunately, to start with a file already split does the trick, as shown in the following Lemma.

Lemma

Let $F$ be a file which stretches over $k \geq 5$ blocks. Let $F$ be split into $F_1$ and $F_2$, both stretching over more than two blocks. Then for all $s \in \{1, \ldots, x - 1\}$ it takes $k - 1$ block reads to split $F_1$ and, after completely sorting $F_1$, to split $F_2$.

Proof. We distinguish two cases:

(a) the split happens to fall within block $i$ ($1 \leq i \leq k$)
(b) the split happens to fall onto a block border.

The situation is depicted in Figure 3 for $k = 6$.

```plaintext
   case (a):
    \[ F_1 \quad F_2 \]

    \[
    \begin{array}{c|c|c|c}
    & 1 & i & i+1 & k \\
    \end{array}
    \]

   case (b):
    \[ F_1 \quad F_2 \]
```

After the split of $F$, block $i$ is in core. Therefore in both cases it always takes $i - 1$ block reads to split $F_1$. According to the preorder strategy, block $i$ is in core again when $F_2$ is about to be sorted. Hence we have $k - i$ block reads to split $F_2$ in both cases (a) and (b), and therefore it takes $k - 1$ load operations to split $F_1$ and $F_2$.

If either $F_1$ or $F_2$ is shorter than 3 blocks, splitting $F_1$ and $F_2$ may require less than $k - 1$ read operations. Thus $k - 1$ remains as upper bound for the partitioning phase up to the point, where the file has size $N + 1$. Clearly $F$ now stretches over at most two blocks and no further block reads are needed to sort $F$ (boundary condition).

Before entering the above result into the recurrence relation, care has to be taken in showing that the assumptions (i)-(iii) above hold again for the subfiles $F_1$ and $F_2$ after partitioning.
Lemma

Let $k_{LF}$ denote the number of blocks occupied by $F$ with starting position $LF$. The recurrence relation

$$g(x) = \frac{1}{N} \sum_{1 \leq LF \leq N} (k_{LF} - 1)$$

$$+ \frac{1}{x - 1} \sum_{1 \leq s \leq x - 1} (g(s) + g(x - s)) \quad \text{for } x > N + 1$$

and $g(x) = 0$ for $x \leq N + 1$ gives an upper bound for the average number of block reads to sort $F$ of size $x$ when $F$ has already been split.

Proof. Follows from the preceding discussion.

The average number of blocks occupied by $F$ is given by the term

$$\frac{1}{N} \sum_{1 \leq LF \leq N} k_{LF}$$

whose evaluation leads to $[(x - 1)/N] + 1$.

We are now in a position to give an upper bound on the expected number of block reads $EBR(x)$ to sort a file $F$ of size $x$ under the assumptions (i)-(iii). $EBR(x)$ is computed from the upper bound $g(x)$ above and the initial effort to split the original file $F$.

Theorem

$$EBR(x) \leq 2 \frac{x}{N} \ln \frac{x}{N + 1} + \frac{3x}{N} + 1 - \frac{1}{N(N + 1)}$$

Proof. Since

$$\frac{1}{N} \sum_{1 \leq LF \leq N} k_{LF} = \frac{x - 1}{N} + 1$$

we arrive at

$$g(x) = \frac{x - 1}{N} + \frac{1}{x - 1} \sum_{1 \leq s \leq x - 1} (g(s) + g(x - s))$$

for $x > N + 1$

and $g(x) = 0$ for $x \leq N + 1$.

The recurrence relation for $g(x)$ can be solved using the familiar techniques of ‘eliminating the sum’, which yields

$$g(x) = 2 \frac{x}{N} - \frac{(x - 1)x}{(x - 1)N} + \frac{x}{x - 1} g(x - 1)$$

and ‘telescoping’ which brings us to

$$g(x) = 2 \frac{x}{N} \left[ \sum_{i=N+2}^{x} \frac{1}{i} - \sum_{i=N+2}^{x-1} \frac{1}{i+1} \right]$$

$$\leq 2 \frac{x}{N} \ln \frac{x}{N + 1} + 1 + \frac{1}{2x} - \frac{1}{2(N+1)}$$

Adding the effort for the initial split of $F$ we have

$$EBR(x) \leq 2 \frac{x}{N} \ln \frac{x}{N + 1} + 1 + \frac{1}{2x} - \frac{1}{2(N+1)}$$

$$+ \frac{x - 1}{N} + 1$$

$$= 2 \frac{x}{N} \ln \frac{x}{N + 1} + \frac{3x}{N} + 1 - \frac{x}{N(n + 1)}$$

For all practical purposes, $M = x/N$ may be taken as the number of blocks which $F$ occupies, and thus we obtain $2M \ln M + 3M$ as an upper bound for the average number of read operations to sort $F$.

4. WORST CASE BEHAVIOUR AND IMPROVED PIVOT SELECTION STRATEGIES

Since a file consisting of $M$ blocks can be sorted with $O(\log M)$ block reads on the average, one might expect that in the worst case $O(M^2)$ block reads are necessary. That is not true, however, because a file of $M$ blocks has a size $x \approx MN$ and when each partition phase splits off one element only, a worst case of $M^2N$ results. Hence, a safe strategy for the choice of the pivot element is even more important for the external than for the internal Quicksort. Furthermore, an external sort must also take composite and non-numerical keys into account.

We address these issues by discussing four pivot selection strategies of increasing complexity and probe with a word of warning.

If the keys are numerical and uniformly distributed, the mean of first and last (see Fig. 2), is a fast and fairly safe method which also works well for multiset input and (almost) presorted files.

As a second alternative, the previous strategy can be upgraded to a mean-of-$k$ pivot selection method, similar to Motzkin’s Meansort, where $k$ is the number of blocks of a subfile to be split. When splitting a subfile, one key from each block is rewritten and taken and added to a variable leftsum, respectively rightsum. After the split is complete, the mean of the terms in leftsum and rightsum gives the pivot values on the left and right subfiles. The method requires no extra block reads and needs only two local variables in the recursive sort procedure.

The third alternative for numerical keys is to compute the fictitious pivot element from the minimal and maximal keys in the (sub)file to be sorted. This sort procedure is sometimes called binary sort. Like the second alternative it has the advantage that additional block read operations are not required, and that partially split off the min and max values of the subfiles $F_1$ and $F_2$ can be done ‘on the side’ with at most two additional key comparisons for every key $F$ during the partitioning phase for $F$. Since for the initial split of $F$ these values are not on hand, the first pivot element must be chosen from the keys of the first and last blocks. In order to force EXQUISIT with the $(\min(F) + \max(F))/2$ pivot selection into an $O(M^2-N)$-type degeneration, the values of the keys have to increase exponentially, which quickly finds its limit in the key size. In practical terms—large file and large span of key values—the new upper bound for the worst case, easily seen to be $O(M \log_2 (\max(F)))$, is of a completely different nature from the previous upper bound. We show this with the following example.

Let $F$ be a file of 50 000 records, the block size $N$ be 50, and the keys 9 digit-numbers. Then $F$ consists of $M = 1000$ blocks and thus $M \log_2 M \approx 10000$, $M^2N = 500000000!$, but $M \log_2 (\max(F)) \leq 30000$.

Unlike Quicksort, the analysis of the average behaviour of the binary sort must take into account the distribution of key values. With the added difficulty of
block handling such an analysis is outside the scope of this paper.

The last alternative is a median-of-k strategy which is best suited for non-numerical keys. During the partitioning of a subfile, a total of k key values (say k ≤ 20) are taken at even intervals from the blocks which are rewritten onto the external device. After the split position has been established, the median of the k' sampled values belonging to the left subfile and the median of the k" values of the right subfile are used as pivot element for the left and right subfile, respectively.

As a result of the preceding discussion, it is only fair to issue a word of warning. Quicksort, and consequently EXQUISIT, are probabilistic algorithms, i.e. 'unstable' methods in the non-technical sense. Extreme input data may cause various kinds of problems: degeneration with an explosion of run-time, or a stack overflow, inability to perform arithmetic with the keys, etc. It therefore seems appropriate to limit the applicability of the new sort method to files with suitable key spaces and to avoid some of the optimizations—such as sorting the left subfile first instead of the shorter one—in order to limit the recursion stack. Similarly the case of a system crash has to be catered for, e.g. by copying the file onto some tertiary storage medium before the sort, since the in situ property would leave the external file in an inconsistent state.

5. EXPERIMENTAL RESULTS

In 1970, Brawn, Gustavson and Mankin examined the behaviour of Quicksort and of the merge sort in a FIFO-paging environment. Their results indicate that for all considered core sizes Quicksort is inferior to the merge sort in a paging environment. Since it seems that in Ref. 8 both source code and data are paged, we are unable to relate their results to ours.

We therefore decided to directly compare a straightforward version of EXQUISIT with a comparable version of a merge sort in a simulated environment. As we used the integers 1, 2, ..., MN in a random permutation as input data, the results are only representative for uniform distributions of a numerical key space; other distributions and orderings are less favourable for EXQUISIT and may reverse the outcome. However, under the just mentioned conditions, the Quicksort approach with I/O-operations controlled by the sort program is competitive to the merge sort in the number of external read/write operations and, of course, has the advantage of being in situ.

EXQUISIT is coded in Burroughs Extended Algol 60. The parameters of EXQUISIT are M (number of blocks) and N (number of records per block). The records to be sorted consist of keys only, which are pairwise distinct integers. The file is represented by an integer array E[1:M, 1:N] where each of the M rows stands for a (logical) block of the file. The internal sort area is declared as an integer array S[1:2N]. As described in Section 2, data are transferred in terms of logical blocks of size N.

EXQUISIT's competitor is a p-way merge sort which starts forming strings of length 2N using Quicksort. In order to minimize the number of external accesses, the merge order p is chosen as follows: let x be the file size and k be the core size (in records) and assume that the initial phase of the merge sort produces r ≤ [x/k] presorted strings. Using a p-way merge, the sort is completed in [log_p(r)] passes where each pass requires external reads (and writes) because the core is partitioned into p input buffers and one output buffer which are of equal size. Hence, the optimal merge order p_opt is computed as the value of p which minimizes the number of external reads given by

$$\left[ \frac{x}{k} \right] \left( \frac{1}{p + 1} \right)$$

The program is also written in Burroughs Algol and the internal sort area is declared as an integer array S[1:2N]. In our test runs, N is always chosen in such a way that 2N is a multiple of p_opt + 1, i.e. 2N = (p_opt + 1)N' for some N'. For the sake of simplicity, the size of a logical block in the merge sort environment is assumed to be N'. Hence, the file to be sorted consists of M' blocks of size N' s.t. M' N' = MN, where M and N are the corresponding parameters in EXQUISIT. The file and the external working space are represented by an integer array E[1:2M', 1:N'] and data are transferred in terms of logical blocks of size N'.

Three counts are implemented in both sort programs:

(i) no. of block reads (NBR)
(ii) no. of block writes (NBW)
(iii) distance of read/write head travelled (DHT).

They must be interpreted as follows: each time a block of N records, respectively N' records, is moved from E to S, or from E' to S, NBR is increased by one. Similarly NBW is increased by one for each transfer from S to E, or from S to E'. After reading or writing block i, the head of the disk was assumed to rest on the border between block i and block i + 1. Then the cost of letting the head travel to block j for the next read/write is computed as abs(j - (i + 1))N, respectively abs(j - (i + 1))N'. Thus DHT gives the total number of records which the arm has passed over seeking next blocks during the sort.

It is important to remember that the keys of the file to be sorted are the integers 1, 2, ..., MN in a random permutation. Obviously, this choice of data yields for EXQUISIT an optimal split if the '(min(F) + max(F))/2' strategy' for the selection of the pivot element is used. We therefore present two tables of results for EXQUISIT: Table 1 gives the figures when the '(min(F) + max(F))/2' strategy' is used, Table 2 those obtained by a 'first key + last key)/2' strategy. Table 3 shows the results for the merge sort.

<table>
<thead>
<tr>
<th>Table 1. EXQUISIT with '(min(F) + max(F))/2 strategy'</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

274 THE COMPUTER JOURNAL, VOL. 27, NO. 3, 1984
Table 2. EXQUISIT with ‘(first + last key)/2’ strategy

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>core</th>
<th>NBR</th>
<th>NBW</th>
<th>DHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>6</td>
<td>12</td>
<td>4262</td>
<td>3726</td>
<td>786960</td>
</tr>
<tr>
<td>200</td>
<td>12</td>
<td>24</td>
<td>1953</td>
<td>1832</td>
<td>440352</td>
</tr>
<tr>
<td>100</td>
<td>24</td>
<td>48</td>
<td>868</td>
<td>845</td>
<td>257760</td>
</tr>
<tr>
<td>40</td>
<td>60</td>
<td>120</td>
<td>300</td>
<td>298</td>
<td>132120</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>240</td>
<td>123</td>
<td>123</td>
<td>78600</td>
</tr>
<tr>
<td>10</td>
<td>240</td>
<td>480</td>
<td>48</td>
<td>48</td>
<td>46560</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>1200</td>
<td>12</td>
<td>12</td>
<td>19800</td>
</tr>
</tbody>
</table>

The actual figures should be interpreted as follows. The file to be sorted consists of 2400 records. This number was chosen because it divides easily for a number of core sizes and is large enough to show the efficiency of the programs.

The core size varies from 12 records to 1200 records. Thus between 0.5% and 50% of the file can be stored in core. Accordingly M and N for EXQUISIT, respectively M’ and N’ for the merge sort, are varied s.t.2N, respectively (popt + 1)N’ equals the core size. As a consequence, the NBR-column gives the number of block reads for blocks of size N, respectively of size N’. Thus EXQUISIT transfers data in larger portions—except during the presorting phase of the merge sort where each filling of the core was counted as one block read—and the slightly smaller number of seek operations might in practice be balanced by larger transfer times. For the core sizes 48 and 240 in Table 3 the counts NBR/NBW and DHT include copying the file back, since after the last pass the file did not come to rest on its original place.

The reader might be surprised by the extreme differences in the DHT values for EXQUISIT and the merge sort within our model. They can be explained through the following moderate analysis. For EXQUISIT it can be shown that the number of blocks which the disk head has to pass over in splitting a (sub)file of M blocks is at most (M + 1)(M + 2)/2 − 2. Neglecting a split on a block border, at most

\[ f(M) = \frac{(M + 1)(M + 2)}{2} - 2 + \frac{1}{M} \sum_{1 \leq i \leq M} f(s) \]

blocks are passed over, on the average, when sorting a file of M blocks. This evaluates to

\[ f(M) = 3M^2/2 + 6(M - 1)H_{M-2} + M/2 - 3 \]

which is of order \( M^2 + M \ln M \).

For the merge sort, when an interleaving technique is not used, we can show, however, that the distance which the disk head has to travel is at least of order \( M^2 \log_{p_{opt}}(M) \), i.e. it is at least by a factor of \( \log_{p_{opt}}(M) \) larger. This is derived from the fact that in the merge sort the disk head has to travel \( 2M/p_{opt} \) times a distance of \( M \) blocks in each pass using the optimal assumption that \( p_{opt} \) successive write operations are always followed by \( p_{opt} \) successive read operations and vice versa.

Thus, in each pass the total distance is \( 2M(p_{opt}) \) blocks which, when multiplied by the number of passes, gives the claimed result. In concluding the discussion of the experimental results, we also mention that in the simulation the execution time of EXQUISIT was, on the average, 25% lower than the corresponding time of the merge sort, whereas a recent implementation of the new sort for actual disk files on a PDP 11/04 and its comparison with a sophisticated merge sort14 gave the merge sort the lead by 5 to 20%. The interpretation of the new results and the development of a stable block chained EXQUISIT version using 3 instead of 2 internal blocks should provide interesting topics for further research.

6. CONCLUSION

For external general purpose sorting, the merge sort will in all likelihood remain the prime candidate. However, for particular applications, where the needed extra space on the external device is not available, the new external Quicksort algorithm introduced and analysed in detail in this paper might provide an interesting alternative. If applied with the necessary precautions against degeneration and if applied to suitable input, the analysis and the experimental results indicate that the new sort is at least competitive to the merge sort.

Acknowledgements

We thank W. Weber and J. Karszt for initiating this research by asking the right question at the right time, and Miss S. Reiniger for her help during the writing and testing of the programs. We are also indebted to a referee for directing our attention to some of the dangers of Quicksort.

REFERENCES


Received July 1983

THE COMPUTER JOURNAL, VOL. 27, NO. 3, 1984