THE EFFECTS OF PAGING ON SORTING ALGORITHMS

by

J. G. Peters

Research Report CS-74-14

Department of Applied Analysis and
Computer Science

University of Waterloo
Waterloo, Ontario, Canada

August, 1974

THE EFFECTS OF PAGING ON SORTING ALGORITHMS

J. G. Peters

Department of Applied Analysis and Computer Science

University of Waterloo

August 1974

Abstract - Several sorting algorithms were run on a machine which uses paging to manage storage. The paging activity which occurred as a result of the algorithms being run was observed as the ratio of virtual storage to real storage was varied.

It was found that partition sorts such as QUICKSORT resulted in the smallest number of page faults being issued while insertion sorts such as BINARYINSERTION caused the most.

Acknowledgement

I would like to thank Pieter Kritzinger for originally suggesting to me the topic of this report and for many of his ideas which I have incorporated into it.

I would also like to thank Romney White for a great deal of help with all aspects of this report.

This research was supported in part by N. R. C. Grant number A8975.

J. G. Peters

August 1974.

INDEX

PAGE

1. INTRODUCTION

The data presented in this report is a result of performance tests of fourteen sorting algorithms run on a machine which uses paging to manage main storage.

The sorting algorithms chosen for study represent the major internal key comparison sorts in common use or variations of them. The algorithms were written for non-paging machines. No attempt was made to modify the algorithms to run more efficiently on a paging machine, as the purpose of this study was to discover which characteristics of the algorithms degraded their performance in a paging environment.

This report is preliminary to a more detailed study which is currently being conducted.

The programmes were run under CMS of VM/370 on an IBM 370 Model 158 at IBM Canada's Toronto DPCE Customer Education Centre.

2. VM/370 FEATURES

2.0 General

The IBM Virtual Machine Facility/370 (VM/370) is a system which provides virtual machine support for multiple users on an IBM System/370 computer. It consists of several parts.

The Control Program (CP) controls the resources of the real machine and simulates simultaneous operation of several virtual machines, each of which is being controlled by its own operating system.

The Conversational Monitor System (CMS) is an individual user operating system which runs in a virtual machine under CP and can be controlled by the user from an interactive remote terminal.

The service programmes which are used by CP perform such functions as remote spooling operations, printing real machine storage dumps, handling disk to tape, tape to disk, and disk to disk file transfers, and formatting direct access volumes.

A detailed description of VM/370 is available in the references included in the bibliography and only the paging algorithm will be discussed here.

2.1 Page Replacement Algorithm

In order to be able to map addressable virtual storage onto available real main storage, virtual storage is divided into pages of size 4K bytes while real main storage is divided into page frames of the same size. Normally a large address-space is available to each virtual machine so that, as a result, there will be more pages in the virtual machines than real page frames. A page manager is needed to put pages referenced by a virtual machine into page frames, at the same time removing pages as necessary.

The pages in the dynamic paging area (the pageable pages) are on three linked lists. The USERLIST contains entries for all nonlocked page frames currently allocated to virtual machines in the in-queue group (the group of virtual machines that are currently competing for use of the real CPU). When a virtual machine is dropped from the in-queue group, the page frames it has been allocated are placed at the end of the FLUSHLIST. The FREELIST contains entries for page frames not currently assigned to a virtual machine and is maintained in first-in, first-out order.

The page manager attempts to keep frequently used virtual pages in real storage and when pages are removed, the least frequently used pages are removed first. The current status of a page frame may be temporarily locked for an I/O operation, permanently locked, in the FLUSHLIST, in the

FREELIST, shared, reserved, allocated to CP, or unavailable for allocation.

When a request for real storage is made, page frames from the FREELIST are allocated. If this is not possible, page frames from the FLUSHLIST are used. If the FLUSHLIST is also empty, the most inactive page frame in the USERLIST is used.

If a page frame can be allocated from the FREELIST, a page write can be avoided, so a threshold minimum of page frames are kept in the FREELIST. Whenever a page is reclaimed from the FREELIST or the number of page frames in the list falls below the threshold, a page selection routine is called to replenish the list. The FLUSHLIST is first inspected and the change bits (which indicate whether the virtual page currently in the page frame has been modified) for the first page frame are inspected. The page is written out if it has been changed and then is added to the end of the FREELIST.

If more page frames are still required for the FREELIST to contain the threshold number, the USERLIST is inspected, starting at the beginning of the list, to find a page frame that hasn't been referenced since the last inspection. Each page frame inspected that has been referenced is moved to the end of the list and its reference bits (which indicate whether it has been referenced since last inspection) are turned off. If an unreferenced page frame isn't found, the

first page inspected is selected because its reference bits are now turned off. A page write is performed if the change bits are on and the page frame is added to the end of the FREELIST.

The FREELIST is always searched before selecting a page frame for allocation. If the desired virtual page is in a page frame in the FREELIST, the page can be reclaimed and a page read avoided. If reclamation is impossible and the page is not currently being paged in as a result of a previous request, a page frame must be selected for allocation.

The page manager routines also maintain statistics for each virtual machine in a table called the VMBLOK. It is from this table that the figures for the number of page reads and page writes were obtained.

3. MACHINE ENVIRONMENT

3.0 Hardware

The machine used for these tests was an IBM 370 Model 158 with two megabytes of semiconductor main storage which included 8K of cache memory.

An IBM 3330 Direct Access Storage Device was used for paging operations. The maximum paging rate achieved was approximately 66 page reads and writes per second with the average being between 55 and 60 per second. 3330 disk packs were also used for real secondary storage.

The programmes were run interactively using an IBM 3270 display terminal as the virtual operator's console.

3.1 Real Machine Environment

The tests used to obtain the data in this report were run stand-alone. This was necessary for two reasons.

Firstly, since the purpose of the tests was to study paging activity, the test programmes were designed to cause a large number of page faults. This would degrade the performance of a shared machine to a point that would be unacceptable to other users of the system.

Secondly, although a virtual machine can be guaranteed a minimum working set size (the number of page frames al-

located to the machine) , there is at present no facility in VM/370 for guaranteeing a maximum working set size. In order to limit the tests to files of realistic size while keeping execution time low it was necessary to limit the number of real pages available. This had to be accomplished by locking all pages not used by the test programmes into another user's virtual machine or to the system or operator. Clearly, every time another user logged on or logged off of a shared system, the number of available pages would change.

Also, each of the three sets of test runs took at least five hours real time and would take substantially longer on a shared system.

3.2 Virtual Machine Environment

The virtual machine was defined as four megabytes for most of the tests to guarantee that enough virtual storage was available for all the programmes to run. This was necessary as the array being sorted occupied two megabytes of virtual storage in most cases.

Version 2.4 of CMS was used to maintain the virtual machine environment for all tests.

4. TEST PROGRAMMES

4.0 General

This section contains listings of the fourteen sorting algorithms tested plus a utility routine called ORDER. ORDER uses the list of pointers produced by LISTIN and LISTNATURALMERGE to reorder the array of elements being sorted.

The same mainline programme was used to drive all fourteen routines. The functions of the mainline programme were initialization of variables, generation of the random data (discussed in Section 5), and acquisition and output of the statistics presented in Sections 7, 8, and 9.

The programmes were compiled using an IBM ALGOL F compiler with Princeton modifications which was adapted by the author for use on VM/370 for the purpose of these tests. The code produced was not self-modifying. Most input and output in the mainline programme was performed with FORTRAN subroutines through an ALGOL-FORTRAN interface.

For a description of the sorting algorithms, the reader is referred to Lecture Notes on Sorting by P. S. Kritzinger.

4.1 Samplesort

```
'PROCEDURE' SAMPLESORT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'PROCEDURE' QUICKSORT(A,L1,U1,D);
   'VALUE' L1,U1,D; 'INTEGER' L1,U1,D; 'ARRAY' A;
   'BEGIN'
      'INTEGER' P,Q,IX,IZ,I,J,M,L,U;
      'REAL' X,XX,Y,Z,ZZ;
      'INTEGER' 'ARRAY' LT,UT(/1:LN((U1-L1+1)/D)/LN(2)+0.5/);
      M:=0; L:=L1; U:=U1;
PART:
      P:=L; MOVE:=MOVE+2; Q:=U; X:=A(/P,1/);
      Z:=A(/Q,1/); I:=0; J:=Q-P-D; COMP:=COMP+1;
      'IF' X>Z 'THEN'
      'BEGIN'
         Y:=X; MOVE:=MOVE+2; A(/P,1/):=X:=Z; A(/Q,1/):=Z:=Y;
         INTER:=INTER+1;
      'END';
      'IF' U-L>D 'THEN'
      'BEGIN'
         XX:=X; MOVE:=MOVE+2; IX:=P; ZZ:=Z; IZ:=Q;
LEFT:
         'FOR' P:=P+D 'WHILE' P<Q 'DO'
         'BEGIN'
            X:=A(/P,1/); MOVE:=MOVE+1; COMP:=COMP+1;
            'IF' X>=XX 'THEN' 'GO TO' RIGHT;
         'END' P;
         P:=Q-D; 'GO TO' OUT;
RIGHT:
         'FOR' Q:=Q-D 'WHILE' Q>P 'DO'
         'BEGIN'
            Z:=A(/Q,1/); MOVE:=MOVE+1; COMP:=COMP+1;
            'IF' Z<=ZZ 'THEN' 'GO TO' DIST;
         'END' Q;
         Q:=P; MOVE:=MOVE+2; P:=P-D; Z:=X; X:=A(/P,1/);
         'GO TO' COMPARE;
DIST:
         COMP:=COMP+1;
         'IF' X>Z 'THEN'
         'BEGIN'
            Y:=X; MOVE:=MOVE+2; A(/P,1/):=X:=Z; A(/Q,1/):=Z:=Y;
            INTER:=INTER+1;
COMPARE:
            COMP:=COMP+2;
            'IF' X>XX 'THEN'
            'BEGIN'
               XX:=X; MOVE:=MOVE+1; I:=I+D; IX:=P;
            'END';
            'IF' Z<ZZ 'THEN'
            'BEGIN'
               ZZ:=Z; MOVE:=MOVE+1; I:=I+D; IZ:=Q;
            'END';
         'END'
         'ELSE'
         'BEGIN'
```

```
            XX:=X; MOVE:=MOVE+2; IX:=P; ZZ:=Z; IZ:=Q; I:=I+D+D;
        'END';
        'GO TO' LEFT;
OUT:
        'IF' P 'NOTEQUAL' IX 'THEN'
        'BEGIN'
          A(/P,1/):=XX; A(/IX,1/):=X; MOVE:=MOVE+2;
        'END';
        'IF' Q 'NOTEQUAL' IZ 'THEN'
        'BEGIN'
          A(/Q,1/):=ZZ; A(/IZ,1/):=Z; MOVE:=MOVE+2;
        'END';
        'IF' U-Q>P-L 'THEN'
        'BEGIN'
          L1:=L; U1:=P-D; L:=Q+D;
        'END'
        'ELSE'
        'BEGIN'
          U1:=U; L1:=Q+D; U:=P-D;
        'END';
        'IF' I 'NOTEQUAL' J 'THEN'
        'BEGIN'
          'IF' U1>L1 'THEN'
          'BEGIN'
            M:=M+1; LT(/M/):=L; UT(/M/):=U; L:=L1; U:=U1;
            'GO TO' PART;
          'END';
          'IF' U>L 'THEN' 'GO TO' PART;
        'END';
      'END';
      'IF' M 'NOTEQUAL' 0 'THEN'
      'BEGIN'
        L:=LT(/M/); U:=UT(/M/); M:=M-1;
        'IF' U>L 'THEN' 'GO TO' PART;
      'END';
    'END' QUICKSORT;
    'INTEGER' I,J,P,Q,L,U,S;
    'REAL' X,Y;
    I:=L:=1;
    'FOR' I:=I+1 'WHILE' 2**I<N*LN(2)/LN(N) 'DO' L:=2**I-1;
    'IF' N>50 'THEN'
    'BEGIN'
      'REAL' 'ARRAY' TEMP(/0:L+1/);
      S:=(N-1)'/'(L-1); I:=1;
      QUICKSORT(A,1,1+(I-1)*S,S);
      'FOR' J:=1 'STEP' 1 'UNTIL' I 'DO'
      'BEGIN'
        TEMP(/J/):=A(/I,1/); I:=I+S; MOVE:=MOVE+1;
      'END';
      P:=L+1; TEMP(/0/):=0; TEMP(/P/):=N;
      'FOR' P:=P'/'2 'WHILE' P>0 'DO'
      'FOR' Q:=P 'STEP' 2*P 'UNTIL' L 'DO'
      'BEGIN'
        I:=TEMP(/Q-P/); U:=TEMP(/Q+P/);
        'IF' I+1<U 'THEN'
```

```
                'BEGIN'
                   J:=U+1; X:=TEMP(/Q/);
FIRST:
                   'FOR' I:=I+1 'WHILE' I<J 'DO'
                   'BEGIN'
                      COMP:=COMP+1;
                      'IF' A(/I,1/)>=X 'THEN' 'GO TO' SECOND;
                   'END';
                   I:=J;
SECOND:
                   'FOR' J:=J-1 'WHILE' J>I 'DO'
                   'BEGIN'
                      COMP:=COMP+1;
                      'IF' A(/J,1/)<=X 'THEN'
                      'BEGIN'
                         Y:=A(/I,1/); A(/I,1/):=A(/J,1/); A(/J,1/):=Y;
                         INTER:=INTER+1;
                         'GO TO' FIRST;
                      'END';
                   'END';
                   TEMP(/Q/):=I-1;
                'END'
                'ELSE'
                TEMP(/Q/):=U;
             'END';
             'FOR' I:=0 'STEP' 1 'UNTIL' L 'DO'
             'BEGIN'
                P:=TEMP(/I/)+1; Q:=TEMP(/I+1/);
                'IF' P<Q 'THEN' QUICKSORT(A,P,Q,1);
             'END';
          'END'
          'ELSE'
          QUICKSORT(A,1,N,1);
       'END' SAMPLESORT;
```

4.2 Qsort2

```
'PROCEDURE' QSORT2(A,L1,U1);
'VALUE' L1,U1; 'INTEGER' L1,U1; 'ARRAY' A;
'BEGIN'
   'INTEGER' P,Q,IX,IZ,I,J,M,L,U;
   'REAL' X,XX,Y,Z,ZZ;
   'INTEGER' 'ARRAY' LT,UT(/1:LN(ABS(U1-L1)+2)/LN(2)+0.01/);
   M:=0; L:=L1; U:=U1;
PART:
   P:=L; MOVE:=MOVE+2; Q:=U; X:=A(/P,1/);
   Z:=A(/Q,1/); I:=0; J:=Q-P-1; COMP:=COMP+1;
   'IF' X>Z 'THEN'
   'BEGIN'
      Y:=X; MOVE:=MOVE+2; A(/P,1/):=X:=Z; A(/Q,1/):=Z:=Y;
      INTER:=INTER+1;
   'END';
   'IF' U-L>1 'THEN'
   'BEGIN'
      XX:=X; MOVE:=MOVE+2; IX:=P; ZZ:=Z; IZ:=Q;
LEFT:
      'FOR' P:=P+1 'WHILE' P<Q 'DO'
      'BEGIN'
         X:=A(/P,1/); MOVE:=MOVE+1; COMP:=COMP+1;
         'IF' X>=XX 'THEN' 'GO TO' RIGHT;
      'END' P;
      P:=Q-1; 'GO TO' OUT;
RIGHT:
      'FOR' Q:=Q-1 'WHILE' Q>P 'DO'
      'BEGIN'
         Z:=A(/Q,1/); MOVE:=MOVE+1; COMP:=COMP+1;
         'IF' Z<=ZZ 'THEN' 'GO TO' DIST;
      'END' Q;
      Q:=P; MOVE:=MOVE+2; P:=P-1;
      Z:=X; X:=A(/P,1/); 'GO TO' COMPARE;
DIST:
      COMP:=COMP+1;
      'IF' X>Z 'THEN'
      'BEGIN'
         Y:=X; MOVE:=MOVE+2; A(/P,1/):=X:=Z; A(/Q,1/):=Z:=Y;
         INTER:=INTER+1;
COMPARE:
         COMP:=COMP+2;
         'IF' X>XX 'THEN'
         'BEGIN'
            XX:=X; MOVE:=MOVE+1; I:=I+1; IX:=P;
         'END';
         'IF' Z<ZZ 'THEN'
         'BEGIN'
            ZZ:=Z; MOVE:=MOVE+1; I:=I+1; IZ:=Q;
         'END';
      'END'
      'ELSE'
      'BEGIN'
         XX:=X; MOVE:=MOVE+2; IX:=P; ZZ:=Z; IZ:=Q; I:=I+2;
      'END';
      'GO TO' LEFT;
```

```
OUT:
          'IF' P 'NOTEQUAL' IX 'THEN'
          'BEGIN'
             A(/P,1/):=XX; A(/IX,1/):=X; MOVE:=MOVE+2;
          'END';
          'IF' Q 'NOTEQUAL' IZ 'THEN'
          'BEGIN'
             A(/Q,1/):=ZZ; A(/IZ,1/):=Z; MOVE:=MOVE+2;
          'END';
          'IF' U-Q>P-L 'THEN'
          'BEGIN'
             L1:=L; U1:=P-1; L:=Q+1;
          'END'
          'ELSE'
          'BEGIN'
             U1:=U; L1:=Q+1; U:=P-1;
          'END';
          'IF' I 'NOTEQUAL' J 'THEN'
          'BEGIN'
             'IF' U1>L1 'THEN'
             'BEGIN'
                M:=M+1; LT(/M/):=L; UT(/M/):=U; L:=L1; U:=U1;
                'GO TO' PART;
             'END';
             'IF' U>L 'THEN' 'GO TO' PART;
          'END';
       'END';
       'IF' M 'NOTEQUAL' 0 'THEN'
       'BEGIN'
          L:=LT(/M/); U:=UT(/M/); M:=M-1;
          'IF' U>L 'THEN' 'GO TO' PART;
       'END';
    'END' QSORT2;
```

```
'PROCEDURE' QUICKSORT(A,M,N);
'VALUE' M,N; 'INTEGER' M,N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J;
   'IF' M<N-1 'THEN'
   'BEGIN'
      PARTITION(A,M,N,I,J);
      QUICKSORT(A,M,J);
      QUICKSORT(A,I,N);
   'END'
   'ELSE'
   'IF' N-M=1 'THEN'
   'BEGIN'
      'IF' A(/N,1/)<A(/M,1/) 'THEN'
      'BEGIN'
         YY:=A(/M,1/); A(/M,1/):=A(/N,1/); A(/N,1/):=YY;
         INTER:=INTER+1;
      'END';
      COMP:=COMP+1;
   'END';
'END' QUICKSORT;

'PROCEDURE' PARTITION(A,M,N,I,J);
'VALUE' M,N; 'INTEGER' I,J,M,N; 'ARRAY' A;
'BEGIN'
   'REAL' X;
   I:=M-1; J:=N;
   'IF' M<N-2 'THEN'
   'BEGIN'
      X:=A(/N-(N-M+1)'/'2,1/); A(/N-(N-M+1)'/'2,1/):=A(/N,1/);
      MOVE:=MOVE+1;
   'END'
   'ELSE'
   X:=A(/N,1/);
LEFT:
   'FOR' I:=I+1 'WHILE' I<J 'DO'
   'BEGIN'
      COMP:=COMP+1;
      'IF' X<A(/I,1/) 'THEN'
      'BEGIN'
         A(/J,1/):=A(/I,1/); MOVE:=MOVE+1; 'GO TO' RIGHT;
      'END';
   'END' I;
   I:=J;
RIGHT:
   'FOR' J:=J-1 'WHILE' I<J 'DO'
   'BEGIN'
      COMP:=COMP+1;
      'IF' X>A(/J,1/) 'THEN'
      'BEGIN'
         A(/I,1/):=A(/J,1/); MOVE:=MOVE+1; 'GO TO' LEFT;
      'END';
   'END' J;
   A(/I,1/):=X; MOVE:=MOVE+2;
   'IF' I 'NOTEQUAL' M 'THEN' J:=I-1;
   'IF' I 'NOTEQUAL' N 'THEN' I:=I+1;
'END' PARTITION;
```

```
'PROCEDURE' MERGE(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J,K,L,M,D,P,Q,R; 'BOOLEAN' S;
   S:='FALSE'; P:=1;
PASS:
   S:='NOT' S; D:=1; Q:=P; R:=P;
   'IF' S 'THEN'
   'BEGIN'
      I:=1; J:=N; K:=N; L:=2*N+1;
   'END'
   'ELSE'
   'BEGIN'
      I:=N+1; J:=2*N; K:=0; L:=N+1;
   'END';
   'FOR' K:=K+D 'WHILE' K 'NOTEQUAL' L 'DO'
   'BEGIN'
      'IF' R>0 'THEN'
      'BEGIN'
         'IF' Q>0 'THEN'
         'BEGIN'
            COMP:=COMP+1; MOVE:=MOVE+1;
            'IF' A(/I,1/)>A(/J,1/) 'THEN'
LEFT:
            'BEGIN'
               A(/K,1/):=A(/J,1/); J:=J-1; R:=R-1;
            'END'
            'ELSE'
RIGHT:
            'BEGIN'
               A(/K,1/):=A(/I,1/); I:=I+1; Q:=Q-1;
            'END';
         'END'
         'ELSE'
         'GO TO' LEFT;
      'END'
      'ELSE'
      'IF' Q>0 'THEN' 'GO TO' RIGHT
      'ELSE'
      'BEGIN'
         Q:=P; R:=P; M:=K-D; K:=L; L:=M; D:=-D;
      'END';
   'END';
   P:=P+P;
   'IF' P<N 'THEN' 'GO TO' PASS;
   'IF' S 'THEN'
   'FOR' I:=1 'STEP' 1 'UNTIL' N 'DO' A(/I,1/):=A(/I+N,1/);
'END' MERGE;
```

```
'PROCEDURE' NATURALMERGE(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J,K,L,D,H; 'BOOLEAN' S,F;
   S:='FALSE';
PASS:
   S:='NOT' S;
   'IF' S 'THEN'
   'BEGIN'
      I:=1; J:=N; K:=N+1; L:=2*N;
   'END'
   'ELSE'
   'BEGIN'
      I:=N+1; J:=2*N; K:=1; L:=N;
   'END';
   D:=1; F:='TRUE'; H:=K;
   'FOR' K:=K+D 'WHILE' I 'NOTEQUAL' J 'DO'
   'BEGIN'
      H:=K; COMP:=COMP+1;
      'IF' A(/I,1/)>A(/J,1/) 'THEN'
      'BEGIN'
         A(/K-D,1/):=A(/J,1/); MOVE:=MOVE+1; J:=J-1; COMP:=COMP+1;
         'IF' A(/J+1,1/)>A(/J,1/) 'THEN'
         'BEGIN'
            A(/K,1/):=A(/I,1/); MOVE:=MOVE+1; COMP:=COMP+1;
            'FOR' K:=K+D 'WHILE' A(/I,1/)<=A(/I+1,1/) 'DO'
            'BEGIN'
               I:=I+1; A(/K,1/):=A(/I,1/); MOVE:=MOVE+1; H:=K;
               COMP:=COMP+1;
            'END';
            I:=I+1; F:='FALSE'; K:=L; L:=H+D; D:=-D;
         'END';
      'END'
      'ELSE'
      'BEGIN'
         A(/K-D,1/):=A(/I,1/); MOVE:=MOVE+1; I:=I+1; COMP:=COMP+1;
         'IF' A(/I-1,1/)>A(/I,1/) 'THEN'
         'BEGIN'
            A(/K,1/):=A(/J,1/); MOVE:=MOVE+1; COMP:=COMP+1;
            'FOR' K:=K+D 'WHILE' A(/J,1/)<=A(/J-1,1/) 'DO'
            'BEGIN'
               J:=J-1; A(/K,1/):=A(/J,1/); MOVE:=MOVE+1; H:=K;
               COMP:=COMP+1;
            'END';
            J:=J-1; F:='FALSE'; K:=L; L:=H+D; D:=-D;
         'END';
      'END';
      H:=K;
   'END';
   A(/H,1/):=A(/I,1/); MOVE:=MOVE+1;
   'IF' F 'THEN'
   'BEGIN'
      'IF' S 'THEN'
      'FOR' I:=1 'STEP' 1 'UNTIL' N 'DO' A(/I,1/):=A(/I+N,1/);
   'END'
   'ELSE'
   'GO TO' PASS;
'END' NATURALMERGE;
```

```
'PROCEDURE' LISTNATURALMERGE(A,LINK,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A; 'INTEGER' 'ARRAY' LINK;
'BEGIN'
   'INTEGER' I,P,Q,S,T;
   LINK(/0/):=1; T:=N+1; COMP:=COMP+N-1;
   'FOR' P:=1 'STEP' 1 'UNTIL' N-1 'DO'
   'IF' A(/P,1/)<=A(/P+1,1/) 'THEN'
   LINK(/P/):=P+1
   'ELSE'
   'BEGIN'
      LINK(/T/):=-(P+1); T:=P;
   'END';
   LINK(/T/):=LINK(/N/):=0; LINK(/N+1/):=ABS(LINK(/N+1/));
PASS:
   S:=0; T:=N+1; P:=LINK(/S/); Q:=LINK(/T/);
   'FOR' I:=SIGN(LINK(/S/)) 'WHILE' Q 'NOTEQUAL' 0 'DO'
   'BEGIN'
      COMP:=COMP+1;
      'IF' A(/P,1/)>A(/Q,1/) 'THEN'
      'BEGIN'
         LINK(/S/):=I*ABS(Q); S:=Q; Q:=LINK(/Q/);
         'IF' Q<=0 'THEN'
         'BEGIN'
            LINK(/S/):=P; S:=T;
            'FOR' T:=P 'WHILE' LINK(/P/)>0 'DO' P:=LINK(/P/);
            T:=P; P:=-LINK(/P/); Q:=-Q;
            'IF' Q=0 'THEN'
            'BEGIN'
               LINK(/S/):=SIGN(LINK(/S/))*ABS(P);
               LINK(/T/):=0; 'GO TO' PASS;
            'END';
         'END';
      'END'
      'ELSE'
      'BEGIN'
         LINK(/S/):=I*ABS(P); S:=P; P:=LINK(/P/);
         'IF' P<=0 'THEN'
         'BEGIN'
            LINK(/S/):=Q; S:=T;
            'FOR' T:=Q 'WHILE' LINK(/Q/)>0 'DO' Q:=LINK(/Q/);
            T:=Q; Q:=-LINK(/Q/); P:=-P;
            'IF' Q=0 'THEN'
            'BEGIN'
               LINK(/S/):=SIGN(LINK(/S/))*ABS(P);
               LINK(/T/):=0; 'GO TO' PASS;
            'END';
         'END';
      'END';
   'END';
'END' LISTNATURALMERGE;
```

4.7 Linearinsert

```
'PROCEDURE' LINEARINSERT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J; 'REAL' X;
   'FOR' I:=2 'STEP' 1 'UNTIL' N 'DO'
   'BEGIN'
      X:=A(/I,1/); MOVE:=MOVE+1;
      'FOR' J:=I-1,J-1 'WHILE' J>0 'DO'
      'BEGIN'
         COMP:=COMP+1;
         'IF' A(/J,1/)>X 'THEN'
         A(/J+1,1/):=A(/J,1/)
         'ELSE'
         'GO TO' LOOP;
         MOVE:=MOVE+1;
      'END' J;
      J:=0;
LOOP:
      A(/J+1,1/):=X; MOVE:=MOVE+1;
   'END' I;
'END' LINEARINSERT;
```

4.8 Binaryinsert

```
'PROCEDURE' BINARYINSERT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,K,P,Q; 'REAL' X;
   'FOR' I:=2 'STEP' 1 'UNTIL' N 'DO'
   'BEGIN'
      X:=A(/I,1/); P:=1; Q:=I; MOVE:=MOVE+1;
      'FOR' K:=(P+Q)'/'2 'WHILE' Q>P+1 'DO'
      'BEGIN'
         'IF' X>=A(/K,1/) 'THEN'
         P:=K
         'ELSE'
         Q:=K;
         COMP:=COMP+1;
      'END' K;
      COMP:=COMP+1;
      'IF' X>=A(/P,1/) 'THEN' P:=P+1;
      'IF' P<I 'THEN'
      'BEGIN'
         'FOR' K:=I,K-1 'WHILE' K>P 'DO' A(/K,1/):=A(/K-1,1/);
         A(/P,1/):=X; MOVE:=MOVE+I-P+2;
      'END';
   'END' I;
'END' BINARYINSERT;
```

4.9 Listinsert

```
'PROCEDURE' LISTINSERT(A,LINK,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A; 'INTEGER' 'ARRAY' LINK;
'BEGIN'
   'INTEGER' P,Q;
   LINK(/0/):=N; LINK(/N/):=0;
   'FOR' N:=N-1 'WHILE' N>0 'DO'
   'BEGIN'
      P:=LINK(/0/); Q:=0; COMP:=COMP+1;
LOOP:
      'IF' A(/N,1/)>A(/P,1/) 'THEN'
      'BEGIN'
         Q:=P; P:=LINK(/Q/); COMP:=COMP+1;
         'IF' P>0 'THEN' 'GO TO' LOOP;
      'END';
      LINK(/Q/):=N; LINK(/N/):=P;
   'END' N;
'END' LISTINSERT;
```

4.10 Linearselect

```
'PROCEDURE' LINEARSELECT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J; 'REAL' X;
   'FOR' N:=N 'STEP' -1 'UNTIL' 1 'DO'
   'BEGIN'
      J:=N;
      'FOR' I:=N-1 'STEP' -1 'UNTIL' 1 'DO'
      'BEGIN'
         'IF' A(/I,1/)>A(/J,1/) 'THEN' J:=I;
         COMP:=COMP+1;
      'END' I;
      'IF' J 'NOTEQUAL' N 'THEN'
      'BEGIN'
         X:=A(/J,1/); A(/J,1/):=A(/N,1/); A(/N,1/):=X;
         INTER:=INTER+1;
      'END';
   'END' U;
'END' LINEARSELECT;
```

4.11 Mergeselect

```
'PROCEDURE' MERGESELECT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'PROCEDURE' LINEARSELECT(L,U);
   'VALUE' L,U; 'INTEGER' L,U;
   'BEGIN'
      'INTEGER' I,J;
      'FOR' U:=U 'STEP' -1 'UNTIL' L 'DO'
      'BEGIN'
         J:=U;
         'FOR' I:=U-1 'STEP' -1 'UNTIL' L 'DO'
         'BEGIN'
            'IF' A(/I,1/)>A(/J,1/) 'THEN' J:=I;
            COMP:=COMP+1;
         'END' I;
         'IF' J 'NOTEQUAL' U 'THEN'
         'BEGIN'
            YY:=A(/J,1/); A(/J,1/):=A(/U,1/); A(/U,1/):=YY;
            INTER:=INTER+1;
         'END';
      'END' U;
   'END' LINEARSELECT;
   'INTEGER' M,I,J,K,Q;
   Q:=ENTIER(SQRT(N)+1); M:=N'/'Q+1;
   'IF' N=(M-1)*Q 'THEN' M:=M-1;
   'BEGIN'
      'INTEGER' 'ARRAY' P(/1:M/);
      'FOR' I:=1 'STEP' 1 'UNTIL' M-1 'DO'
      'BEGIN'
         P(/I/):=(I-1)*Q+1; LINEARSELECT(P(/I/),I*Q);
      'END';
      P(/M/):=(M-1)*Q+1;
      LINEARSELECT(P(/M/),N);
      'FOR' K:=N+1 'STEP' 1 'UNTIL' 2*N 'DO'
      'BEGIN'
         'FOR' I:=1 'STEP' 1 'UNTIL' M-1 'DO'
         'IF' P(/I/) 'NOTEQUAL' I*Q+1 'THEN' 'GO TO' SELECT;
         I:=M;
SELECT:
         'FOR' J:=I+1 'STEP' 1 'UNTIL' M 'DO'
         'IF' P(/J/) 'NOTEQUAL' J*Q+1
         'AND' P(/J/) 'NOTEQUAL' N+1 'THEN'
         'BEGIN'
            COMP:=COMP+1;
            'IF' A(/P(/J/),1/)<A(/P(/I/),1/) 'THEN' I:=J;
         'END';
         A(/K,1/):=A(/P(/I/),1/); P(/I/):=P(/I/)+1;
         MOVE:=MOVE+1;
      'END' K;
   'END';
'END' MERGESELECT;
```

4.12 Bubblesort

```
'PROCEDURE' BUBBLESORT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J; 'REAL' X;
   'FOR' I:=1 'STEP' 1 'UNTIL' N-1 'DO'
   'BEGIN'
      'FOR' J:=I 'STEP' -1 'UNTIL' 1 'DO'
      'BEGIN'
         COMP:=COMP+1;
         'IF' A(/J+1,1/)<A(/J,1/) 'THEN'
         'BEGIN'
            X:=A(/J+1,1/); A(/J+1,1/):=A(/J,1/);
            A(/J,1/):=X; INTER:=INTER+1;
         'END'
         'ELSE'
         'GO TO' LOOP;
      'END' J;
LOOP:
   'END' I;
'END' BUBBLESORT;
```

4.13 Shellsort

```
'PROCEDURE' SHELLSORT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'INTEGER' I,J,D,K; 'REAL' X;
   D:=I:=4;
   'FOR' I:=3*I+1 'WHILE' I<N 'DO' D:=3*D+1;
   'FOR' D:=(D-1)'/'3 'WHILE' D>0 'DO'
   'FOR' J:=1 'STEP' 1 'UNTIL' N-D 'DO'
   'BEGIN'
      X:=A(/J+D,1/); K:=J+D; MOVE:=MOVE+1;
      'FOR' I:=J,I-D 'WHILE' I>0 'DO'
      'BEGIN'
         COMP:=COMP+1;
         'IF' X<A(/I,1/) 'THEN'
         'BEGIN'
            A(/I+D,1/):=A(/I,1/); K:=I;
            MOVE:=MOVE+1;
         'END'
         'ELSE'
         'GO TO' LOOP;
      'END' I;
LOOP:
      'IF' K 'NOTEQUAL' J+D 'THEN'
      'BEGIN'
         A(/K,1/):=X; MOVE:=MOVE+1;
      'END';
   'END' J;
'END' SHELLSORT;
```

4.14 Heapsort

```
'PROCEDURE' HEAPSORT(A,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A;
'BEGIN'
   'PROCEDURE' SIFTUP(I,N);
   'VALUE' I,N; 'INTEGER' I,N;
   'BEGIN'
      'INTEGER' J; 'REAL' X;
      X:=A(/I,1/); MOVE:=MOVE+1;
LOOP:
      J:=2*I;
      'IF' J<=N 'THEN'
      'BEGIN'
         'IF' J<N 'THEN'
         'BEGIN'
            'IF' A(/J+1,1/)>A(/J,1/) 'THEN' J:=J+1;
            COMP:=COMP+1;
         'END';
         COMP:=COMP+1;
         'IF' A(/J,1/)>X 'THEN'
         'BEGIN'
            A(/I,1/):=A(/J,1/); MOVE:=MOVE+1; I:=J; 'GO TO' LOOP;
         'END';
      'END';
      A(/I,1/):=X; MOVE:=MOVE+1;
   'END' SIFTUP;
   'INTEGER' I; 'REAL' X;
   'FOR' I:=N'/'2 'STEP' -1 'UNTIL' 2 'DO' SIFTUP(I,N);
   'FOR' I:=N 'STEP' -1 'UNTIL' 2 'DO'
   'BEGIN'
      SIFTUP(1,I);
      'BEGIN'
         X:=A(/1,1/); A(/1,1/):=A(/I,1/); A(/I,1/):=X;
         INTER:=INTER+1;
      'END';
   'END' I;
'END' HEAPSORT;
```

```
'PROCEDURE' ORDER(A,LINK,N);
'VALUE' N; 'INTEGER' N; 'ARRAY' A; 'INTEGER' 'ARRAY' LINK;
'BEGIN'
   'INTEGER' I,K,P,T; 'REAL' X;
   P:=LINK(/0/);
   'FOR' K:=1 'STEP' 1 'UNTIL' N 'DO'
   'BEGIN'
      'FOR' I:=P 'WHILE' I<K 'DO' P:=LINK(/P/);
      X:=A(/K,1/); A(/K,1/):=A(/P,1/); A(/P,1/):=X;
      T:=LINK(/K/); LINK(/K/):=LINK(/P/); LINK(/P/):=T;
      T:=LINK(/K/); LINK(/K/):=P; P:=T;
   'END' K;
'END' ORDER;
```

5. RANDOM INPUT DATA

The random numbers used in these tests were generated in two stages. Firstly, a file of random integers between 0 and 10000 was created on a Honeywell 6050 computer using the FORTRAN programme below.

The linear congruential method was used with a multiplier of 16807=7 and a modulus of 2 . The starting value, 538965, is relatively prime to the modulus. The file of integers was created by choosing four digits from the centre of each number generated.

The second stage was performed in the mainline of the test programmes. The integers produced in the first stage were read into an array and then arrays of random real numbers were generated from them by setting A(i):=A(i)/A(i+j)*A(i+j+1) for i=1,2,3,...,n and j=1,2,...,k where k is the number of test runs, and n is the number of elements being sorted.

None of the arrays of real numbers produced by this second stage were found to contain any repeated elements.

A two stage random number generator was used because tests of a different nature were being run on these sorting algorithms on the Honeywell 6050 and comparable results were desired.

```
      INTEGER A,X,R,I
      X=538965
      A=16807
      I=0
  100 IF(I.GE.1000)STOP
  110 X=X*A
      R=IABS(X/10**6)
      R=R-(R/10000)*10000
      IF(R.LE.0)GO TO 110
      WRITE(10,15)R
   15 FORMAT(1X,I4)
      I=I+1
      GO TO 100
      END
```

6. TEST RESULTS

The next three sections summarize the results obtained from three sets of test runs.

In all three sets of tests, CP and CMS were locked into real storage to prevent them from being paged out. This eliminated all paging activity except that associated with the programmes and their data areas.

As mentioned previously the size of the working set (the number of real pages available for paging operations) was controlled by locking in pages of real storage to the operator and system thus making them unavailable to the sorting programmes.

The numbers of page reads and page writes performed while sorting were obtained by accessing the information available in the VMBLOK immediately before and after the sort routine was called.

The elements of the array being sorted were spaced equal distances apart on the pages in virtual storage. The number of elements per virtual page was varied as indicated in the tables.

It should be noted that the figures given for LISTINSERT and LISTNATURALMERGE do not include the data movement and paging operations that were used by ORDER to reorder the array of elements being sorted. The results for ORDER are listed separately.

It should also be noted that at the completion of MERGESELECT, the sorted numbers will be in a second data area and must be moved back to the original data area. The figures for MERGESELECT include these operations.

The horizontal scale for the graphs is log2(number of array elements on each virtual page). A logarithmic scale was used to improve readability.

Table 1 below gives the number of comparisons, interchanges, and moves of elements of the array being sorted used by each sorting algorithm. An exchange can be considered equal to three moves. The results summarized in table 1 apply to the tests of the following two sections and are averages of tests on sets of 256 random numbers.

NUMBER OF TEST RUNS AVERAGED

	10			5		
	comp	inter	moves	comp	inter	moves
Samplesort	2253.1	392.8	1822.4	2244.6	394.2	1813.0
Qsort2	2667.5	367.0	3097.4	2671.8	371.4	3094.2
Quicksort	2133.1	21.0	1070.5	2043.6	21.4	1069.8
Merge	1728.0	0.0	1728.0	1724.4	0.0	1724.4
Naturalmerge	3656.1	0.0	1920.0	3605.6	0.0	1894.4
Listnaturalmerge	2030.5	0.0	0.0	2031.6	0.0	0.0
Linearinsert	17287.5	0.0	17547.9	17307.0	0.0	17567.6
Binaryinsert	1943.2	0.0	17788.3	1945.4	0.0	17809.8
Listinsert	17292.9	0.0	0.0	17312.6	0.0	0.0
Linearselect	32640.0	250.1	0.0	32640.0	248.8	0.0
Mergeselect	5564.5	201.8	512.0	5557.4	201.4	512.0
Bubblesort	17287.5	17037.9	0.0	17307.0	17057.6	0.0
Shellsort	2481.1	0.0	3193.4	2440.4	0.0	3152.0
Heapsort	3313.5	255.0	2331.5	3314.2	255.0	2331.8
Order	0.0	256.0	0.0	0.0	256.0	0.0

TABLE 1 - Comparisons, Interchanges, and Moves.

7. FIRST SET OF TESTS

This section summarizes the results of test runs on different sets of 256 random numbers. Table 2 indicates the number of sets sorted for each algorithm.

The virtual storage of the virtual machine performing the tests was defined to be 4096K bytes.

As well as locking in CP and CMS, the programmes were locked into real storage as were all data areas except the array being sorted. This was done in order to isolate the paging activity associated with the array.

A dummy array was declared immediately below the array being sorted, the size of which was such that the array being sorted was forced to start on a virtual page boundary. The array being sorted was declared as

```
REAL ARRAY A(/1:N*2*I,1:1024/I/)
```

where I was the number of elements on each virtual page. The array A was, therefore, always 512 virtual pages in length.

The two pages following array A were also locked into real storage. This ensured that the array being sorted was the only pageable part of the virtual machine. A working set of 8 pages was allowed for this array, the remaining pages being locked into the operator and system.

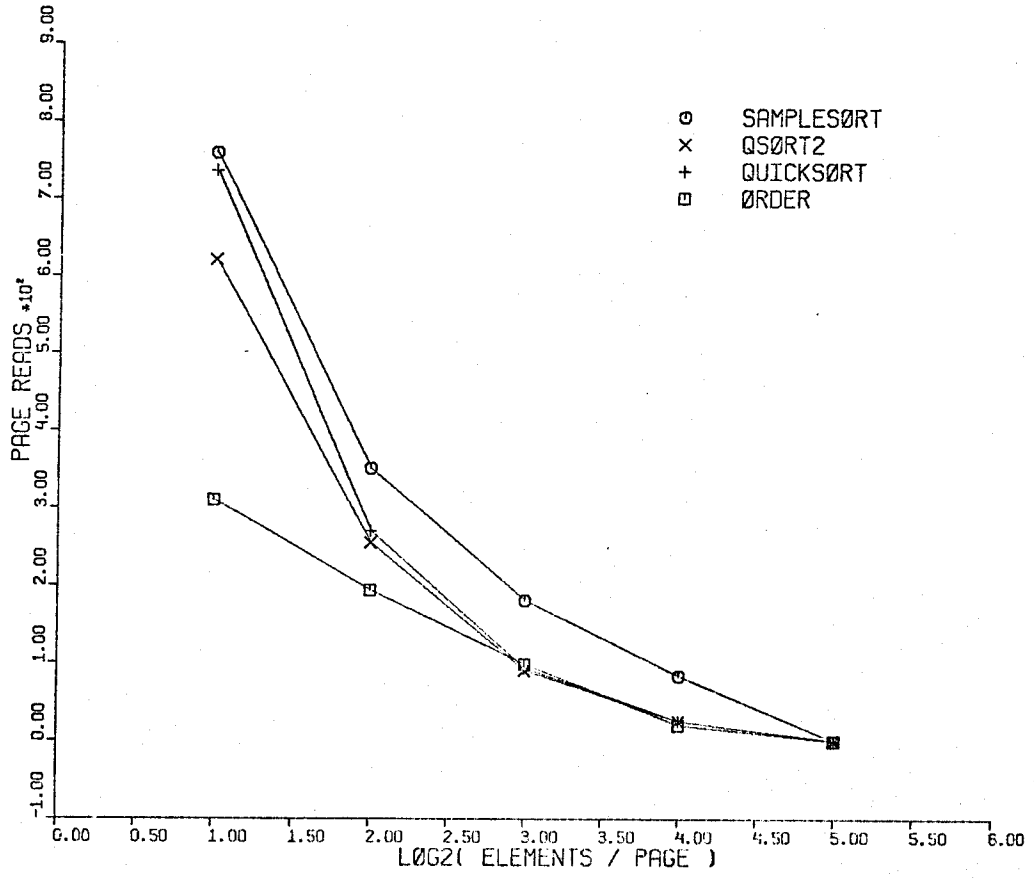
	ELEMENTS / PAGE						
	64	32	16	8	4	2	1
Samplesort	--	5	5	5	5	3	--
Qsort2	--	5	5	5	5	3	--
Quicksort	--	5	5	5	5	3	--
Merge	5	5	5	5	5	3	3
Naturalmerge	5	5	5	5	5	3	3
Listnaturalmerge	--	5	5	5	5	3	--
Linearinsert	--	5	5	5	5	3	--
Binaryinsert	--	5	5	5	5	3	1
Listinsert	--	5	5	5	5	3	--
Linearselect	--	5	5	5	5	3	--
Mergeselect	5	5	5	5	5	3	3
Bubblesort	--	5	5	5	5	3	--
Shellsort	--	5	5	5	5	3	--
Heapsort	--	5	5	5	5	3	--
Order	--	9	10	10	10	6	--

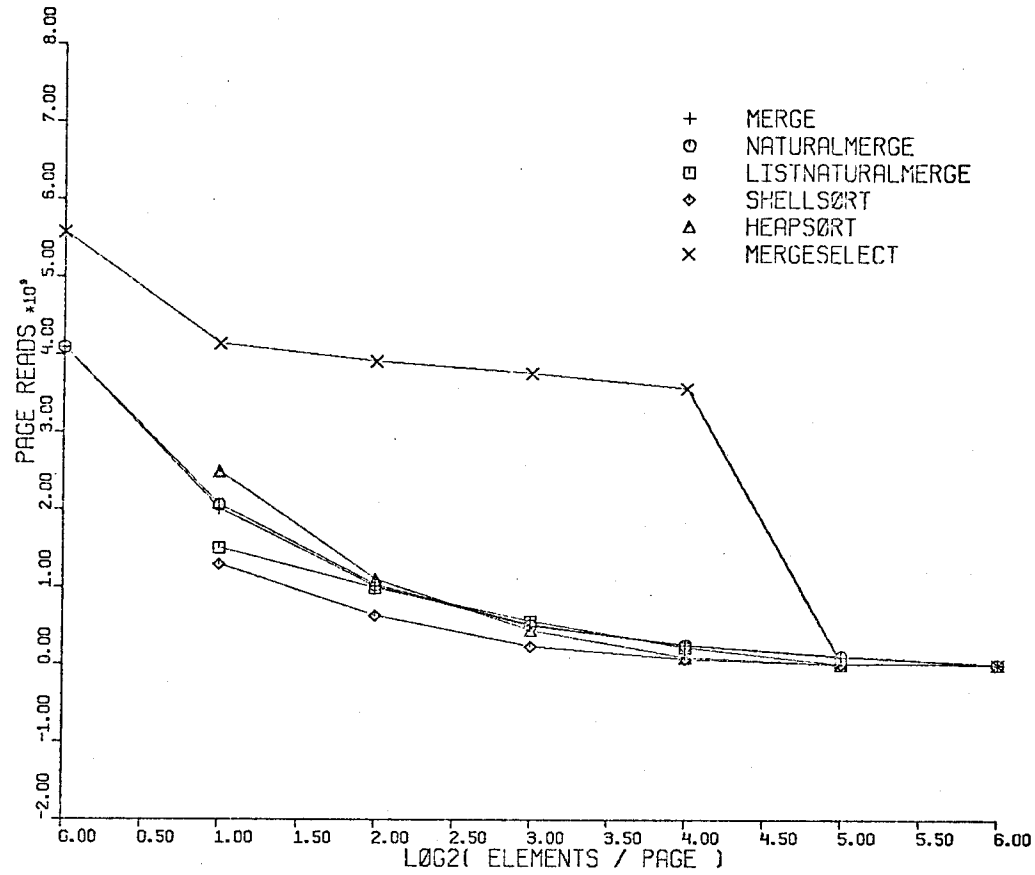
TABLE 2 - Number of files tested.

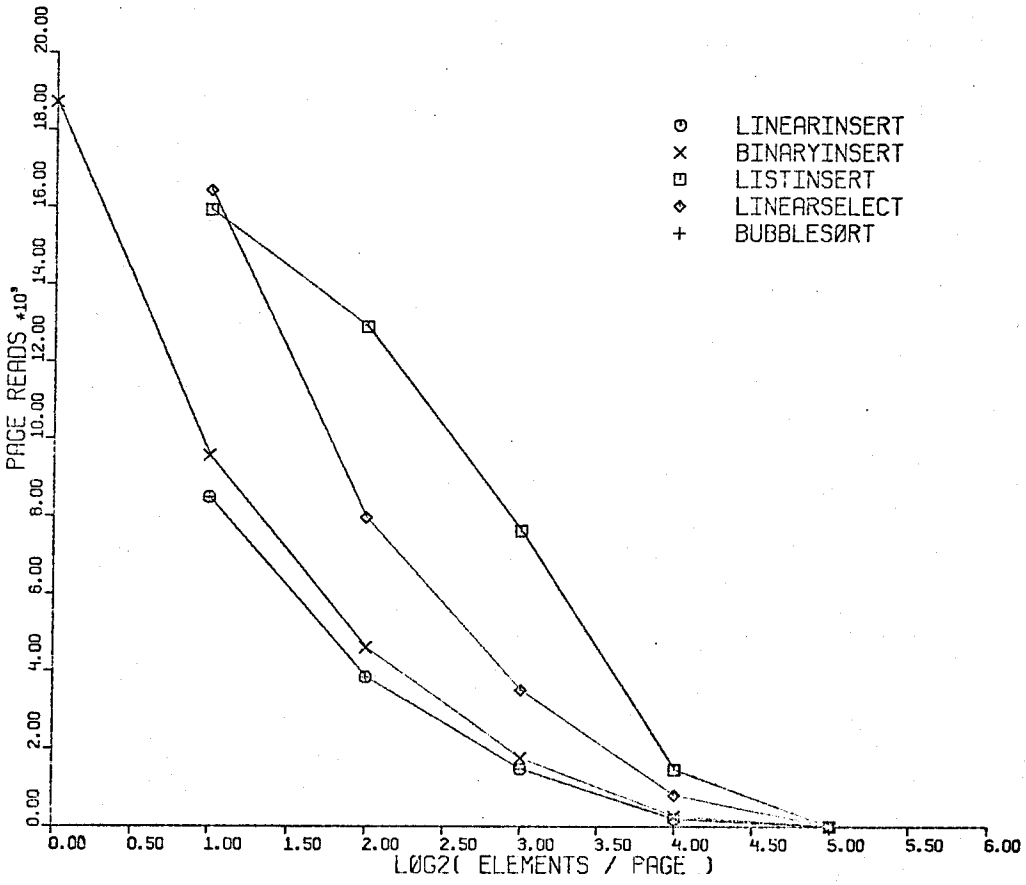
ELEMENTS / PAGE

	64	32	16	8	4	2	1
Samplesort	--	0.0	83.2	180.6	351.2	759.0	--
Qsort2	--	0.0	20.6	90.0	255.4	621.0	--
Quicksort	--	0.0	24.6	93.0	270.2	736.3	--
Merge	0.0	99.4	248.2	500.2	1008.8	2006.0	4091.7
Naturalmerge	0.0	106.0	251.4	509.4	1028.2	2056.7	4090.3
Listnaturalmerge	--	0.0	221.0	557.0	989.8	1500.0	--
Linearinsert	--	0.0	208.6	1474.2	3833.0	8477.3	--
Binaryinsert	--	0.0	258.0	1753.0	4595.8	9559.3	18728.0
Listinsert	--	0.0	1451.6	7622.8	12907.0	15926.0	--
Linearselect	--	0.0	805.6	3496.8	7965.8	16416.3	--
Mergeselect	0.0	20.3	3583.0	3772.2	3921.6	4138.3	5569.0
Bubblesort	--	0.0	184.8	1470.4	3833.4	8481.3	--
Shellsort	--	0.0	73.4	228.4	632.0	1291.3	--
Heapsort	--	0.0	88.4	439.8	1101.8	2486.7	--
Order	--	0.0	20.1	98.1	193.6	309.5	--

TABLE 3 - Page Reads.

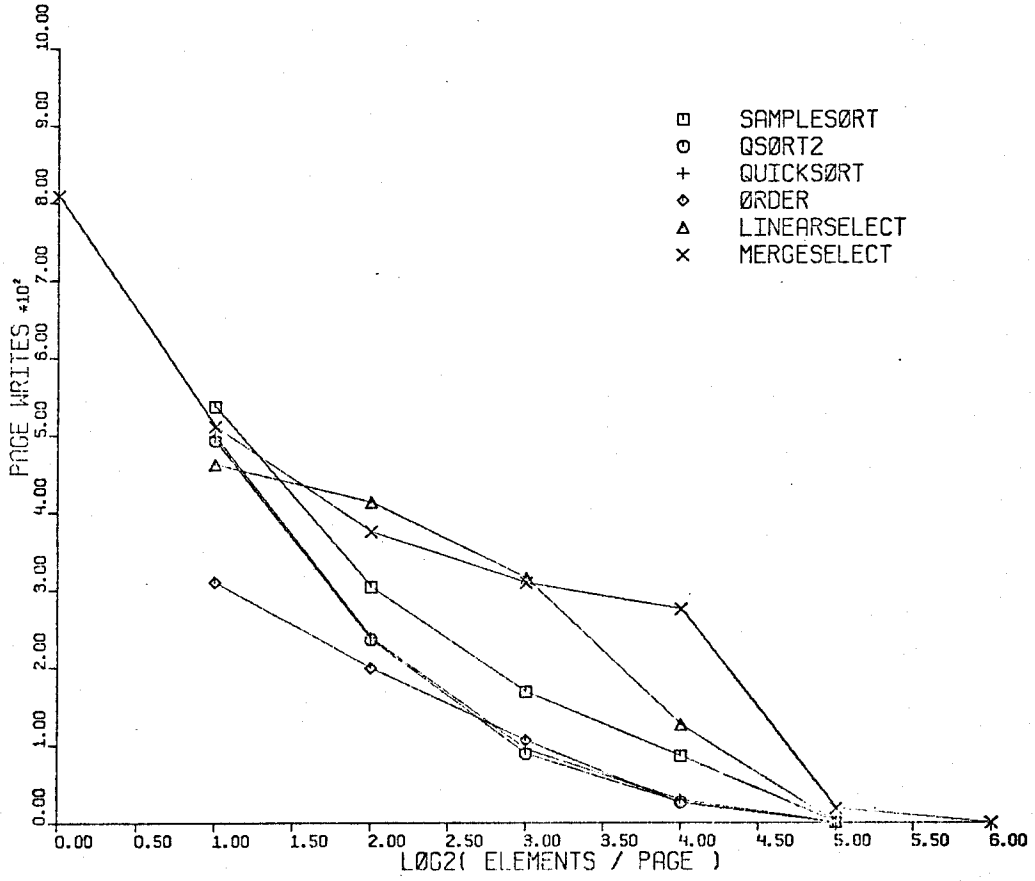


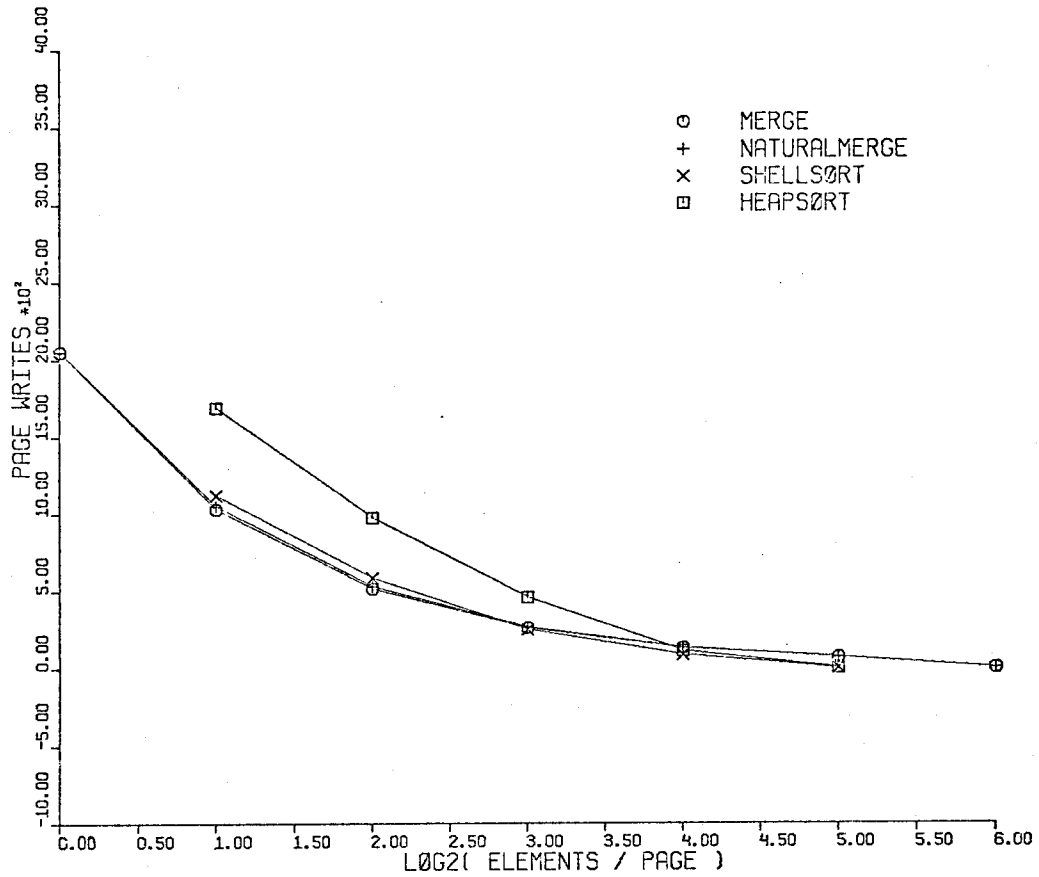

MERGE
NATURALMERGE
LISTNATURALMERGE
SHELLSØRT
HEAPSØRT
MERGESELECT
PAGE READS *10³
LØG2(ELEMENTS / PAGE)

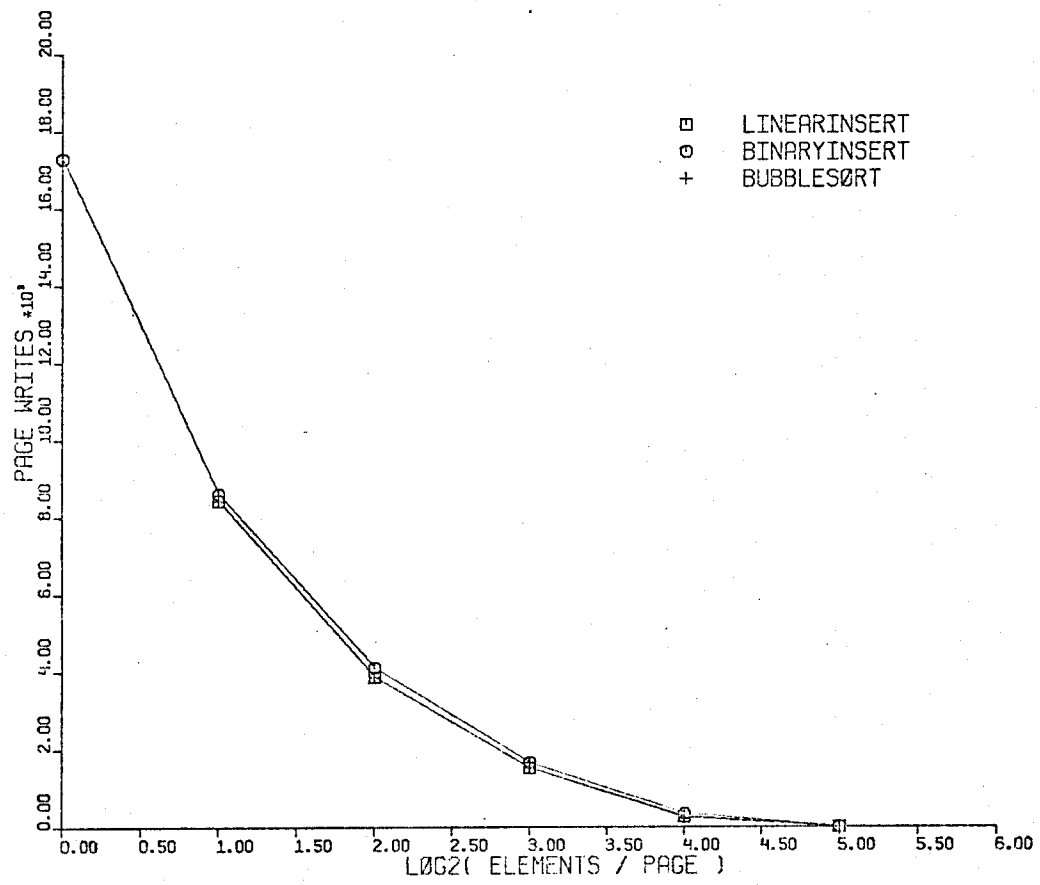

LINEARINSERT
BINARYINSERT
LISTINSERT
LINEARSELECT
BUBBLESØRT
PAGE READS *10³
LØG2(ELEMENTS / PAGE)

	ELEMENTS / PAGE						
	64	32	16	8	4	2	1
Samplesort	--	0.0	86.2	168.0	304.0	537.0	--
Qsort2	--	0.0	26.4	89.4	235.8	492.7	--
Quicksort	--	0.0	29.2	94.6	238.4	498.0	--
Merge	0.0	68.4	131.2	257.8	516.4	1027.7	2049.7
Naturalmerge	0.0	68.2	132.2	265.2	528.8	1047.7	2050.0
Listnaturalmerge	--	0.0	10.2	10.2	10.2	8.7	--
Linearinsert	--	0.0	252.4	1518.0	3864.4	8423.3	--
Binaryinsert	--	0.0	332.6	1660.4	4104.2	8594.0	17281.0
Listinsert	--	0.0	10.2	10.2	10.2	8.7	--
Linearselect	--	0.0	125.8	315.0	413.2	462.0	--
Mergeselect	0.0	18.0	275.2	308.8	374.8	511.0	808.7
Bubblesort	--	0.0	237.2	1513.4	3867.2	8423.7	--
Shellsort	--	0.0	82.8	248.2	587.0	1119.3	--
Heapsort	--	0.0	111.6	459.6	973.0	1689.7	--
Order	--	0.0	24.6	106.0	199.3	310.0	--

TABLE 4 - Page Writes.



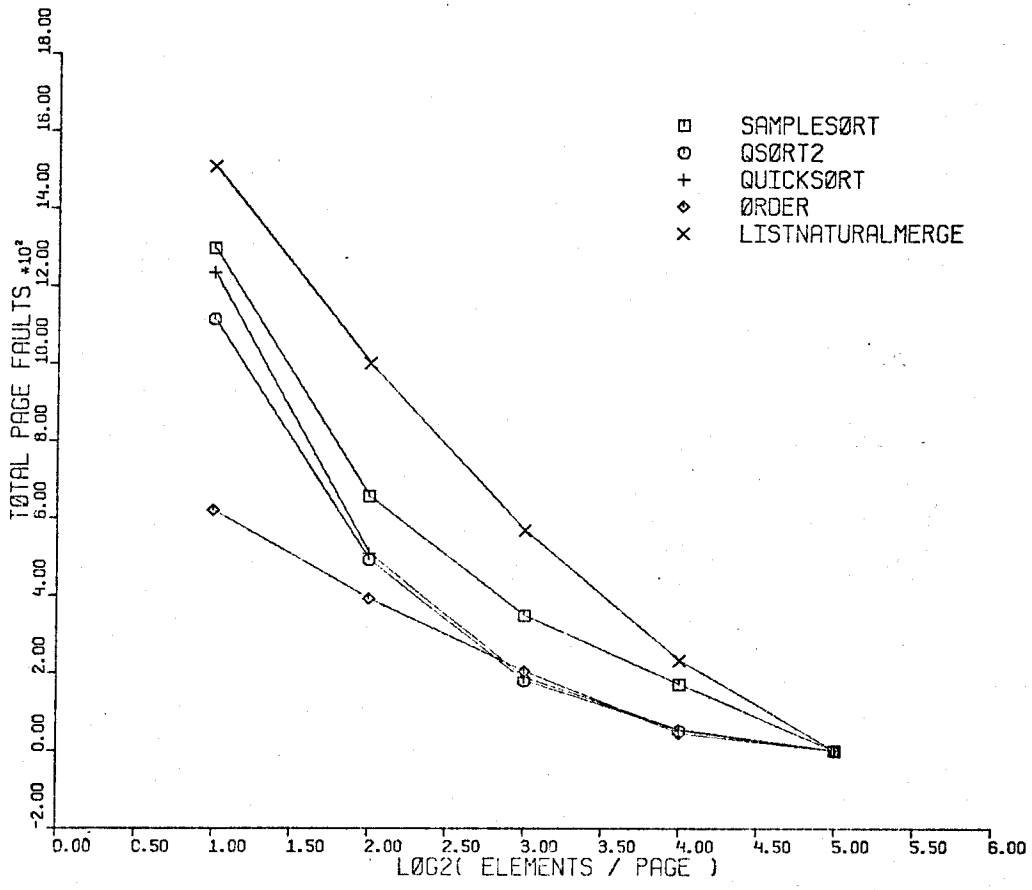

MERGE
NATURALMERGE
SHELLSØRT
HEAPSØRT
PAGE WRITES *10²
LØG2(ELEMENTS / PAGE)

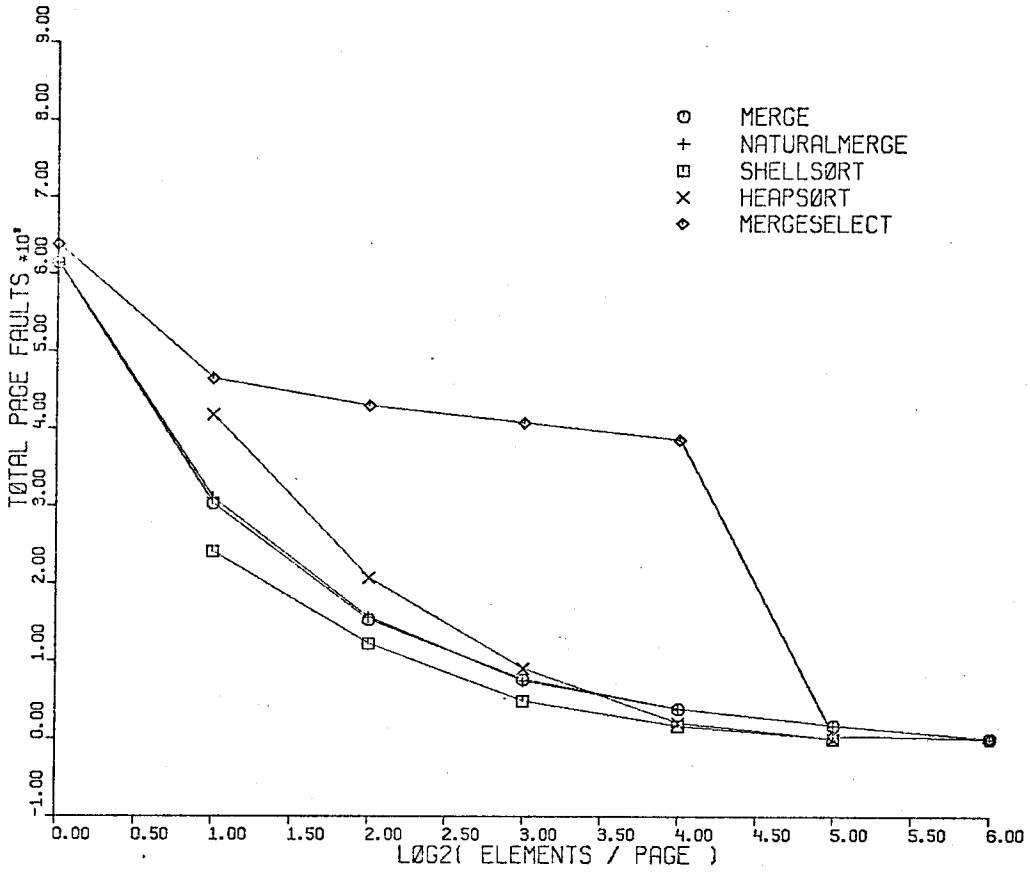

LINEARINSERT
BINARYINSERT
BUBBLESØRT
PAGE WRITES *10²
LØG2(ELEMENTS / PAGE)

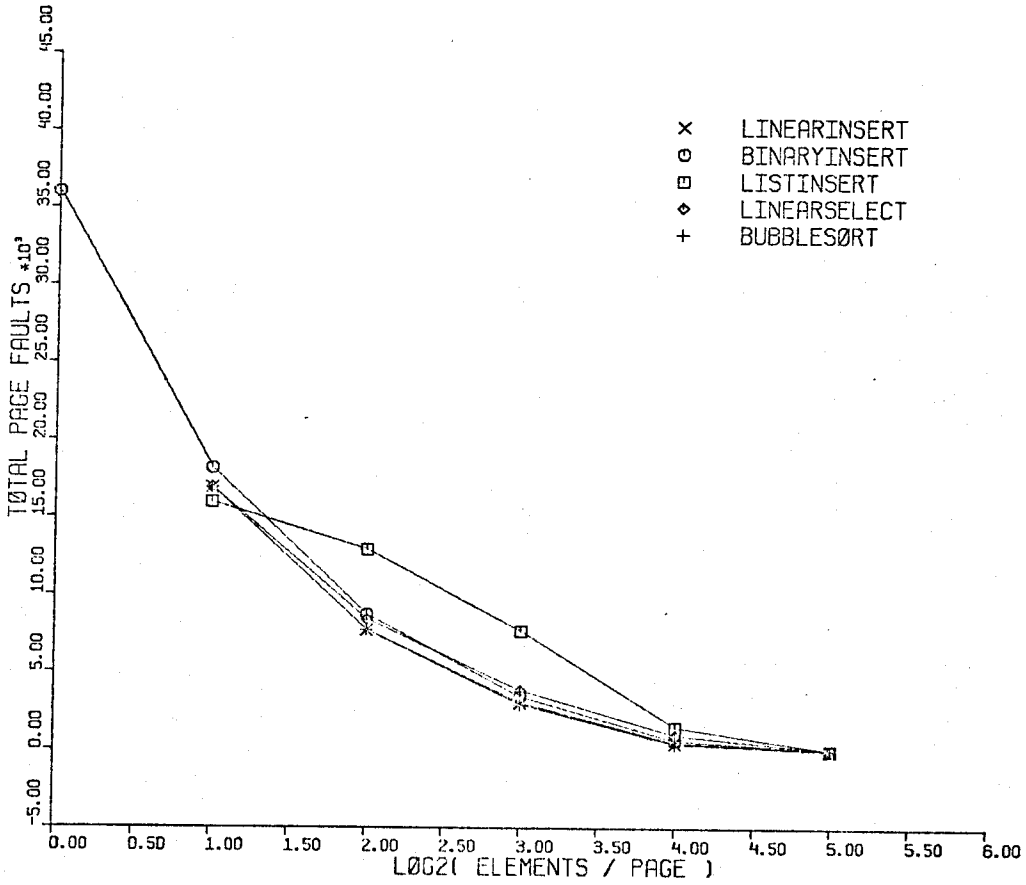
ELEMENTS / PAGE

	64	32	16	8	4	2	1
Samplesort	--	0.0	169.4	348.6	655.2	1296.0	--
Qsort2	--	0.0	51.0	179.4	491.2	1113.7	--
Quicksort	--	0.0	53.8	187.6	508.6	1234.3	--
Merge	0.0	167.8	379.4	758.0	1525.2	3033.7	6141.3
Naturalmerge	0.0	174.2	383.6	744.6	1557.0	3104.3	6140.3
Listnaturalmerge	--	0.0	231.2	567.2	1000.0	1508.7	--
Linearinsert	--	0.0	461.0	2992.2	7697.4	16900.7	--
Binaryinsert	--	0.0	590.6	3413.4	8700.0	18153.3	36009.0
Listinsert	--	0.0	1461.8	7633.0	12917.2	15934.7	--
Linearselect	--	0.0	931.4	3811.8	8379.0	16878.3	--
Mergeselect	0.0	38.3	3858.2	4081.0	4296.4	4649.3	6377.7
Bubblesort	--	0.0	422.0	2893.8	7700.6	16905.0	--
Shellsort	--	0.0	156.2	476.6	1219.0	2410.7	--
Heapsort	--	0.0	200.0	899.4	2074.8	4176.3	--
Order	--	0.0	44.7	204.1	392.9	619.5	--

TABLE 5 - Total Page Faults.




MERGE
NATURALMERGE
SHELLSØRT
HEAPSØRT
MERGESELECT
TØTAL PAGE FAULTS *10³
LØG2(ELEMENTS / PAGE)


LINEARINSERT
BINARYINSERT
LISTINSERT
LINEARSELECT
BUBBLESØRT
TØTAL PAGE FAULTS *10³
LØG2(ELEMENTS / PAGE)

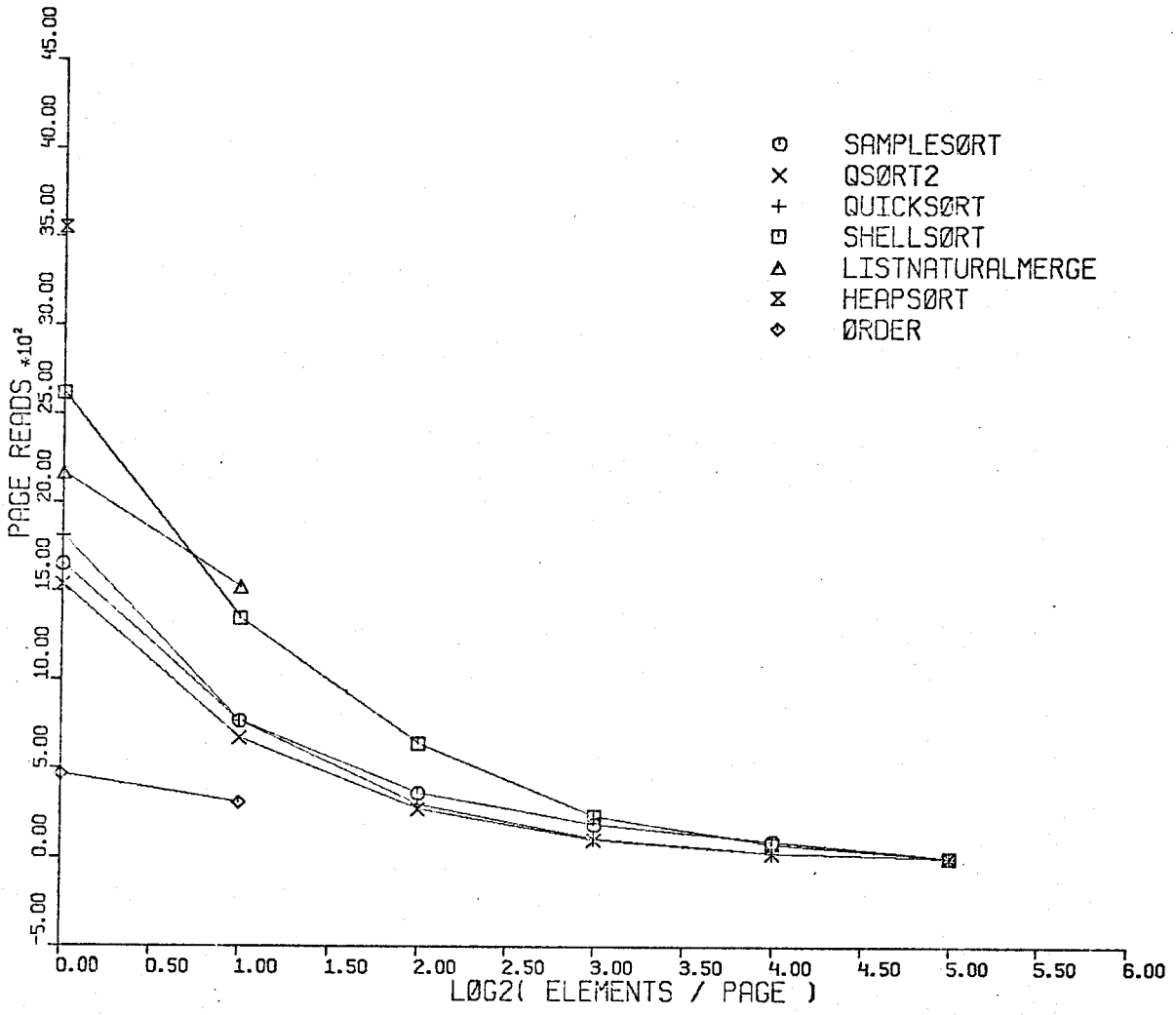
8. SECOND SET OF TESTS

The results in this section are averages of ten test runs on sets of 256 random numbers.

The data areas in the first two pages after the array being sorted were not locked into real storage. Except for this change, the test programmes were the same as those of the previous section, as were the virtual and real environments in which the tests were run.

	ELEMENTS / PAGE					
	32	16	8	4	2	1
Samplesort	0.0	91.4	184.0	359.5	765.4	1648.1
Qsort2	0.0	26.7	97.7	270.9	667.5	1532.6
Quicksort	0.0	25.4	102.6	292.7	764.8	1807.8
Listnaturalmerge	--	--	--	--	1522.0	2164.7
Shellsort	0.0	72.9	230.9	641.9	1343.4	2614.6
Heapsort	--	--	--	--	--	3549.1
Order	--	--	--	--	306.1	464.5

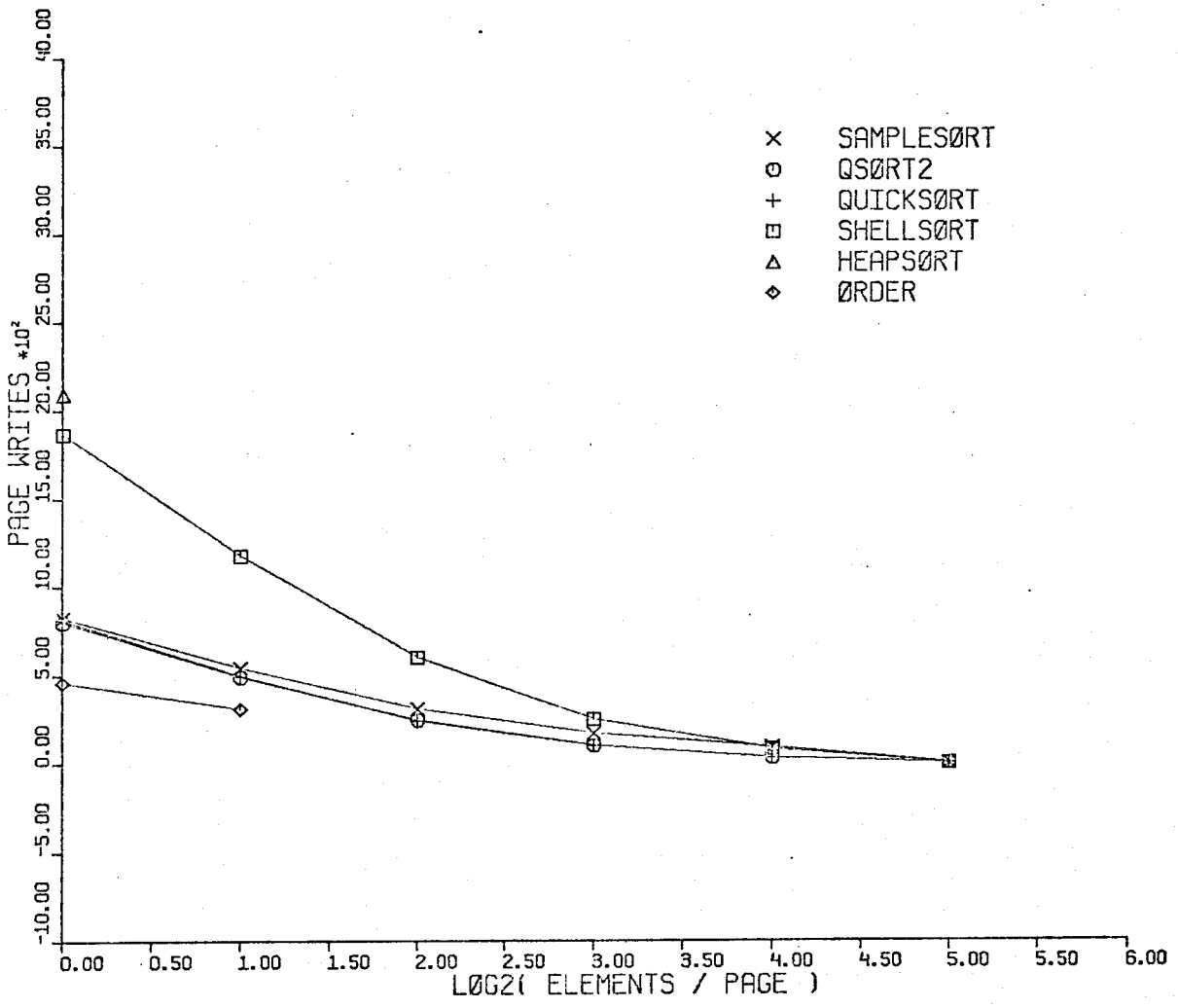
TABLE 6 - Page Reads.



ELEMENTS / PAGE

	32	16	8	4	2	1
Samplesort	0.0	90.4	170.1	307.5	542.4	826.5
Qsort2	0.0	30.4	100.1	246.2	492.3	798.0
Quicksort	0.0	30.3	99.7	241.9	493.8	808.4
Listnaturalmerge	--	--	--	--	6.9	6.3
Shellsort	0.0	81.2	252.2	598.4	1173.6	1863.3
Heapsort	--	--	--	--	--	2088.6
Order	--	--	--	--	307.8	461.1

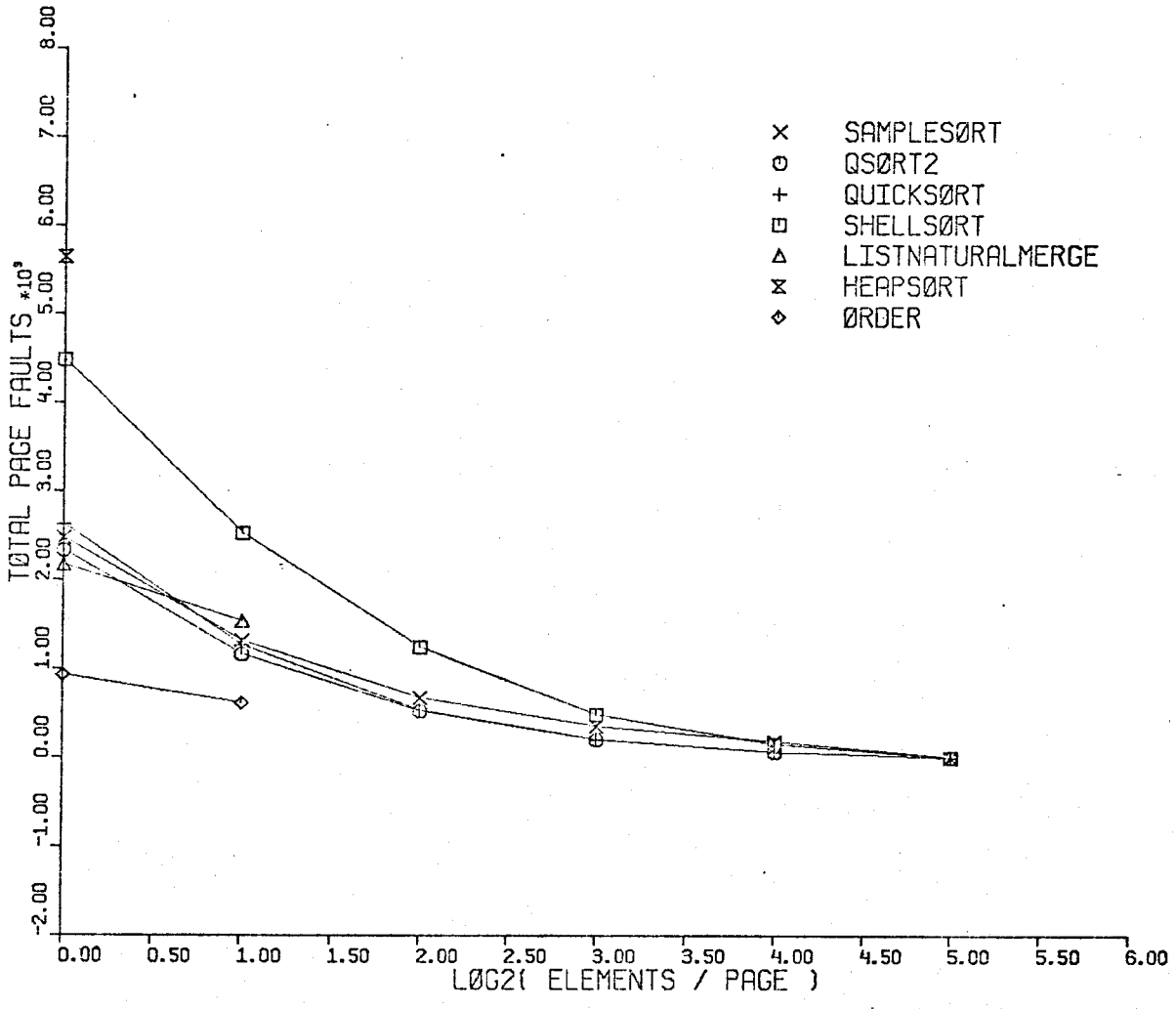
TABLE 7 - Page Writes.



ELEMENTS / PAGE

	32	16	8	4	2	1
Samplesort	0.0	181.8	354.1	667.0	1307.8	2474.6
Qsort2	0.0	57.1	107.8	517.1	1159.8	2330.6
Quicksort	0.0	55.7	202.3	534.6	1258.6	2616.2
Listnaturalmerge	--	--	--	--	1528.9	2171.0
Shellsort	0.0	154.1	483.1	1240.3	2517.0	4477.9
Heapsort	--	--	--	--	--	5637.7
Order	--	--	--	--	613.9	925.6

TABLE 8 - Total Page Faults.



9. THIRD SET OF TESTS

The results in this section are averages of ten test runs on sets of 512 random numbers.

Virtual storage was defined as 2048K bytes for all tests except MERGE were it was defined as 4096K bytes.

A working set of 31 pages was used. Approximately 15 pages were used by the programmes and the variables, leaving 16 for the arrays being sorted. Unlike the tests of the previous two sections, however, the programmes and data areas were not locked into real storage and so were eligible for paging operations as well as the array being sorted.

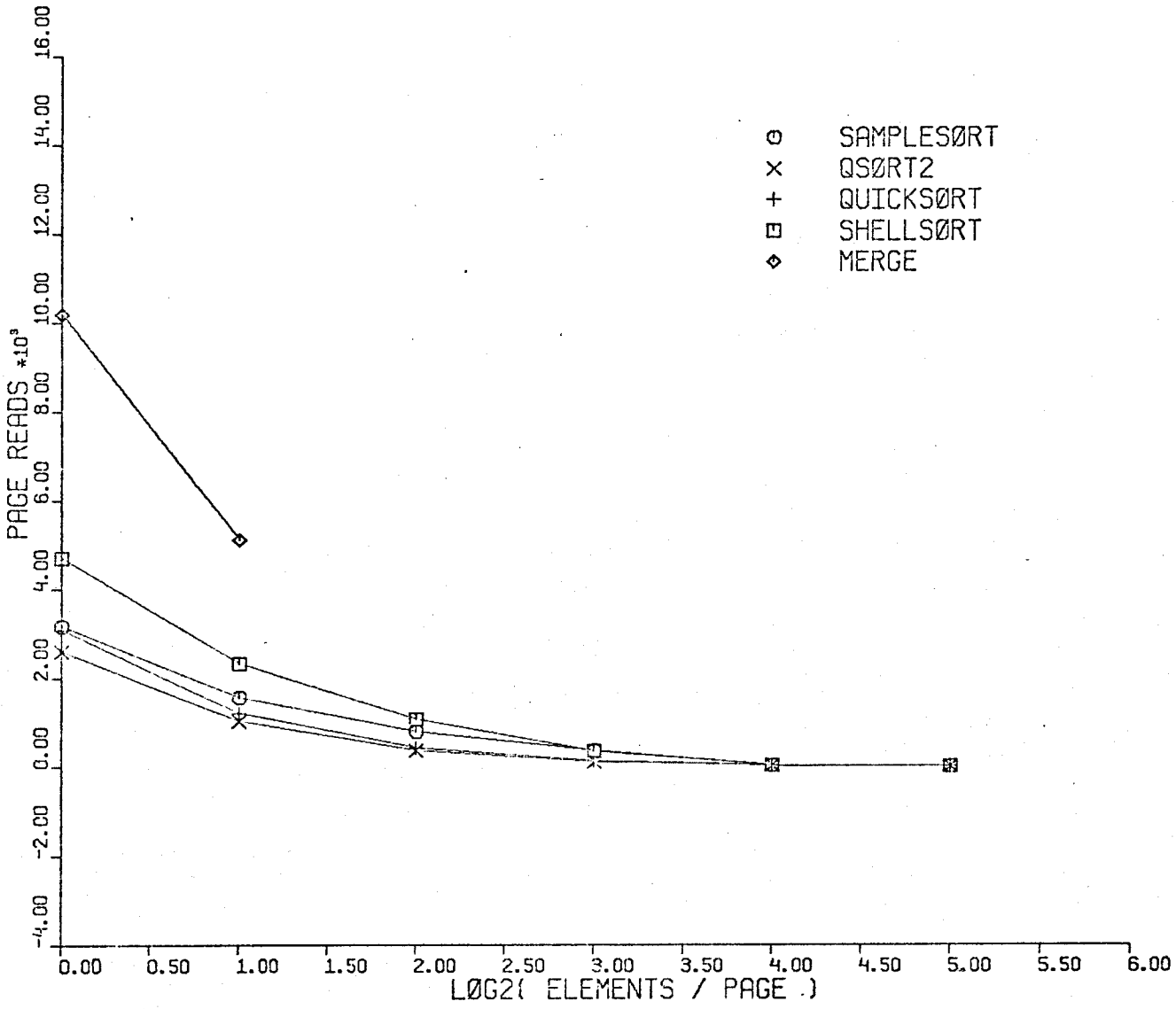
A dummy array was not used, so the array being sorted was not forced to start on a page boundary and could share pages with other data areas and parts of the programme.

This set of tests is intended to be more realistic than the previous two sets as a programme would not normally be locked into main storage on a paging machine. The results, however, are harder to analyze because the code for the programme is never modified. This means that code would sometimes be paged out by the Control Programme (CP) instead of parts of the array which have been modified in order to avoid a page write.

ELEMENTS / PAGE

	32	16	8	4	2	1
Samplesort	0.0	26.6	358.0	784.1	1566.4	3160.4
Qsort2	0.0	17.5	100.4	355.3	1025.2	2598.2
Quicksort	0.0	19.9	124.5	426.4	1202.9	3078.6
Merge	--	--	--	--	5104.7	10182.9
Shellsort	0.0	16.7	344.3	1065.0	2319.6	4694.7

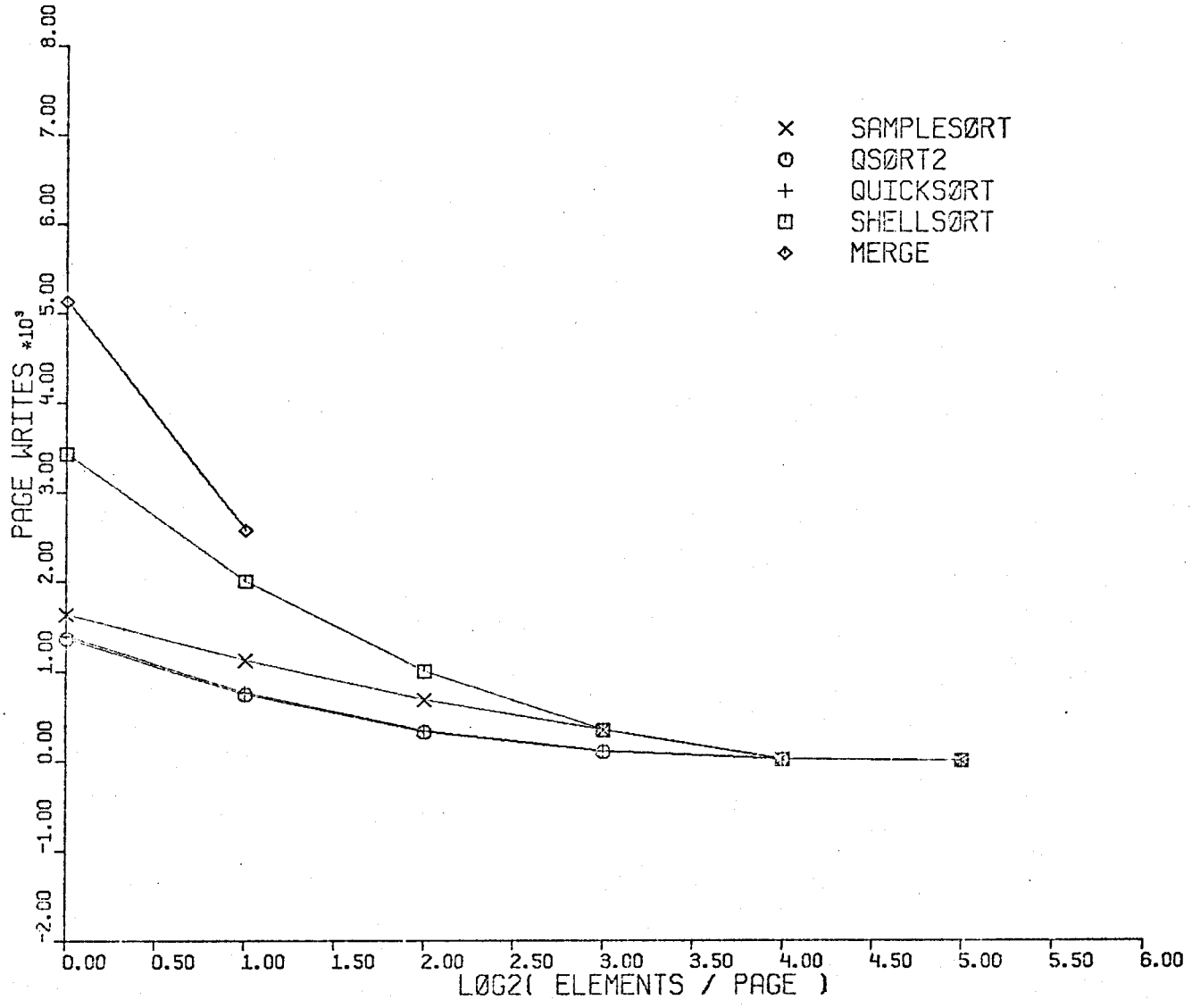
TABLE 9 - Page Reads.



ELEMENTS / PAGE

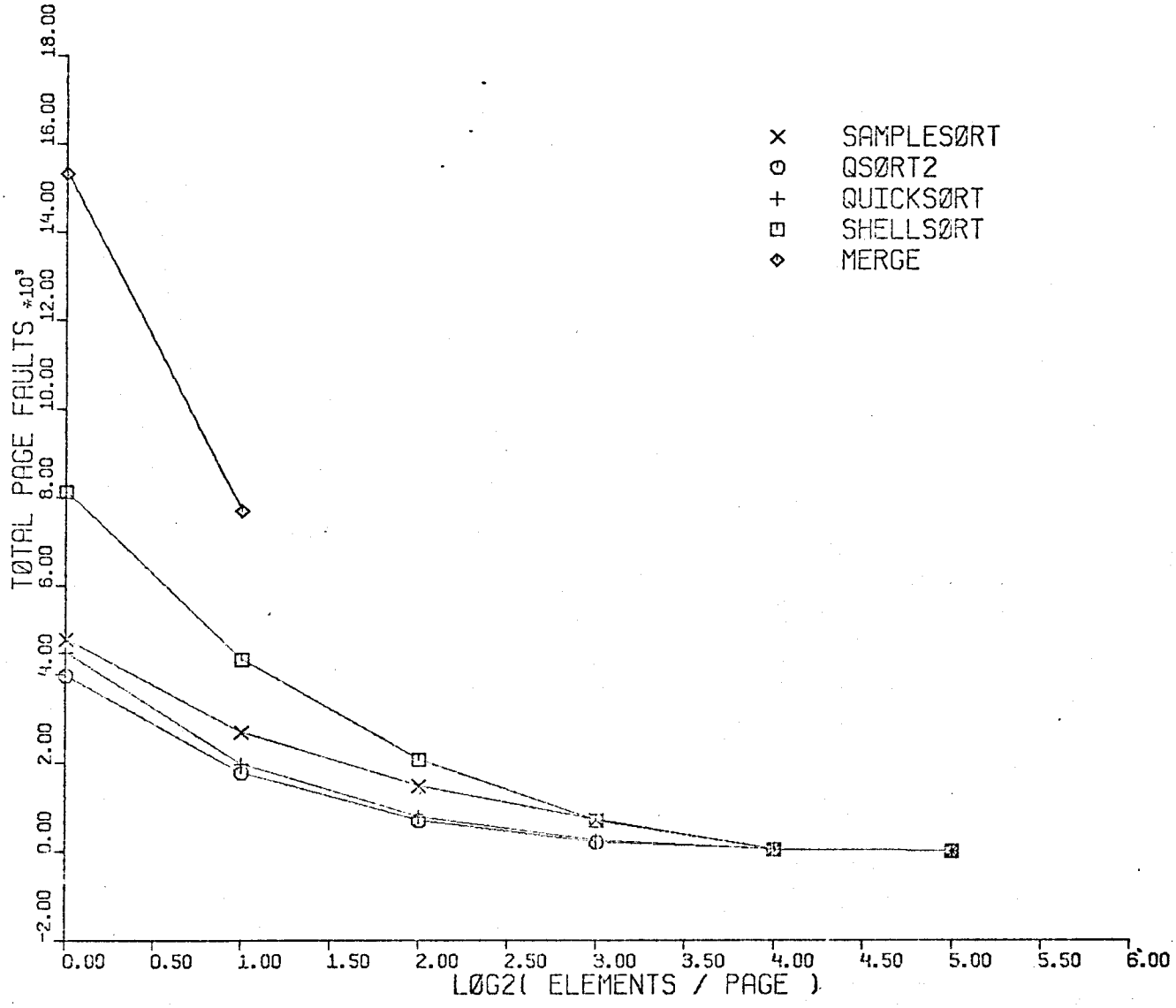
	32	16	8	4	2	1
Samplesort	0.0	21.1	339.9	679.1	1121.0	1627.6
Qsort2	0.0	13.6	97.2	315.3	739.8	1362.8
Quicksort	0.0	16.6	109.9	334.7	755.5	1391.6
Merge	--	--	--	--	2566.9	5132.2
Shellsort	0.0	13.8	342.6	1004.3	1997.4	3434.1

TABLE 10 - Page Writes.



	ELEMENTS / PAGE					
	32	16	8	4	2	1
Samplesort	0.0	47.7	697.9	1463.2	2687.4	4788.0
Qsort2	0.0	31.1	197.6	670.6	1765.0	3961.0
Quicksort	0.0	36.5	234.4	761.1	1958.4	4470.2
Merge	--	--	--	--	7671.6	15315.1
Shellsort	0.0	30.5	686.9	2069.3	4317.0	8128.8

TABLE 11 - Total Page Faults.



10. DISCUSSION OF RESULTS

The figures presented in sections 7, 8, and 9 seem to indicate that a partition sort such as QUICKSORT is the most efficient type of sorting algorithm on a paging machine. This is probably because, as the sort proceeds, the file of numbers being sorted is partitioned into smaller subfiles. Eventually, all the elements of a subfile will fit into the available real storage at the same time and sorting of this subfile can proceed without paging activity.

QSORT2 improves upon QUICKSORT because the partitions are made as close to the middle of the subfiles as possible. This means that small subfiles will be created at an earlier stage in the sort.

By this argument, SAMPLESORT should be still better, but the first phase of SAMPLESORT - sorting the sample, is performed in place. Since the array elements comprising the sample are evenly spread throughout the array, an operation involving two sample elements likely involves two pages also. This extra paging when sorting the sample more than offsets any savings that SAMPLESORT makes by partitioning subfiles more evenly.

MERGE and NATURALMERGE have two characteristics which cause them to perform worse than expected. The first is that they require 2n array locations where n is the number of elements being sorted. This means that the ratio of real

pages to virtual pages used by MERGE and NATURALMERGE is half that of all the other sorts (except MERGESELCT) which is clearly undesirable.

The second problem with these two sorts is that on each pass all the elements being sorted are moved from one of the data areas of n locations to the other, so all 2n array locations and therefore all virtual pages are referenced on every pass. The set of pages in real storage, then, is constantly changing and many paging operations result.

LISTNATURALMERGE avoids almost all the page writes used by the other two merge sorts by constructing a linked list to indicate sorted order. The problem of requiring twice as many virtual pages is also eliminated by using pointers, but all elements are still referenced on every pass.

The set of pages in real storage is probably changing more often in LISTNATURALMERGE than in the other merge sorts because array elements are not necessarily being accessed sequentially as they are in MERGE or NATURALMERGE.

The ratio of virtual pages to real pages is, however, the most important difference as can be seen from Table 5. The page faults for MERGE and NATURALMERGE with 4 elements per page are nearly the same as for LISTNATURALMERGE with 2 elements per page.

LINEARINSERT, LINEARSELECT, and BUBBLESORT all give similar results and are likely inefficient because of the large numbers of comparisons, interchanges, and moves they

perform.

BINARYINSERT is worse yet because a quadratic search is used to find the correct position in which to insert the element in question. This means that, as in LISTNATURALMERGE, array elements and pages are not accessed sequentially and the set of pages in real storage is changing often.

This is also the case for LISTINSERT whose only positive feature is that it performs no page writes.

SHELLSORT performs surprisingly well considering the fact that the same characteristic which makes it efficient in a non-paging environment should make it inefficient in a paging environment. This characteristic is that the subsets being sorted are spread out in a manner similar to the sample in SAMPLESORT and are different on successive passes.

The situation with HEAPSORT is similar to that of SHELLSORT. HEAPSORT, however, performs more comparisons than SHELLSORT and involves approximately the same number of moves (assuming an interchange to be three moves) so, therefore, causes more page faults than SHELLSORT.

MERGESELECT exhibits behaviour quite different from the other sorting algorithms. The number of page reads increases rapidly as the number of elements on each virtual page decreases to 16. After this point, however, there is little increase until the elements per page ratio decreases from two to one.

This behaviour can be explained by the following argument.

MERGESELECT consists of two phases. In the first phase, the input array of n elements is split into n sub-arrays of n elements each and these sub-arrays are sorted individually. In the second phase, the largest element is chosen from among the largest elements of each sub-array. A pointer is updated and the next largest element is chosen.

Now, n was equal to 256 for the tests of Section 7 so each subfile of 16 elements could fit into the 8 pages of real storage until the elements per page ratio dropped below two. This accounts for the change between one and two elements per page.

The second phase of the sort is responsible for the fairly constant paging rate between 16 and 2 elements per page. Since the average distance between elements being considered in the second phase is 16, these elements will all be on seperate real pages until the elements per page ratio increases above 16. The set of pages in real storage will, therefore, be changing with nearly every comparison in the second phase of the sort.

The above intuitive discussion is based on the data collected to date and the author plans a more comprehensive and analytical analysis in the future.

The topics to be dealt with in this future study in-

clude a more detailed analysis of the behaviour of the sorting algorithms such as the order in which array elements are referenced. It is hoped that this will lead to a set of measurable characteristics of algorithms and data base management schemes which can be used to estimate the amount of paging that will be performed when using a given paging algorithm. Locality of reference, for example, appears to play a major role in the efficiency of a programme executed in a paging environment, as does the ratio of virtual pages used to real pages available. Such characteristics could be used to advantage in writing more efficient software for paging machines.

11. BIBLIOGRAPHY

A Guide to the IBM System/370 Model 158 (GC20-1754-1).
IBM Corporation, January 1974.

IBM Virtual Machine Facility/370 : Control Program (CP) Program Logic, Release 2 PLC 4 (SY20-0880-3).
IBM Corporation, March 1974.

IBM Virtual Machine Facility/370 : Conversational Monitor System Program Logic, Release 2 PLC 4 (SY20-0881).
IBM Corporation, March 1974.

IBM Virtual Machine Facility/370 : Service Routines Program Logic, Release 2 PLC 4 (SY20-0882-1).
IBM Corporation, March 1974.

Lecture Notes on Sorting. P. S. Kritzinger.
University of Waterloo, April 1974.

Virtual Machine Facility/370 Features Supplement (GC20-1757-0).
IBM Corporation, January 1974.