

Sorting Algorithms

by [Leif Svalgaard](#)

Sorting is one of the fundamental devices in information processing. It is also one of the most subtle and least understood. The difference between a good and a mediocre sorting algorithm is hard to see when dealing with small amounts of data. However, with large amounts, the difference is so massive that it can totally destroy the effectiveness of a program.

To see why this difference is so important, look at the way that sorting works. We compare the data items with each other, and re-arrange them in order. Simply comparing two values takes a certain amount of time. Moving data around takes significantly longer. The best sorting algorithm does as few compares and moves as possible.

In the worst algorithm (simple bubble sort), each data item is compared to the remaining unsorted items, so that the maximum number of comparisons needed for n items is $n(n-1)$. For ten items, this means around 100 comparisons. For 1000 items, 1,000,000 comparisons. At ten thousand items, the system is no longer responding. The fastest sorting methods approach $kn(\log_2 n)$ comparisons, where k is a small constant depending on the method. If k is 2, this means 46 comparisons for 10 items, 14 thousand comparisons for 1000 items, and 28 million comparisons for 1 million items.

As well as reducing the number of comparisons, good sorting algorithms try to reduce the number of moves. Typically, the data part of an item is much larger than the key part, so moving data around is significantly slower than comparing. A useful way to reduce the costs of moving data is to use indirect tables.

Note that the code required to implement a really efficient quicksort is large and complex compared to bubble sort. The sort algorithm you use depends on the amount and type of data you may need to sort.

In all examples of code below, we assume that the data being sorted is defined as follows:

```

01  TABLE-TO-SORT.
    02  TABLE-SIZE          PIC S9(5)  COMP.
    02  TABLE-MAX          PIC S9(5)  COMP  VALUE +1600.
    02  TABLE-ITEM
        03  TABLE-KEY      PIC X(9).
        03  TABLE-DATA    PIC X(11).
    OCCURS 1600 TIMES
  
```

The size of the key and data simply reflect the data used in the test programs. The table size, 1600 items, makes the table 32000 bytes large. COBOL does not allow tables greater than 32767 bytes (except as a non-portable extension on some systems). A useful device for working with larger tables (memory permitting) is to split the table:

```

01  LARGE-TABLE-ITEM1.
    02  TABLE-KEY          PIC X(9)  OCCURS 2900 TIMES.
  
```

```

01  LARGE-TABLE-ITEM2.
02  TABLE-DATA                PIC X(11) OCCURS 2900 TIMES.

```

1. Bubble Sort

As the name suggests, bubble sort moves items up a table like bubbles in a tube. The algorithm can be explained as follows: pass over the data, comparing and exchanging items so that the largest item ends up at the end of the table. Repeat for the remaining items until the table is sorted, that is, no exchanges were made during the latest pass.

The following code does a bubble sort:

```

01  VARIOUS-INDICES.
02  ITEM-NBR                    PIC S9(5)  COMP.
02  SWAP-NBR                    PIC S9(5)  COMP.
02  JUMP-SIZE                    PIC S9(5)  COMP.
02  UPPER-LIMIT                 PIC S9(5)  COMP.

01  VARIOUS-VALUES.
02  SWAP-ITEM                    PIC X(20).
02  SWAP-INDICATOR              PIC X(1).
08  NO-MORE-SWAPS                VALUE IS SPACE.

BUBBLE-SORT-THE-ARRAY.
  MOVE "SWAP" TO SWAP-INDICATOR
  MOVE 1 TO JUMP-SIZE
  PERFORM BUBBLE-SORT
  UNTIL NO-MORE-SWAPS
  .

BUBBLE-SORT.
  MOVE SPACE TO SWAP-INDICATOR
  COMPUTE UPPER-LIMIT = TABLE-SIZE - JUMP-SIZE

  PERFORM COMPARE-AND-SWAP-KEYS
    VARYING ITEM-NBR FROM 1 BY 1
    UNTIL ITEM-NBR > UPPER-LIMIT
  .

COMPARE-AND-SWAP-KEYS.
  COMPUTE SWAP-NBR = ITEM-NBR + JUMP-SIZE
  IF TABLE-KEY (ITEM-NBR) > TABLE-KEY (SWAP-NBR)
    MOVE TABLE-ITEM (ITEM-NBR) TO SWAP-ITEM
    MOVE TABLE-ITEM (SWAP-NBR) TO TABLE-ITEM (ITEM-NBR)
    MOVE SWAP-ITEM TO TABLE-ITEM (SWAP-NBR)
    MOVE "SWAP" TO SWAP-INDICATOR
  .

```

2. Heapsort

Heapsort uses an intermediate structure, a heap, which is rather like a binary tree, with each son less than or equal to its father. Once the data is partially ordered into a heap, it can be sorted very quickly. The heap is built up within the actual table being sorted, so does not need any extra storage. The detailed workings of heapsort are quite complex, though very efficient.

The advantage of heapsort over bubble sort is simply that it is a lot faster for large amounts of data; $2n(\log_2 n)$ comparisons are needed on average. Additionally, heapsort is quite compact compared to the quicksort algorithm presented later:

```

01  VARIOUS-INDICES.
    02  ITEM-NBR          PIC S9(5)  COMP.
    02  COPY-PTR         PIC S9(5)  COMP.
    02  LEFT-PTR         PIC S9(5)  COMP.
    02  RIGHT-PTR        PIC S9(5)  COMP.
    02  TO-PTR           PIC S9(5)  COMP.
    02  FROM-PTR         PIC S9(5)  COMP.

01  VARIOUS-VALUES.
    02  EXCHANGE-ITEM.
        03  EXCHANGE-KEY  PIC X(9).
        03  EXCHANGE-DATA PIC X(11).

HEAP-SORT-THE-TABLE.
    COMPUTE LEFT-PTR  = TABLE-SIZE / 2 + 1
    COMPUTE RIGHT-PTR = TABLE-SIZE

    PERFORM SIFT-THE-HEAP-BY-LEFT
        UNTIL LEFT-PTR < 2

    PERFORM SIFT-THE-HEAP-BY-RIGHT
        UNTIL RIGHT-PTR < 2
    .

SIFT-THE-HEAP-BY-LEFT.
    SUBTRACT 1 FROM LEFT-PTR
    PERFORM SIFT-THE-HEAP
    .

SIFT-THE-HEAP.
    COMPUTE TO-PTR  = LEFT-PTR
    COMPUTE FROM-PTR = LEFT-PTR + LEFT-PTR
    MOVE TABLE-ITEM (TO-PTR) TO EXCHANGE-ITEM
    PERFORM BUILD-ITEMS-INTO-HEAP
        UNTIL FROM-PTR > RIGHT-PTR

    MOVE EXCHANGE-ITEM TO TABLE-ITEM (TO-PTR)
    .

BUILD-ITEMS-INTO-HEAP.
    IF FROM-PTR < RIGHT-PTR
        COMPUTE COPY-PTR = FROM-PTR + 1
        IF TABLE-KEY (FROM-PTR) < TABLE-KEY (COPY-PTR)
            MOVE COPY-PTR TO FROM-PTR
        .
    IF EXCHANGE-KEY NOT < TABLE-KEY (FROM-PTR)
        COMPUTE FROM-PTR = RIGHT-PTR + 1
    ELSE
        MOVE TABLE-ITEM (FROM-PTR) TO TABLE-ITEM (TO-PTR)
        MOVE          FROM-PTR      TO          TO-PTR
        COMPUTE FROM-PTR = TO-PTR + TO-PTR
    .

SIFT-THE-HEAP-BY-RIGHT.
    MOVE TABLE-ITEM (1)          TO EXCHANGE-ITEM
    MOVE TABLE-ITEM (RIGHT-PTR) TO TABLE-ITEM (1)
    MOVE EXCHANGE-ITEM          TO TABLE-ITEM (RIGHT-PTR)
    SUBTRACT 1 FROM RIGHT-PTR
    PERFORM SIFT-THE-HEAP
    .

```

3. Quicksort

Sorting is a problem which has been solved. The fastest sort algorithm is quicksort. If

you ever need to sort significant amounts of data, use this method. There are some notable aspects of quicksort that you should be aware of, even if the detailed mechanism is not important:

- Quicksort is a recursive procedure. This is easiest to implement in a language such as Pascal or C, but quite possible in COBOL, as we shall see.
- Quicksort is unstable. This means that two records with the same key may not end up in the same order after sorting. Heapsort and Combsort are also unstable. Bubble sort is, in contrast, stable (its one good feature). You can add stability to the sort by extending the key to include a sequence number, so that there are in fact no duplicate keys.
- Quicksort performs badly once the amounts of data become small due to the overhead of recursion. This can be avoided by using a secondary sort when the partition size is less than some magic figure.
- Quicksort performs badly with certain types of data; this can be improved by judicious choice of a pivot point at each sort, f. ex. the middle point.

The following highly-optimised implementation is a good example of [pseudo-recursive](#) programming in COBOL. It uses an insertion sort to handle small partitions:

```

01  SORT-HANDLING.
    02  CUR-LOWER-LIMIT          PIC S9(5)  COMP.
    02  CUR-UPPER-LIMIT          PIC S9(5)  COMP.
    02  PARTITION-SIZE           PIC S9(5)  COMP.
    02  MINIMUM-PARTITION        PIC S9(5)  COMP  VALUE +13.
    02  LOWER-PTR                PIC S9(5)  COMP.
    02  MIDDLE-PTR               PIC S9(5)  COMP.
    02  UPPER-PTR                PIC S9(5)  COMP.
    02  PREV-PTR                 PIC S9(5)  COMP.

    02  STACK-TOP                PIC S9(5)  COMP  VALUE +20.
    02  STACK-PTR                PIC S9(5)  COMP.
    02  STACK-NXT                PIC S9(5)  COMP.
    02  QUICKSORT-STACK          OCCURS 20.
        03  LOWER-LIMIT          PIC S9(5)  COMP.
        03  UPPER-LIMIT          PIC S9(5)  COMP.

    02  PIV-ITEM.
        03  PIV-KEY               PIC X(9).
        03  PIV-DATA              PIC X(11).
    02  COMP-ITEM.
        03  COMP-KEY              PIC X(9).
        03  COMP-DATA             PIC X(11).

    02  EXCHG-ITEM               PIC X(20).

    02  COMPARE-RESULT            PIC X.
        88  COMPARE-GREATER-THAN  VALUE "G".
        88  COMPARE-LESS-THAN    VALUE "L".

QUICKSORT-THE-TABLE.
    MOVE 1 TO STACK-PTR, LOWER-LIMIT (STACK-PTR)
    MOVE TABLE-SIZE TO UPPER-LIMIT (STACK-PTR)

    PERFORM QUICKSORT-THE-ENTRIES
        UNTIL STACK-PTR = ZERO

    PERFORM INSERTION-SORT
        VARYING UPPER-PTR FROM 2 BY 1
        UNTIL UPPER-PTR > TABLE-SIZE
    .

```

```

QUICKSORT-THE-ENTRIES.
  MOVE LOWER-LIMIT (STACK-PTR) TO CUR-LOWER-LIMIT
  MOVE UPPER-LIMIT (STACK-PTR) TO CUR-UPPER-LIMIT
  IF CUR-LOWER-LIMIT < CUR-UPPER-LIMIT
    MOVE CUR-LOWER-LIMIT TO LOWER-PTR
    MOVE CUR-UPPER-LIMIT TO UPPER-PTR

    COMPUTE PARTITION-SIZE = UPPER-PTR - LOWER-PTR + 1
    IF PARTITION-SIZE > MINIMUM-PARTITION
      PERFORM FIND-A-PIVOT-ENTRY
      PERFORM PARTITION-ENTRIES
      UNTIL UPPER-PTR < LOWER-PTR

      PERFORM QUICKSORT-THE-PARTITIONS
      PERFORM INCREASE-AND-CHECK-STACK-PTR
    ELSE
      SUBTRACT 1 FROM STACK-PTR
  ELSE
    SUBTRACT 1 FROM STACK-PTR
  .

INCREASE-AND-CHECK-STACK-PTR.
  IF STACK-PTR < STACK-TOP
    ADD 1 TO STACK-PTR
  ELSE
    MOVE ZERO TO STACK-PTR
  .

FIND-A-PIVOT-ENTRY.
  COMPUTE MIDDLE-PTR = (LOWER-PTR + UPPER-PTR) / 2
  MOVE TABLE-ITEM (MIDDLE-PTR) TO PIV-ITEM
  .

PARTITION-ENTRIES.
  MOVE "LT" TO COMPARE-RESULT
  PERFORM INCREASE-LOWER
  UNTIL COMPARE-GREATER-THAN

  MOVE "GT" TO COMPARE-RESULT
  PERFORM DECREASE-UPPER
  UNTIL COMPARE-LESS-THAN

  IF LOWER-PTR NOT > UPPER-PTR
    PERFORM EXCHANGE-UPPER-AND-LOWER
    ADD 1 TO LOWER-PTR
    SUBTRACT 1 FROM UPPER-PTR
  .

INCREASE-LOWER.
  MOVE TABLE-ITEM (LOWER-PTR) TO COMP-ITEM
  IF COMP-KEY < PIV-KEY
    ADD 1 TO LOWER-PTR
  ELSE
    MOVE "GT" TO COMPARE-RESULT
  .

DECREASE-UPPER.
  MOVE TABLE-ITEM (UPPER-PTR) TO COMP-ITEM
  IF COMP-KEY > PIV-KEY
    SUBTRACT 1 FROM UPPER-PTR
  ELSE
    MOVE "LT" TO COMPARE-RESULT
  .

```

```

EXCHANGE-UPPER-AND-LOWER.
  MOVE TABLE-ITEM (LOWER-PTR) TO EXCHG-ITEM
  MOVE TABLE-ITEM (UPPER-PTR) TO TABLE-ITEM (LOWER-PTR)
  MOVE EXCHG-ITEM                TO TABLE-ITEM (UPPER-PTR)
  .

QUICKSORT-THE-PARTITIONS.
  COMPUTE STACK-NXT = STACK-PTR + 1
  IF UPPER-PTR - CUR-LOWER-LIMIT < CUR-UPPER-LIMIT - LOWER-PTR
    MOVE CUR-LOWER-LIMIT TO LOWER-LIMIT (STACK-NXT)
    MOVE    UPPER-PTR    TO UPPER-LIMIT (STACK-NXT)

    MOVE    LOWER-PTR    TO LOWER-LIMIT (STACK-PTR)
    MOVE CUR-UPPER-LIMIT TO UPPER-LIMIT (STACK-PTR)
  ELSE
    MOVE    LOWER-PTR    TO LOWER-LIMIT (STACK-NXT)
    MOVE CUR-UPPER-LIMIT TO UPPER-LIMIT (STACK-NXT)

    MOVE CUR-LOWER-LIMIT TO LOWER-LIMIT (STACK-PTR)
    MOVE    UPPER-PTR    TO UPPER-LIMIT (STACK-PTR)
  .

INSERTION-SORT.
  MOVE "GT" TO COMPARE-RESULT
  PERFORM INSERT-IN-PLACE
  VARYING LOWER-PTR FROM UPPER-PTR BY -1
  UNTIL NOT COMPARE-GREATER-THAN
  .

INSERT-IN-PLACE.
  COMPUTE PREV-PTR = LOWER-PTR - 1
  IF PREV-PTR < 1
    MOVE "LT" TO COMPARE-RESULT
  ELSE
    MOVE TABLE-ITEM (PREV-PTR) TO COMP-ITEM
    MOVE TABLE-ITEM (LOWER-PTR) TO PIV-ITEM
    IF COMP-KEY > PIV-KEY
      MOVE PIV-ITEM TO TABLE-ITEM (PREV-PTR)
      MOVE COMP-ITEM TO TABLE-ITEM (LOWER-PTR)
    ELSE
      MOVE "LT" TO COMPARE-RESULT
  .

```

4. Combsort

Just as we thought that the last word had been said about sorting, a breakthrough comes along and spoils everything. In the April 1991 issue of BYTE magazine, Stephen Lacey and Richard Box show that a simple modification to bubble sort makes it a fast and efficient sort method on par with heapsort and quicksort.

In a bubble sort, each item is compared to the next; if the two are out of order, they are swapped. This method is slow because it is susceptible to the appearance of what Box and Lacey call turtles. A turtle is a relatively low value located near the end of the table. During a bubble sort, this element moves only one position for each pass, so a single turtle can cause maximal slowing. Almost every long table of items contains a turtle.

Their simple modification of bubble sort which they call 'combsort' eliminates turtles quickly by allowing the distance between compared items to be greater than one. This distance - the JUMP-SIZE - is initially set to the TABLE-SIZE. Before each pass, the JUMP-SIZE is divided by 1.3 (the shrink factor). If this causes it to become less than 1, it is simply set to 1, collapsing combsort into bubble sort. An exchange of items moves

items by JUMP-SIZE positions rather than only one position, causing turtles to jump rather than crawl. As with any sort method where the displacement of an element can be larger than one position, combsort is not stable - like elements do not keep their relative positions. This is rarely a problem in practice.

Successively shrinking the JUMP-SIZE is analogous to combing long, tangled hair - stroking first with your fingers alone, then with a pick comb that has widely spaced teeth, followed by finer combs with progressively closer teeth - hence the name comb sort. Lacey and Box came up with a shrink factor of 1.3 empirically by testing combsort on over 200,000 random tables. There is at present no theoretical justification for this particular value; it just works...

Here is then the magic code (re-using [declarations](#) and a [code paragraph](#) from bubble sort). It is clearly correct, as it (unless the table is empty) ends with JUMP-SIZE = 1 (ensured by the '+3') and therefore degenerates into bubble sort:

```

COMB-SORT-THE-ARRAY.
  MOVE TABLE-SIZE TO JUMP-SIZE
  PERFORM COMB-SORT
    UNTIL NO-MORE-SWAPS
      AND JUMP-SIZE NOT > 1
  .

COMB-SORT.
  COMPUTE JUMP-SIZE = (10 * JUMP-SIZE + 3) / 13
  COMPUTE UPPER-LIMIT = TABLE-SIZE - JUMP-SIZE
  MOVE SPACE TO SWAP-INDICATOR
  PERFORM COMPARE-AND-SWAP-KEYS
    VARYING ITEM-NBR FROM 1 BY 1
    UNTIL ITEM-NBR > UPPER-LIMIT
  .

```

The careful termination test (JUMP-SIZE NOT > 1) also caters for the case where the table is empty.

5. Comparison of Table Sorting Methods

The above implementations of bubble sort, heapsort, quicksort, and combsort were tested on from 200 to 1600 items, each consisting of a 9-character key and 11 characters of data. The times below are measured in 1/1000ths of a second and are the averages of sorting four different random sequences. The test program was run on a PC (16 MHz 386SX) and on an AS/400 (B35). After the tables had been sorted, they were sorted again to measure the time for sorting already sorted data.

a. Results for Bubble Sort

Items:	Order:	Time PC:	Time AS/400:
200	random	1072	5490
	sorted	9	36
400	random	4372	24126
	sorted	16	62
800	random	17875	100725
	sorted	28	113
1600	random	71390	401785
	sorted	55	213

Bubble sort is very fast on already sorted data, but very slow on random data. As the number of data elements grows, bubble sort slows to a crawl and becomes useless.

b. Results for Heapsort

Items:	Order:	Time PC:	Time AS/400:
200	random	82	342
	sorted	68	362
400	random	151	770
	sorted	165	813
800	random	330	1734
	sorted	357	1829
1600	random	783	3857
	sorted	811	4046

Heapsort is of medium complexity and does a very good job. On already sorted data it slows down a trifle.

c. Results for Quicksort

Items:	Order:	Time PC:	Time AS/400:
200	random	68	275
	sorted	20	127
400	random	165	650
	sorted	68	273
800	random	330	1410
	sorted	151	613
1600	random	715	2974
	sorted	316	1374

Quicksort is of high complexity and owes its performance on sorted data to the built-in insertion-sort.

d. Results for Combsort

Items:	Order:	Time PC:	Time AS/400:
200	random	110	453
	sorted	55	326
400	random	192	1103
	sorted	151	814
800	random	467	2621
	sorted	316	1826
1600	random	1076	6299
	sorted	715	4254

Combsort is just a single, easy line more than bubble sort, but performs spectacularly; in fact, almost as well as the grotesquely more complex quicksort. Because of its magic

simplicity, this is the method we recommend - except for the most demanding applications where the 50 percent improvement quicksort offers may be important.

6. Pseudo-sorting

Sometimes it is not necessary to do a full sort of the array. The classical case is that of finding the median, ie. that value for which half of the array items are greater and the other half smaller.

The following very fast algorithm is due to C.A.R. Hoare (CACM, Jan. 1971). The program finds the element of the array whose value is f th in order of magnitude; and rearranges the array such that this element is placed at index f ; and furthermore that all elements with subscripts lower than f have lesser values, and all elements with subscripts greater than f have greater values.

To find the median, set $f = (1 + \text{ARRAY-SIZE})/2$; to find the first quartile, set $f = (1 + \text{ARRAY-SIZE})/4$, etc.:

```

FIND-THE-MEDIAN.
  MOVE      1      TO LOWER-BOUND
  MOVE ARRAY-SIZE TO UPPER-BOUND
  COMPUTE MEDIAN-INDEX = (1 + ARRAY-SIZE) / 2

  PERFORM REDUCE-MIDDLE-PART
  UNTIL LOWER-BOUND NOT < UPPER-BOUND
  .

REDUCE-MIDDLE-PART.
  MOVE ARRAY-KEY (MEDIAN-INDEX) TO MEDIAN-KEY
  MOVE LOWER-BOUND TO LOWER-INDEX
  MOVE UPPER-BOUND TO UPPER-INDEX
  PERFORM NARROW-THE-RANGE
  UNTIL LOWER-INDEX > UPPER-INDEX

  IF MEDIAN-INDEX NOT > UPPER-INDEX
    MOVE UPPER-INDEX TO UPPER-BOUND
  ELSE
    IF MEDIAN-INDEX NOT < LOWER-INDEX
      MOVE LOWER-INDEX TO LOWER-BOUND
    ELSE
      MOVE LOWER-BOUND TO UPPER-BOUND
  .

NARROW-THE-RANGE.
  PERFORM INCREASE-LOWER-INDEX
  UNTIL ARRAY-KEY (LOWER-INDEX) NOT < MEDIAN-KEY

  PERFORM DECREASE-UPPER-INDEX
  UNTIL ARRAY-KEY (UPPER-INDEX) NOT > MEDIAN-KEY

  IF LOWER-INDEX NOT > UPPER-INDEX
    MOVE ARRAY-ITEM (LOWER-INDEX) TO SWAP-ITEM
    MOVE ARRAY-ITEM (UPPER-INDEX) TO ARRAY-ITEM (LOWER-INDEX)
    MOVE SWAP-ITEM TO ARRAY-ITEM (UPPER-INDEX)
    PERFORM INCREASE-LOWER-INDEX
    PERFORM DECREASE-UPPER-INDEX
  .

INCREASE-LOWER-INDEX.
  ADD 1 TO LOWER-INDEX
  .

```

```

DECREASE-UPPER-INDEX.
  SUBTRACT 1 FROM UPPER-INDEX
  .

```

The algorithm uses the [standard](#) array declarations of this paper with the following additional data declarations:

```

01  FIND-HANDLING.
    02  LOWER-BOUND          PIC S9(5)  COMP.
    02  UPPER-BOUND          PIC S9(5)  COMP.
    02  MEDIAN-INDEX         PIC S9(5)  COMP.
    02  LOWER-INDEX          PIC S9(5)  COMP.
    02  UPPER-INDEX          PIC S9(5)  COMP.

```

The Author

[Leif Svalgaard](#) has been a software developer for the past 35 years. Leif's software development projects include extensive work with system level APIs (application programming interfaces) for a great many systems including AS/400, Unix, and PC; implementation of large networking systems used by several telephone companies; and as co-developer of the historically important RC4000 multiprogramming Operating System.

From 1999 - 2000 he served as Senior Developer at PentaSafe Security Technologies, Inc., (Houston, TX) developing security-related software for the IBM AS/400.

Leif served as the Director of Development from 1994 - 1998 at T.O.S.C. International, Inc., (Houston, TX) with responsibility for developing and maintaining the ETK (Easy ToolKit) product. ETK is a large (1,000,000 lines) application written in Cobol (writing over half the code himself) that generates Cobol code that will run on any platform that has a Cobol compiler.

At Stanford University, from 1972 through 1978 Leif worked both as a computer software specialist and as a space physics scientist, participating in the design and implementation of the operating system and real-time control software for the Wilcox Solar Observatory (then the Stanford Solar Observatory) telescope.

Other areas in which Leif has extensive experience include PC Assembler, Unix C-programming and operating system interfaces, AS/400 COBOL and MI (assembler), IBM/MVS/CICS Assembler and CICS command/macro interface, VAX/VMS System Internals and assembler language, Data General System interfacing and assembler language, HP-3000 System interfacing, and Bull GCOS7/8 System interfacing and integration. Leif recently developed a GUI interface for legacy screens using the AWT for Java.

This website is copyrighted by Leif Svalgaard.
Last Updated May 6, 2001.
