Auteur :	GUIDAL (Philippe)
Titre :	Quicksort
Lieu :	Nottingham
Éditeur :	*Ipso Facto*
Date :	October 1995 (vol. IX, n° 7, pp. 55-56)
Dewey :	005.741 GUI
Classe :	Méthodes d'accès et organisation des fichiers (adressage calculé, algorithmes de recherche ou de tri, arborescences de recherche, formats de fichiers de données, regroupement, tri)
Observations :	La pagination d'*Ipso Facto* a été reportée en rouge dans le texte, entre crochets. La deuxième partie de cet article est restée inédite.

[55]

QUICKSORT: ANALYSIS

The present subject will be first discussed using assembly language. Readers not familiar with it should consult *Machine code programming on the Psion Organiser* by Bill AITKEN[1] (despite some bugs which may be found in this book). Later on, we'll study OPL implementation.

PRELIMINARIES

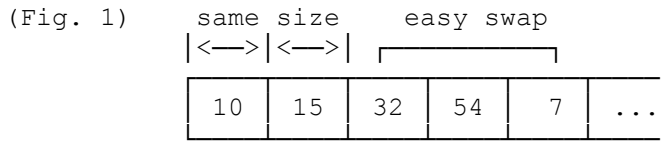
What's the sort about? Well, just imagine the phone book unsorted… Sorting is a very important matter in data processing. Because of a better readability, access to data is made easier, particularly when it comes to make a selection of data. For instance, go to your phone book: you want to call Mike O'REGAN and complain about this silly article. Assuming that the phone book is sorted, you just have to know where is the "O" block, then find the "O'R" sub-block inside, and so on, without reading the book from the first to the last page.

The problem of sorting has often been discussed for long and many good books are covering the subject (some references will be given later). There are several algorithms to perform sorting. Some are easy (*i.e.* simple to code and understand), some are more complicated. Depending on the context, some are efficient, some are less. They are all comparison-based algorithms, that is to say the basic operation on data will be performed with "<" and ">" operators, the result of comparison being used to place data in the required order by successive swaps.

[1] Kuma Publishing Ltd, 1990.

The efficiency of a sorting algorithm largely depends on the number and the type of data to be sorted. Whatever the sorting criterion is, sorting five items is obviously faster and easier than sorting a million. On the other hand, consider an array of integers, each of them being coded on two bytes (MSB and LSB): sorting can be performed "in place", since each element of the array can be swapped with any other, without any extra memory requirement:

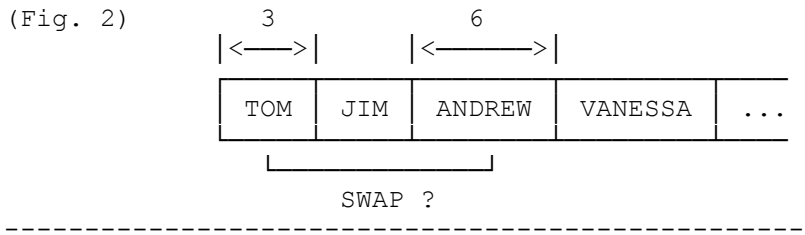
(Fig. 1)



Consider now an array of floating point values. In the Organiser, these values are coded on eight bytes (6-byte mantissa, exponent and sign), although an extra ninth carry byte may be used too in some particular cases. As for integers, all the values have the same length and swaps remain easy. But the comparison becomes more tricky… It is simple to know whether an integer is lower or greater than another: check the sign first (assuming that -1<+1), then if signs are the same, check the numbers (assuming that -2<-1 and 1<2). When it comes to floating point values, a third comparison is required for exponents.

A worse case is with strings of characters. According to ASCII code, the Organiser assumes that "Z"<"a" or that "JIM"<"andrew"… Should this always be? Furthermore, strings have, most often, different lengths, which is another source of problems when swapping must be done:

(Fig. 2)



Even worse is the mix case of common database records including string fields (such as name, address,…) and numeric fields (such as registration numbers, amount of orders,…), and multi-criterion sort to be performed…

STRATEGY

A first element of a good strategy is to deal with data as simply as possible. Instead of swapping actual data, tags will be used. A tag may be a record

number, the address of data (or a pointer to this address), or any code [56] which fits the purpose of the sort. This technique is known as INDIRECT SORTING.

A second improvement is to leave the comparison task to the user, through a user-supplied routine which will be called when required and will return the result of comparison to the main sorting routine. A bit of laziness? Maybe, but it also provides the user with the control of the whole process, allowing any trick, such as stating that "andrew"<"JIM"… or dealing with integers in range 0 to 65535 instead of usual range -32768 to +32767!

If you're familiar with the binary system, you'll quickly understand this example:
When the integer 25432 comes to be compared with -20524, the Organiser assumes that -20524<25432. If it's up to you to do the comparison, you may decide that -20524 is actually the unsigned integer 45012, and therefore invert the result of comparison by stating that 45012>25432 (that is, from Organiser's point of view, -20524>25432)!

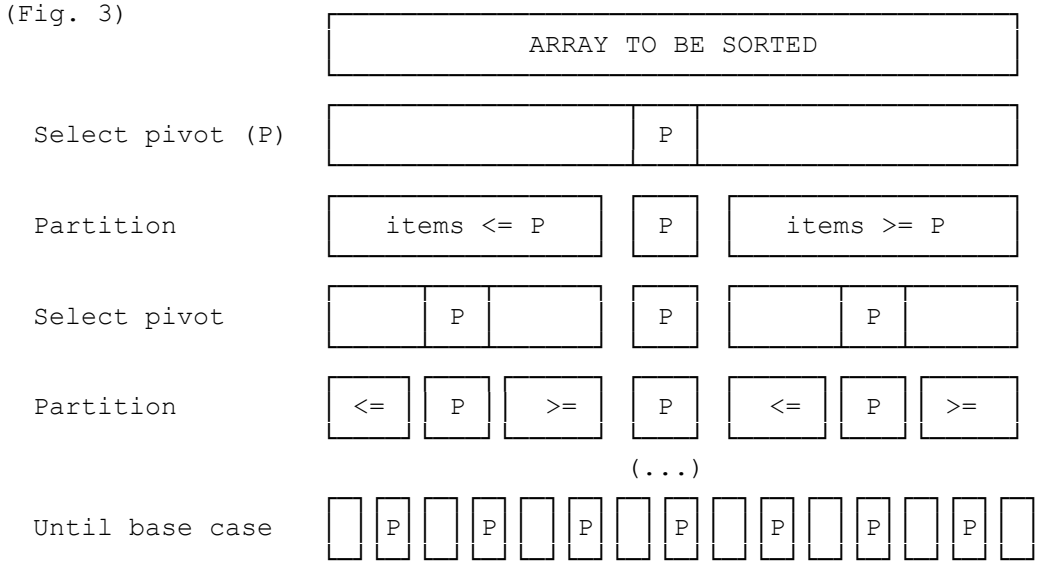
Anyway, those two elements (tags and user-supplied routine) were adopted by PSION to build up a general-purpose sorting utility which therefore provides only a basis for sorting. As mentioned earlier, there are several sorting algorithms. The fastest (when properly implemented) is known as QUICKSORT. It was invented, *ca.* 1962, by C.A.R. HOARE, and was chosen by PSION. Let's see that.

QUICKSORT (at last)

The Quicksort algorithm is based on the "divide-and-conquer" strategy (that politicians do know very well): dividing a problem into smaller problems which at their turn are divided into smaller ones, etc., until a BASE CASE is reached, which can be solved without any other division. Base case is an important notion to be kept in mind in order to avoid infinite loops. Then the conquering phase may start, by patching together the previous computations.

The basic Quicksort algorithm is as follows:
1/ Choose one item in the array to be sorted. This item is called the PIVOT;
2/ Divide the array in two smaller arrays by placing the smaller (relatively to the pivot) items in the left-hand sub-array, and the larger (relatively to the pivot) in the right-hand. This phase is called PARTITION;
3/ Loop to 1/ until each sub-array contains only one item (base case).

(Fig. 3)



QUICKSORT is a classical example of what is called a RECURSIVE algorithm. If you've never heard of recursion: basically, a procedure is said to be recursive when it calls itself, such as the following:

```
INFINITE:(x)
RETURN INFINITE:(x+1)
```

Don't ever try to implement this procedure! As its name implies, it runs an infinite loop (almost infinite in fact, as the "OUT OF MEMORY" message would be soon displayed on screen!). The Quicksort algorithm could be expressed as follows:

```
QSORT:(left,right,array())          \left,right = bound of array()
IF left<right
      PART:(left,right,array())     \pick pivot and partition array()
      QSORT:(left,pivot-1,array())
      QSORT:(pivot+1,right,array())
ENDIF
```

(to be continued)

[Fin de la première partie; la suite est restée inédite]

Recursion is mainly used when defining an algorithm: it allows the algorithm to be clearly stated in a few lines. When it comes to run an actual program, recursion often happens to be a source of problems.
For some high-level languages, such as OPL, the program is copied in memory each time it is called. It means that if a recursive procedure calls itself 1000 times (which could be the case when sorting), the procedure will be copied

1000 times in memory. Suppose the length of code of the procedure is 33 bytes, there would be about 32k used for the whole process to be done. In a little machine such as the Organiser, with little memory size, it would soon turn out to be a disaster.
For some other languages (just like assembly language or machine code), there's no code copied, but return addresses associated to recursive calls (BSR statements, for instance) are stored in a special area of the memory, the stack. If 1000 successive return addresses are stacked, each of them being coded on two bytes, there must be enough stack space to hold those 2000 bytes, and the Organiser's stack will not handle such an amount of data.
Otherwise, non-recursive programs are proven to be, most often, faster than recursive ones, but this will not be discussed here.
The solution is to simulate recursion by using a loop (DO/UNTIL, WHILE/WEND) or even a simple "IF condition GOTO label:: ENDIF", which doesn't require any extra memory.
The resulting code is not as simple and clear as the source algorithm, but it can be run safely.

GENERAL DESIGN

To study how PSION Quicksort is actually implemented, we'll take a practical example as it is given in Bill AITKEN's *Machine code programming*: sorting the "QWERTY" keyboard alphabet into standard order.
The OS interrupt n° 165 is called with the following input:

- the D register holds the number of items to be sorted;
- the IX register holds the address of the user-supplied routine.

Then the calling routine will be:

```
-----------------------------------------------------------------------
START:      LDD #26           \Number of items
            LDX #MYROUTINE    \Address of user-supplied routine
            OS  UT$SORT       \SWI #165
            BCC OKRTS         \RTS if no error
            OS  ER$MESS       \SWI #32
OKRTS:      RTS               \All done
-----------------------------------------------------------------------
```

As a matter of fact, UT$SORT is the "driver" for Quicksort:

```
-----------------------------------------------------------------------
{Program UT$SORT part 1}

START:      STX US_ROUTINE    \Store input
            STD ITEMS
{...}
-----------------------------------------------------------------------
```

The first step, according to the strategy chosen, is to create a cell which will hold a tag for each of the items to be sorted:

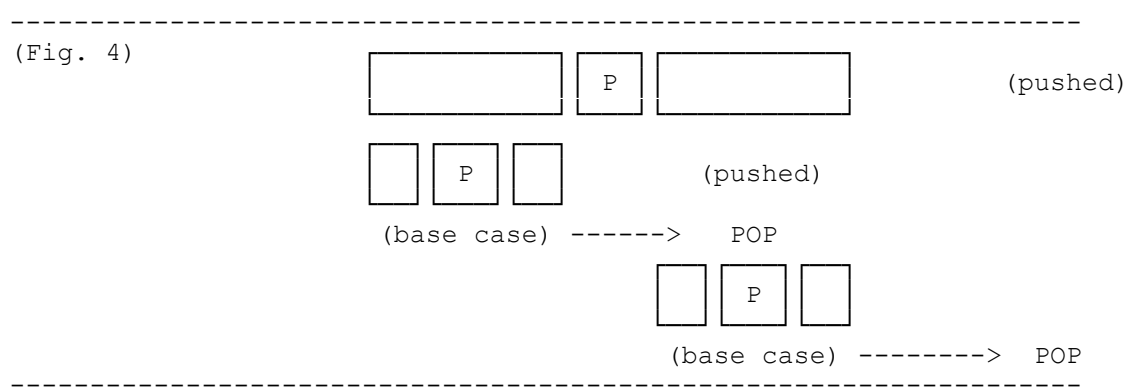
```
-----------------------------------------------------------------------
{Program UT$SORT part 2}

INIT_CELL:  ASL D             \size of cell = 2*(number of items)
            SWI #1            \create sort cell
            BCS END           \RTS if error
            STX SORT_CELL     \Save tag of sort cell
{...}
-----------------------------------------------------------------------
```

As you see, PSION decided that the size of the sort cell would be 2*(number of items to be sorted). Thus each tag will be stored in two bytes, and addresses of items will fit in perfectly.

Second step: obtain the tag for each of the items, by calling the user-supplied routine, and store the tags in the sort cell. The B register will be used as a flag, indicating what is required. Here, the B register holds 0, indicating that the IX register contains a "running number" in range 0 to n-1, n being the number of items to be sorted. As often, the range starts to 0 for easier computation; but don't forget that item n° 0 is actually the first item, item n° 1 being the second, etc. The running number is passed to the user-supplied routine; it's then up to the user to decide what to do with it. In the example provided by PSION in Bill AITKEN's book, the user-supplied routine returns the address of each item. Suppose that the address of the string "QWERTY…" is 1000, this start address is added to the contents of the IX register in order to get the address of each individual element (letter) of the string to be sorted: when IX holds 0 (input tag of the first item), adding 1000 gives 1000, which is the address of the first letter ("Q"); when IX holds 1, adding 1000 gives 1001, which is the address of the second letter ("W"), etc.
If you want to deal with record numbers only, you should leave the contents of IX unchanged, and it will be returned to Quicksort utility as it was before.

```
-------------------------------------------------------------------
{Program UT$SORT part 3}

            LDX #0              \initialize running number
INIT_LOOP:  PSH X               \save it twice
            PSH X
            CLR B               \clear flag (B=0)
            JSR CALL_USER       \get tag
            PUL A               \restore running number
            PUL B
            JSR SET_POINTER     \set pointer accordingly within sort cell
            STD 0,X             \store tag (address of item, for instance)
            PUL X               \restore running number
            INX                 \next one
            CPX ITEMS           \check range
            BCS INIT_LOOP       \loop while tags still to be obtained
{...}
-------------------------------------------------------------------
{User-supplied routine part 1}

MYROUTINE:  TST B               \check flag
            BNE 1$              \branch if not 0
            XGDX                \transfer running number in D register
            ADD D #ORIGINAL     \add start address of string to be sorted
            XGDX                \transfer result in IX register
            RTS                 \all done
{...}
-------------------------------------------------------------------
{String to be sorted}

ORIGINAL: ascii "QWERTYUIOPASDFGHJKLZXCVBNM"
-------------------------------------------------------------------
```

Then comes a little subtility: after each partitioning, only one of the two resulting sub-arrays will be quicksorted again until the base case is reached, as the CPU can’t perform several parallel quicksorts at a time. What about the remaining sub-array(s)? Well, their bounds (LEFT and RIGHT) are pushed onto a temporary stack after each partitioning; then, when a base case is reached, the program retrieves the last remaining sub-array by popping its bounds from the sort stack, as shown in this figure:



In PSION's implementation, the runtime buffer is used as a temporary sort stack. It starts at 8588 and the pointer is saved in 8584/8585, leaving location 8586/8587 as a 2-byte "sentinel" in case of stack underflow.

```
------------------------------------------------------------------
{Program UT$SORT part 4}

INIT_STACK: LDX #STACKBASE
            STX STACKPTR
{...}
------------------------------------------------------------------
```

As mentioned earlier, the recursion of Quicksort algorithm will be simulated by a loop. Within the loop, two parameters will be passed to the actual Quicksort routine (program QSORT): the item number to start sort from and the size of the current (sub-)array (or, if you prefer, the number of items to be sorted from the left-hand one).

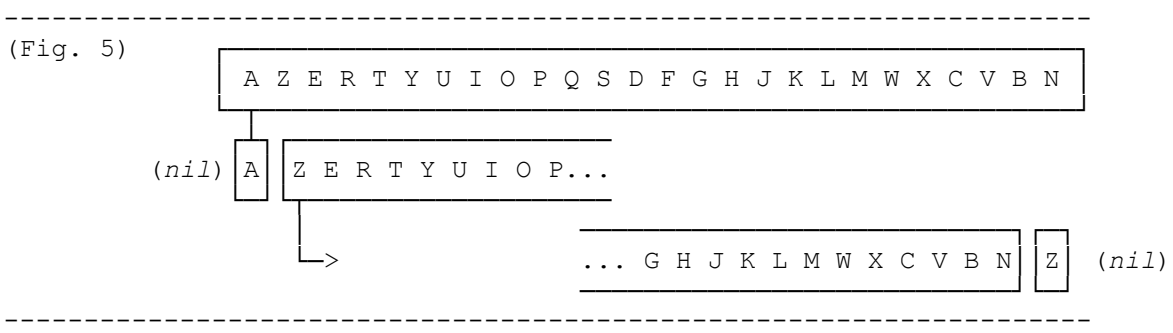
```
------------------------------------------------------------------
{Program UT$SORT part 5}

INIT_BOUND: LDX ITEMS         \number of items to be sorted (start: 26)
            CLR A             \left-hand item n° (start: 0)
            CLR B

MAINLOOP:   PSH X
            LDX STACKPTR      \check for stack overflow
            CPX #STACKTOP
            PUL X
            BCS SORT          \branch if no overflow
            LDX SORT_CELL
            SWI #0            \otherwise release sort cell
            LDA B #ERR254     \"OUT OF MEMORY"
            SEC               \signal error case
            BRA END           \RTS
{...}
------------------------------------------------------------------
```

Because of its rather big machinery (pivoting, partitioning and stack management), Quicksort doesn't perform well for small arrays. For instance, picking a pivot within a sub-array holding only two items would not be worthwhile. The general strategy is to set a "cutoff" which will stop quicksort, then finish up with another sorting algorithm that is efficient for small arrays, such as INSERTION SORT. Furthermore, when the cutoff is reached, the resultant sub-arrays are nearly sorted and insertion sort is particularly efficient in such a case. Detailed analysis show that a good cutoff should range between 4 and 20 (KNUTH's choice is 9, WEISS's is 10). For some unknown reason, PSION made a bad choice with a cutoff equal to 1.

```
------------------------------------------------------------------
{Program UT$SORT part 6}

SORT:       CPX #CUTOFF       \if cutoff reached, load bounds
            BLS POP           \of last remaining sub-array
            BSR QSORT         \main routine
{...}
------------------------------------------------------------------
------------------------------------------------------------------
{Program QSORT part 1}        {input : D=LEFT   IX=SIZE}

START:      STX SIZE          \size of current array to be sorted
            DEX               \get range 0 to n-1
            XGDX              \transfer left-hand item n° in IX
            STX LEFT
INIT_LPTR:  DEX               \initialize left-hand pointer
            STX LEFT_PTR      \(will be discussed later)
{...}
------------------------------------------------------------------
```

QSORT : CHOOSING THE PIVOT

The QUICKSORT algorithm works no matter which item is chosen as pivot. Detailed analysis and empirical study show that some choices are better than others.
Figure 3 was the ideal case when successive partitions are equally balanced around the pivot. Each pass is then profitable, progress being made toward the final base case. The worst case is a left-hand sub-array holding one item, and the right-hand holding n-1 items; that is to say almost all items go into one of the sub-arrays.
Intuitively (in my opinion at least), we would pick the center item of the array in order to get an equally balanced partition; this is a safe choice but it can be improved as we'll see.
An alternative choice is to use a "random" pivot. This is not bad, except that the random number generator is a time-consuming service, which doesn't fulfill the purpose of *QUICK*SORT. Furthermore, random number generators only appear to be random, but this is beyond the scope of this article.
At last, other choices could be to use the first (or the last) item of the array. This is to be avoided as, quite frequently, arrays are already presorted (or include a presorted section, as "DFGHJKL" in the QWERTY alphabet). This pivoting strategy often provides a poor partition. For instance, take the French version of the keyboard alphabet:



You can see that only two items (the pivots "A" and "Z") were sorted in two passes, that is to say Quicksort did essentially nothing at all…

Anyway, PSION opted for the common course, using the center item as pivot. A better idea would have been to choose the "median-of-three" strategy, as described by WEISS; this improvement will be discussed later.
In our example (the "QWERTY" alphabet), the center is "D". Not "F"! Don't forget the size of the array was decremented to get a range 0-25, instead of 1-26. The integer division 25\2 (LSR D statement) gives the result 12, and item n° 12 happens to be "D". Thus it will be the pivot:

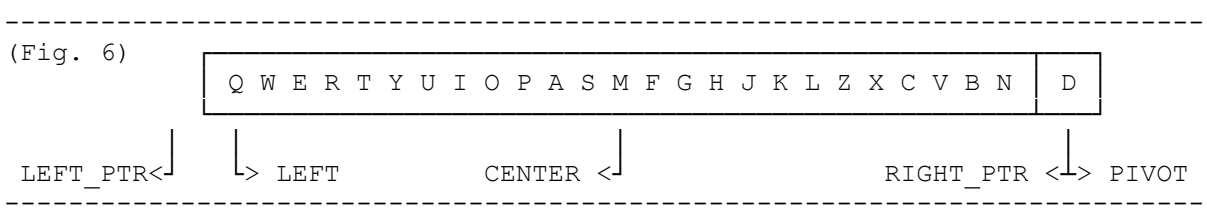
```
{Program QSORT part 2}

PICKPIVOT:  LSR D       \size of array divided by two
            ADD D LEFT
            STD CENTER
{...}
```

PARTITIONING STRATEGY

Here also there are several partitioning strategies available. The one chosen by PSION is probably the best. The first step was done in INIT_LPTR: part: a left-hand pointer (that we'll call LEFT_PTR) is initialized, pointing to the left-hand item (of the current array) minus one, because it will be immediately incremented at the beginning of the first partitioning loop (LEFT_PART:), as we'll see. A right-hand pointer (RIGHT_PTR) is then initialized too, pointing to the right-hand item of the current array.
At last, the pivot is extracted by swapping it with the right-hand item:

```
-------------------------------------------------------------------
{Program QSORT part 3}

INIT_RPTR:  LDD SIZE          \load size of current array
            ADD D LEFT_PTR    \add position of LEFT_PTR
            STD RIGHT_PTR     \                       (pointing to LEFT-1)
OUT_PIVOT:  STD PIVOT
            LDX CENTER
            BSR SWAP
{...}
-------------------------------------------------------------------
```

The current situation is then as follows:



The partitioning now consists in placing the small (relatively to the pivot) items in the left-hand part of the array, and the large (relatively to the pivot) items in the right-hand part.
We’re gonna move LEFT_PTR right until we find an item larger than the pivot, the comparison task being done by calling the user-supplied routine:

```
-------------------------------------------------------------------
{Program QSORT part 4}

LEFT_PART:  LDX LEFT_PTR
LEFT_LOOP:  INX
            BSR CALL_USER     \do comparison
            BCS LEFT_LOOP     \loop if left-hand item < pivot
            STX LEFT_PTR
{...}
-------------------------------------------------------------------
```

In our example, LEFT_PTR doesn’t move far, stopping to "Q". Then we’re gonna move RIGHT_PTR left until we find an item smaller than the pivot; we’ll also check that both pointers don’t cross (if so, partitioning stage is completed):

```
------------------------------------------------------------------
{Program QSORT part 5}

RIGHT_PART: LDX RIGHT_PTR
RIGHT_LOOP: DEX
            CPX LEFT_PTR      \check position
            BLS END_PART      \exit partition if pointers cross
            BSR CALL_USER     \do comparison
            BHI RIGHT_LOOP    \loop if right-hand item > pivot
            CPX LEFT_PTR
            BLS END_PART
            STX RIGHT_PTR
{...}
------------------------------------------------------------------
```

In our example, RIGHT_PTR will move to "B".
At this stage, we know that LEFT_PTR is pointing to a "large" item and RIGHT_PTR is pointing to a "small" item. Those elements are gonna be swapped, placing the "large" item in the right-hand part of the array, and the "small" item in the left-hand part:

```
------------------------------------------------------------------
{Program QSORT part 6}

XCHANGE:    LDD LEFT_PTR
            BSR SWAP
            LDX LEFT_PTR
            BRA LEFT_LOOP     \repeat process
{...}
------------------------------------------------------------------
```

The current situation is then as follows:

The process is repeated until both pointers cross. The final step of the partitioning is to place the pivot as a separator between the two resulting subarrays by swapping it with the item pointed to by LEFT_PTR:

```
-------------------------------------------------------------------
{Program QSORT part 7}

END_PART:   LDX LEFT_PTR
            LDD PIVOT
            BSR SWAP
{...}
-------------------------------------------------------------------
```

(Fig. 8)


```
-------------------------------------------------------------------
```

Before we go further, we could see the comparison task as it is done by the user-supplied routine (B holds flag 1, IX holds the tag of the item to be compared with the pivot, and GWV0 holds the tag of the pivot):

```
-------------------------------------------------------------------
{User-supplied routine part 2}

1$:   DEC B       \check flag
      BNE 2$      \branch if not 1

      LDA A 0,X   \load item to be compared with pivot
      LDX UTW_SO  \GWV0 holding tag of pivot
      LDA B 0,X   \load pivot
      CBA         \do comparison, setting CCR accordingly
      RTS         \all done
{...}
-------------------------------------------------------------------
```

The last stage of our QSORT program is to calculate the bound and sizes of the resulting sub-arrays:

```
      ------------------------------------------------------------------
{Program QSORT part 8}

END_PART:   LDX LEFT
            STX LEFT_BOUND    \start item of left-hand sub-array
            LDD SIZE
            LDX LEFT_PTR      \LEFT_PTR is pointing to PIVOT after last swap
            XGDX              \transfer LEFT_PTR in D register
            SUB D LEFT_BOUND  \PIVOT-LEFT_BOUND = LEFT_SIZE
            STD LEFT_SIZE     \size of resulting left-hand sub-array
            DEX               \decrement size of initial array (- PIVOT)
            XGDX              \transfer size in D register
            LDX LEFT_PTR      \LEFT_PTR still pointing to PIVOT
            INX               \next item
            STX RIGHT_BOUND   \start item of right-hand sub-array
            SUB D LEFT_SIZE   \subtract sizes (initial array - sub-array)
            STD RIGHT_SIZE    \size of resulting right-hand sub-array
            LDX LEFT_SIZE
            LDD LEFT_BOUND
            RTS
      ------------------------------------------------------------------
```

After the first pass, you should find the following values:

```
LEFT_BOUND = 0
LEFT_SIZE = 3    corresponding to sub-array "BCA"
RIGHT-BOUND = 4
RIGHT-SIZE = 22  corresponding to sub-array "TYUIOPESMFGHJKLZXWVQNR"
```

Then we come back to the driver UT$SORT. Now we have two resulting sub-arrays, we should loop to MAINLOOP: and repeat the process (picking a pivot and partitioning) until successive base cases are reached. But, as mentioned earlier, the CPU can’t perform several parallel quicksorts at a time. So we’ll continue the sort in the shortest array, storing bound and size of the longest array on a temporary stack:

```
------------------------------------------------------------------
{Program UT$SORT part 7}

CHECK_SIZE: CPX RIGHT_SIZE    \IX holds LEFT_SIZE
            BCC PUSH_LEFT     \branch if left sub-array > right sub-array

PUSH_RIGHT: PSH B             \save LEFT_BOUND
            PSH A
            LDD RIGHT_SIZE
            BSR PUSH          \push D onto sort stack
            LDD RIGHT_BOUND
            BSR PUSH          \push D onto sort stack
            PUL A             \restore LEFT_BOUND
            PUL B
            BRA MAINLOOP

PUSH_LEFT:  XGDX              \transfer LEFT_SIZE in D register
            BSR PUSH          \push D onto sort stack
            XGDX              \transfer LEFT_BOUND in D register
            BSR PUSH          \push D onto sort stack
            LDD RIGHT_BOUND
            LDX RIGHT_SIZE
            BRA MAINLOOP
{...}
------------------------------------------------------------------
```

When a resulting sub-array is empty or holds only one item, the base case is reached (*cf.* statement in SORT: CPX #CUTOFF). The sort may be continued through the remaining sub-arrays, whose bounds and sizes have been preserved on the temporary sort stack. The following part of UT$SORT pulls the required values from the stack. If the base of the stack is reached (*i.e.* the stack is empty), there are no more sub-arrays to be sorted, which means the sort is finished:

```
------------------------------------------------------------------
{Program UT$SORT part 8}

POP:  LDX STACKPTR
      DEX
      DEX
      LDD 0,X           \load bound
      DEX
      DEX
      CPX #STACKBASE    \check for underflow
      BCS SORT_END      \exit sort if stack empty
      STX STACKPTR
      LDX 0,X           \load size
      BRA MAINLOOP
------------------------------------------------------------------
```

The sort is finished; the tags for each sorted item are gonna be passed to the user-supplied routine in the IX register. In our example, the tag is the address

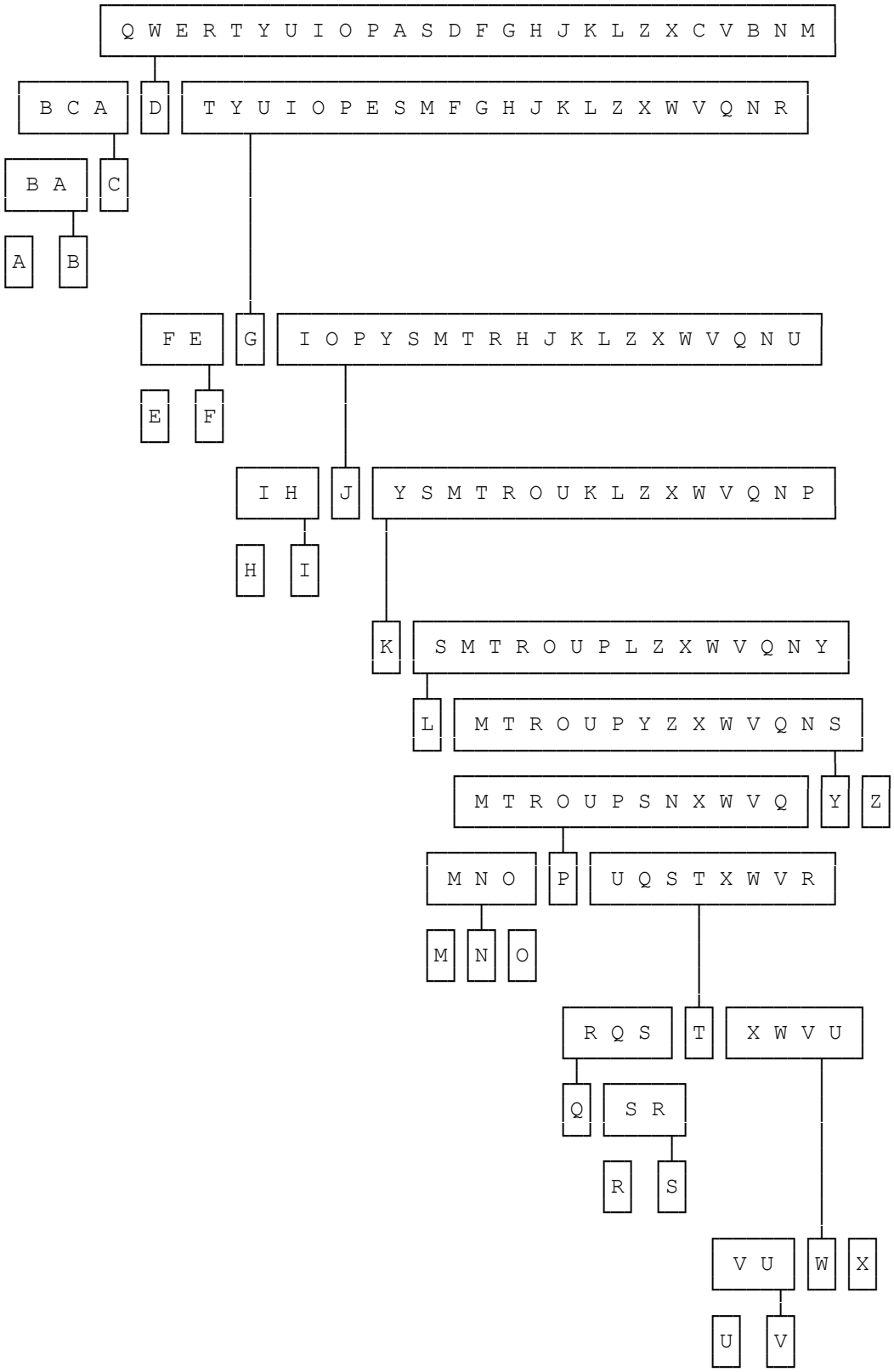
of each letter; the user-supplied routine will load the letter and copy it in a "target buffer".

```
------------------------------------------------------------------
{Program UT$SORT part 9}

SORT_END:   LDX #0            \initialize item number
END_LOOP:   PSH X             \save item number
            XGDX              \transfer item number in D register
            JSR SET_POINTER   \set pointer accordingly within sort cell
            LDA B #2          \set flag
            BSR CALL_USER     \pass tag to user-supplied routine
            PUL X             \restore item number
            INX               \next one
            CPX ITEMS         \check range
            BCS END_LOOP      \loop while tags still to be passed

            LDX SORT_CELL     \load tag of sort cell
            SWI #0            \release sort cell
            CLC               \signal everything's OK
END:        RTS               \all done
------------------------------------------------------------------
{User-supplied routine part 3}

2$:         LDA A 0,X         \load letter pointed to by IX
            LDX SRTTAG        \set pointer to target buffer
            STA A 0,X         \copy sorted letter in order
            INX               \consequently increment target pointer
            STX SRTTAG        \store pointer
            RTS               \all done

SRTTAG:     word SORTED

SORTED:     ascii "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
------------------------------------------------------------------
```

Actual PSION implementation (variables used):

```
GWV0              SIZE, PIVOT and RIGHT_SIZE
GWV1              LEFT and RIGHT_BOUND
GWV2              LEFT_BOUND
GWV3              LEFT_SIZE
GWBV0             US_ROUTINE
GWBV1             ITEMS and RIGHT_PTR
GWBV2             SORT_CELL
GWBV3             LEFT_PTR
Runtime buffer    temporary sort stack
```

(Fig. 9) THE WHOLE SORT



REFERENCES

- KNUTH (D. E.), *The Art of Computer Programming. Volume 3: Sorting and Searching,* Reading, Addison-Wesley, 1966.
- SEDGEWICK (R.), *Quicksort,* New York, Garland Publishing, 1978.
- WEISS (M. A.), *Data Structures and Algorithm Analysis,* Redwood, Benjamin/Cummings, 1992.