TRUE BASIC

The Original BASIC

Sorting & Searching Libraries

Reference Guide

Sorting and Searching Libraries

The library file SORTLIB.TRC contains several sorting and searching utilities. Each sorting and searching subroutine comes in two forms, one for numbers and one for strings. The name of the subroutine ends with an “N” for numbers, and in “S” for strings.

The two subroutines **SORTN** and **SORTS** perform ordinary in-place sorts. The two subroutines **PSORTN** and **PSORTS** perform indexed (or pointer) sorts.

The two subroutines **CSORTN** and **CSORTS** perform sorting according to a relation specified by the programmer. The two subroutines **CPSORTN** and **CPSORTS** perform indexed (or pointer) sorts according to a relation defined by the programmer.

The four subroutines **CSEARCHN**, **CSEARCHS**, **SEARCHN**, and **SEARCHS** search lists (numeric or string) for a match. **SEARCHN** and **SEARCHS** use the ordinary relational operator “=”. **CSEARCHN** and **CSEARCHS** perform searches according to a relation specified by the programmer.

CSORTN, CPSORTN, and CSEARCHN call a subroutine COMPAREN, which is included in SortLib.tru. It is currently coded to produce the usual ascending sort. If you require a different sorting relation, you can proceed in one of two ways. First, you can make the changes in the subroutine COMPAREN in SortLib.tru, and then recompile SortLib.tru. Second, you can include your own version of COMPAREN following the END statement in your main program; this definition takes precedence over the one in the library file.

CSORTS, CPSORTS, and CSEARCHS performing sorts and searches using special ordering relations specified by calling one of several relation-specifying subrouintes before invoking the sort. These special subroutine calls include:

Sort_Off	Sort using ASCII sorting order and entire string
Sort_ObserveCase	Treat upper- and lowercase as different (default)
Sort_IgnoreCase	Treat upper- and lowercase as equivalent
Sort_NiceNumbers_on	See the header of SortLib.tru for definitions
Sort_NiceNumbers_off	Ditto (default)
Sort_NoKeys	Sort using the entire string
Sort_OneKey	Sort on the substring field specified
Sort_TwoKeys	Sort on the two substring fields specified

CSEARCHN and CSEARCHS require the list to have been previously sorted using the same relations; i.e., use the same COMPAREN for CSEARCHN, and the same options for CSEARCHS as for CSORTS.The two subroutines **REVERSEN** and **REVERSES** simply reverse the order of the elements in the numeric or string array. That is, the first element will become the last, and so on.

CPSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CPSORTN (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CPSORTN (values(), indices())`

Summary: Performs a pointer sort on the values stored in `values` and stores the pointers, or indices, to the elements in `indices` in the order specified by a customized comparison routine.

Details: The **CPSORTN** subroutine performs a “pointer sort” on the values stored in the numeric array `values`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array that contains the first array’s indices arranged in the order of the sorted values. The **CPSORTN** subroutine returns this array of indices as `indices`.

For a more detailed discussion of pointer sorts, see the **PSORTN** subroutine later in this chapter.

The **PSORTN** subroutine compares elements based upon the standard relational operators in order to create a list of indices that represent the values sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CPSORTN** subroutine allows you to specify a particular comparison that will be used to determine the way in which the items will be ordered.

Note that the **CPSORTN** subroutine sorts the entire `values` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 zero-valued elements of `values` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,       93, 95,   68,     84,    88

CALL CPSortN(grade, indices)   ! Sort by grades
FOR i = 1 to 6
   LET j = indices(i)
   PRINT name$(j); grade(j)
NEXT i

END

SUB CompareN (a, b, compflag)
   IF a > b then
      LET compflag = -1
   ELSEIF a = b then
      LET compflag =  0
   ELSE
      LET compflag =  1
   END IF
END SUB
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted into descending order by grade. The result is the same as that of using PSORTN followed by CALL ReverseN (indicies).

Exceptions: None

See also: **CPSORTS**, **PSORTN**, **SORTN**

CPSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CPSORTS (*strarrarg*, *numarrarg*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CPSORTS (values$(), indices())`

Summary: Performs a pointer sort on the values stored in `values$` and stores the pointers, or indices, to the elements in `indices` in the order specified by the programmer.

Details: The **CPSORTS** subroutine performs a “pointer sort” on the values stored in the string array `values$`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array’s indices arranged in the order of the sorted values. The **CPSORTS** subroutine returns this array of indices as `indices`.

For a more detailed discussion of pointer sorts, see the **PSORTS** subroutine later in this chapter.

The **PSORTS** subroutine compares elements based upon the standard relational operators in order to create a list of indices that represent the values sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CPSORTS** subroutine allows you to specify the comparison that will be used to determine the way in which the items will be ordered.

Note that the **CPSORTS** subroutine sorts the entire `values$` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 null-valued elements of `values$` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,       93, 95,    68,     84,    88

CALL Sort_IgnoreCase
CALL CPSortS(name$, indices)   ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END
```

performs a case-blind pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by name.

Exceptions: None

See also: **CPSORTN**, **PSORTS**, **SORTS**

CSEARCHN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSEARCHN (*numarrarg*, *numex*, *numvar*, *numvar*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CSEARCHN (array(), number, index, found)`

Summary: Searches `array` for the value `number` utilizing a user-defined comparison and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `number` within `array`.

Details: The **CSEARCHN** subroutine searches through the numeric array `array` for an element with the value `number` and returns the subscript of its location in `index`. This search is performed using a customized comparison subroutine defined by the programmer.

The **SEARCHN** subroutine compares elements based upon the standard relational operators in order to locate the value `number` within `array`. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSEARCHN** subroutine requires that you have sorted the array using `CSORTN`, and that you continue to use the same `CompareN` subroutine.

It is your responsibility to ensure that the behavior of the `CompareN` subroutine is well-defined and bug-free. If your `CompareN` subroutine is not well-behaved, the search results may not be valid.

You may define `CompareN` in the main program file.

Since the **CSEARCHN** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **CSORTN** subroutine) before being passed to the **CSEARCHN** subroutine. In general, the **CSEARCHN** subroutine should utilize the same form of the `CompareN` subroutine used by the **CSORTN** subroutine which sorted the array.

If the value of `number` exists in `array`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `number` cannot be located in `array`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `number` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `number`. If `number` is greater than every element in `array`, the value of `index` will be returned equal to `array`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(100)
RANDOMIZE
FOR i = 1 to 100
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL CSortN(array)

DO
   INPUT PROMPT "Search value (0 to quit): ": number
   IF number <= 0 then EXIT DO
   CALL CSearchN(array,number,i,found)
   IF found <> 0 then
      PRINT "Found: "; array(i)
   ELSE
      PRINT "Not found."
   END IF
LOOP

END

SUB CompareN (a, b, compflag)
    IF a > b then
       LET compflag = -1
    ELSEIF a = b then
       LET compflag =  0
    ELSE
       LET compflag =  1
    END IF
END SUB
```

sorts a list of 20 random numbers between 1 and 100 into descending order and allows the user to search the results.

Exceptions: None

See also: **CSORTN**, **SEARCHN**, **CSEARCHS**, **CSORTS**

CSEARCHS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSEARCHS (*strarrarg*, *strex*, *numvar*, *numvar*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL CSEARCHS (array$(), string$, index, found)`

Summary: Searches `array$` for the value `string$` utilizing a user-specified relation and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `string$` within `array$`.

Details: The **CSEARCHS** subroutine searches through the string array `array$` for an element with the value `string$` and returns the subscript of its location in `index`. This search is performed using the relations specified by the programmer.

The **SEARCHS** subroutine compares elements based upon the standard relational operators in order to locate the value `string$` within `array$`. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSEARCHS** subroutine allows you to specify the comparison that will be used to locate the items.

Since the **CSEARCHS** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **CSORTS** subroutine) before being passed to the **CSEARCHS** subroutine. In general, the **CSEARCHS** subroutine should use the same options used by the **CSORTS** subroutine which sorted the array.

If the value of `string$` exists in `array$`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `string$` cannot be located in `array$`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `string$` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `string$`. If `string$` is greater than every element in `array$`, the value of `index` will be returned equal to `array$`'s upper bound plus 1.

Example: The following program:

```
! Sort by last 3 letters, then search for same.
!
DIM array$(10)
MAT READ array$
DATA operculum, partlet, pettifog, grisette, douceur
DATA pollex, sannup, duende, keeshond, maccaboy

CALL Sort_OneKey (4, 6)
CALL CSortS(array$)
DO
   INPUT PROMPT "Search string (aaa to quit): ": string$
   IF string$ = "aaa" then EXIT DO
   CALL CSearchS(array$,string$,i,found)
   IF found<>0 then
      PRINT "Found: "; array$(i)
   ELSE
      PRINT "Not found."
   END IF
LOOP
END
```

sorts a list of string data by characters 4 through 6 in each element and then allows the user to search the list based on these same characters in an element.

Exceptions: None

See also: **CSORTS**, **SEARCHS**, **CSEARCHN**, **CSORTN**

CSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSORTN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CSORTN (array())`

Summary: Sorts the specified numeric array using the customized comparison routine named `CompareN`.

Details: The **CSORTN** subroutine sorts the elements of the specified numeric array into the order determined by a customized comparison subroutine.

The **SORTN** subroutine compares elements based upon the <= relational operator in order to create a list sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSORTN** subroutine allows you to define a particular comparison that will be used to determine the ordering of the items. You do so by defining an external subroutine named `CompareN` as in the following example:

The **CSORTN** subroutine performs an “in-place” sort, which means that it uses very little memory over and above that already occupied by the array itself.

Although it is of little consequence, you may also be interested to know that the sorting algorithm used by the **CSORTN** subroutine is not stable; if you require a stable sort, use the **CPSORTN** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **CSORTN** routine a very efficient, general-purpose sorting routine. Note, however, that since the **CSORTN** subroutine calls the `CompareN` subroutine for each comparison, it is not as fast as the **SORTN** subroutine.

Note that the **CSORTN** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 zeroes merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(100)
RANDOMIZE
FOR i = 1 to 100
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL CSortN(array)
MAT PRINT array

END

SUB CompareN (a, b, compflag)
    IF a > b then
       LET compflag = -1
    ELSEIF a = b then
       LET compflag =  0
    ELSE
       LET compflag =  1
    END IF
END SUB
```

generates an array of 100 random numbers, sorts it into descending order, and prints the sorted result on the screen.

Exceptions: None

See also: **CSORTS**, **CPSORTN**, **SORTN**, **REVERSEN**

CSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSORTS (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL CSORTS (array())`

Summary: Sorts the specified string array using the customized comparison specified by the programmer.

Details: The **CSORTS** subroutine sorts the elements of the specified string array into the order determined by a customized comparison.

The **SORTS** subroutine compares elements based upon the <= relational operator in order to create a list sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSORTS** subroutine allows you to specify the comparison that will be used to determine the ordering of the items.

The **CSORTS** subroutine performs an “in-place” sort, which means that it uses very little memory over and above that already occupied by the array itself.

Although it is of little consequence, you may also be interested to know that the sorting algorithm used by the **CSORTS** subroutine is not stable; if you require a stable sort, use the **CPSORTS** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **CSORTS** routine a very efficient, general-purpose sorting routine. Note, however, that since the **CSORTS** subroutine calls the `CompareS` subroutine for each comparison, it is not as fast as the **SORTS** subroutine.

Note that the **CSORTS** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 null strings merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"
LIBRARY "CompNum.TRC"

DIM array$(100)
RANDOMIZE
FOR i = 1 to 100
    LET array$(i) = "Item " & Str$(Int(100*Rnd) + 1)
NEXT i
CALL Sort_NiceNumbers_on
CALL CSortS(array$)
MAT PRINT array$

END
```

generates an array of 100 strings containing numeric values, sorts it using the version of `CompareS` contained in the COMPNUM library file, and prints the sorted result on the screen.

Exceptions: None

See also: **CSORTN**, **CPSORTS**, **SORTS**, **REVERSES**

PSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL PSORTN (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL PSORTN (values(), indices())`

Summary: Performs a pointer sort on the values stored in `values` and stores the pointers, or indices, to the elements in `indices` in sorted order.

Details: The **PSORTN** subroutine performs a "pointer sort" on the values stored in the numeric array `values`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array's indices arranged in the order of the sorted values. The **PSORTN** subroutine returns this array of indices as `indices`.

For example, if `values` contained the following items:

```
10   12   23   14   -8   11   6
```

the resulting `indices` array would contain the following items:

```
5   7   1   6   2   4   3
```

but the items in `values` will still be in their original order:

```
10   12   23   14   -8   11   6
```

Notice that you can therefore print the elements of `values` in sorted order with code similar to the following:

```
FOR i = Lbound(indices) to Ubound(indices)
    PRINT values(indices(i))
NEXT i
```

Because they do not change the ordering of information in the `values` array, pointer sorts are particularly useful when working with "parallel arrays."

Note that the **PSORTN** subroutine sorts the entire `values` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 zero-valued elements of `values` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,       93, 95,    68,     84,    88

CALL PSortN(grade, indices)    ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by grades.

Exceptions: None

See also: **PSORTN**, **CPSORTS**, **SORTS**

PSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL PSORTS (*strarrarg*, *numarrarg*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: CALL PSORTS (values$(), indices())

Summary: Performs a pointer sort on the values stored in `values$` and stores the pointers, or indices, to the elements in `indices` in sorted order.

Details: The **PSORTS** subroutine performs a "pointer sort" on the values stored in the string array `values$`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array's indices arranged in the order of the sorted values. The **PSORTS** subroutine returns this array of indices as `indices`.

For example, if `values$` contained the following items:

```
bat   zoo   cat   ant   dog   pig
```

the resulting `indices` array would contain the following items:

```
4   1   3   5   6   2
```

but the items in `values$` will still be in their original order:

```
bat   zoo   cat   ant   dog   pig
```

Notice that you can therefore print the elements of `values$` in sorted order with code similar to the following:

```
FOR i = Lbound(indices) to Ubound(indices)
   PRINT values$(indices(i))
NEXT i
```

Because they do not change the ordering of information in the `values$` array, pointer sorts are particularly useful when working with "parallel arrays."

Note that the **PSORTS** subroutine sorts the entire `values$` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 null-valued elements of `values$` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,       93, 95,    68,     84,    88

CALL PSortS(grade$, indices)   ! Sort by grades
FOR i = 1 to 6
   LET j = indices(i)
   PRINT name$(j); grade(j)
NEXT i

END
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by name.

Exceptions: None

See also: **PSORTN**, **CPSORTS**, **SORTS**

REVERSEN Subroutine

Library: SORTLIB.TRC

Syntax: CALL REVERSEN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL REVERSEN (array())`

Summary: Reverses the order of the elements within `array`.

Details: The **REVERSEN** subroutine reverses the order of the elements stored within the specified numeric array. In other words, it swaps the first and last elements, the second and next-to-last, and so forth.

Although it can be used on any numeric array, the **REVERSEN** subroutine is most often used to reverse the results of the **SORTN** or **CSORTN** subroutines to produce a list sorted in descending order. It can also be used to reverse the pointer list produced by **PSORTN**, **CPSORTN**, **PSORTS** or **CPSORTS**.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(20)
FOR i = 1 to 20
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL SortN(array)
CALL ReverseN(array)
MAT PRINT array

END
```

generates an array of random values between 1 and 100 and prints it sorted into descending order.

Exceptions: None

See also: **SORTN**, **CSORTN**, **REVERSES**

REVERSES Subroutine

Library: SORTLIB.TRC

Syntax: CALL REVERSES (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL REVERSES (array$())`

Summary: Reverses the order of the elements within `array$`.

Details: The **REVERSES** subroutine reverses the order of the elements stored within the specified string array. In other words, it swaps the first and last elements, the second and next-to-last, and so forth.

Although it can be used on any string array, the **REVERSES** subroutine is most often used to reverse the results of the **SORTS** or **CSORTS** subroutines to produce a list sorted in descending order.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(20)
FOR i = 1 to 20
    LET array$(i) = Chr$(Int(26*Rnd) + 65)
NEXT i
CALL SortS(array$)
CALL ReverseS(array$)
MAT PRINT array$

END
```

generates an array of random uppercase letters and prints it sorted into descending order.

Exceptions: None

See also: **SORTS**, **CSORTS**, **REVERSEN**

SEARCHN Subroutine

Library: SORTLIB.TRC

Syntax: CALL SEARCHN (*numarrarg*, *numex*, *numvar*, *numvar*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SEARCHN (array(), number, index, found)`

Summary: Searches `array` for the value `number` and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `number` within `array`.

Details: The **SEARCHN** subroutine searches through the numeric array `array` for an element with the value `number` and returns the subscript of its location in `index`.

Since the **SEARCHN** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **SORTN** subroutine) before being passed to the **SEARCHN** subroutine.

If the value of `number` exists in `array`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `number` cannot be located in `array`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `number` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `number`. If `number` is greater than every element in `array`, the value of `index` will be returned equal to `array`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(20)
FOR i = 1 to 20
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL SortN(array)

DO
   INPUT PROMPT "Enter a number 1 to 100 (0 to quit): ": number
   IF number <= 0 then EXIT DO
   CALL SearchN(array, number, index, found)
   IF found <> 0 then
      PRINT "Found at"; index
   ELSE
      PRINT "Not found"
   END IF
LOOP

END
```

generates an array of random values between 1 and 100 and allows the user to search it.

Exceptions: None

See also: **SORTN**, **SEARCHS**, **CSEARCHN**, **CSORTN**

SEARCHS Subroutine

Library: SORTLIB.TRC

Syntax: CALL SEARCHS (*strarrarg*, *strex*, *numvar*, *numvar*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL SEARCHS (array$(), string$, index, found)`

Summary: Searches `array$` for the value `string$` and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `string$` within `array`.

Details: The **SEARCHS** subroutine searches through the string array `array$` for an element with the value `string$` and returns the subscript of its location in `index`.

Since the **SEARCHS** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **SORTS** subroutine) before being passed to the **SEARCHS** subroutine.

If the value of `string$` exists in `array$`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `string$` cannot be located in `array$`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `string$` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `string$`. If `string$` is greater than every element in `array$`, the value of `index` will be returned equal to `array$`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(20)
FOR i = 1 to 20
    LET array$(i) = Chr$(Int(26*Rnd) + 65)
NEXT i
CALL SortS(array$)

DO
   INPUT PROMPT "Enter an uppercase letter (a to quit): ": string$
   IF string$ = "a" then EXIT DO
   CALL SearchS(array$, string$, index, found)
   IF found <> 0 then
      PRINT "Found at"; index
   ELSE
      PRINT "Not found"
   END IF
LOOP

END
```

generates an array of random uppercase letters and allows the user to search it.

Exceptions: None

See also: **SORTS**, **SEARCHN**, **CSEARCHS**, **CSORTS**

SORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL SORTN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SORTN (array())`

Summary: Sorts the specified numeric array using a quick sort.

Details: The **SORTN** subroutine sorts the elements of the specified numeric array into ascending order. Thus, the array element with the lowest value will be found in the first element of `array` after the sort, and the array element with the highest value will be found in the last element of `array`.

The **SORTN** subroutine performs an “in-place” sort, which means that it uses very little memory over and above that already occupied by the array itself.

The sorting algorithm used by the **SORTN** subroutine is not stable; if you require a stable sort, use the **PSORTN** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **SORTN** routine a very efficient, general-purpose sorting routine.

Note that the **SORTN** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 zeroes merged into the sorted result.

To sort an array into descending order, use the **REVERSEN** subroutine to reverse the results of the **SORTN** subroutine.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(1000)
RANDOMIZE
FOR i = 1 to 1000
    LET array(i) = Rnd
NEXT i
CALL SortN(array)
MAT PRINT array
END
```

generates an array of 1000 random numbers, sorts it, and prints the sorted result on the screen.

Exceptions: None

See also: **SORTS**, **CSORTN**, **PSORTN**, **CPSORTN**, **REVERSEN**

SORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL SORTS (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL SORTS (array$())`

Summary: Sorts the specified string array using a quick sort.

Details: The **SORTS** subroutine sorts the elements of the specified string array into ascending order. Thus, the array element with the lowest value will be found in the first element of `array` after the sort, and the array element with the highest value will be found in the last element of `array`.

The values of the elements will be compared as strings, which means that they are compared character by character on the basis of each character’s numeric code. Thus, the string value `"Zebra"` will be considered less than the string value `"apple"`. This is particularly important when sorting strings which represent numeric constants, for the string value `"123"` will be considered less than the string value `"2"`, which can lead to unexpected results.

The **SORTS** subroutine performs an “in-place” sort, which means that it uses very little memory over and above that already occupied by the array itself.

The sorting algorithm used by the **SORTS** subroutine is not stable; if you require a stable sort, use the **PSORTS** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **SORTS** routine a very efficient, general-purpose sorting routine.

Note that the **SORTS** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 null strings merged into the sorted result.

To sort an array into descending order, use the **REVERSES** subroutine to reverse the results of the **SORTS** subroutine.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(1)
MAT INPUT array$(?)
CALL SortS(array$)
MAT PRINT array$
END
```

obtains an array of string values from the user, sorts it, and prints the sorted result on the screen.

Exceptions: None

See also: **SORTN**, **CSORTS**, **PSORTS**, **CPSORTS**, **REVERSES**