[A slightly shortened version of this article
was published as “Made to Order Sorts” in
PC Resource (January 1988), pp. 87-90, 92, 94.]

ALPHABETIZING DATA

As children in elementary school we were taught to recite the alphabet *in order*: “Aay, Bee, See, Dee, Eii, Eff, Ghee, Aaych, …, Why and Zee”. There is nothing natural about this particular ordering: it is strictly a matter of convention. (When and where it was settled upon I haven’t the remotest notion.) Then, having mastered the ordering, we were taught to apply that knowledge to alphabetize lists of words. The procedure is surprisingly complex, and its mastery by mere eight-year olds attests to the elevated intellectual capacities of human beings. One merely has to try to write down the procedure in a flow chart to see how complex it truly is. In any event, most of us probably emerged from the exercise of learning to use a dictionary believing that we knew all there is to know about alphabetizing. If only that were all there is to it. The trouble is that our third-grade teachers had not reckoned on our having to program computers to alphabetize data. Nowadays we have to make our rules very much more explicit, and when we do, we begin to discover all sorts, er, kinds, of problems.

Mapping strings onto numbers

Every sorting algorithm, from the simplest (the bubble sort) to the most sophisticated (for example, the fastsort) at some point uses a comparison test. Two items are selected for comparison; if the first is greater than the second, then an exchange or swap is effected: either the items themselves or pointers in a separate list are interchanged. But what exactly does it mean to say that “the first is *greater* than the second” when one is talking about *alphabetic* data rather than numerical data?

Alphabetic strings may be reckoned as “less than”, “equal to” or “greater than” by coding each of the symbols in the strings onto the whole numbers. For example, if we let “a” = “001”, “b” = “002”, …, and “z” = “026”, then the two words “`apple`” and “`after`” may be represented by the numerical sequences “001 016 016 012 005” and “001 006 020 005 018” respectively. Starting at the left, each term of these numerical sequences can be compared with the corresponding term in the other sequence. In our example, the first positions (the “001”s) are the same, and thus we move on to the second. Here the two values (“016” and “006”) are unequal. Since 016 is greater than 006, the two items (“`apple`” and “`after`”) are exchanged and this comparison is complete; the sorting routine proceeds then to compare another pair of items.

An indefinite number of different numerical values can be used for coding alphabetic strings. One of these, ASCII (American Standard Code for Information Interchange) coding, is built into the hardware of every micro computer.[1,2] In ASCII coding (see *Table 1*), the uppercase letters “A” through “Z” are represented by the numbers 065 through 090; and the lowercase letters “a” through “z” by the numbers 097 through 122.

ASCII Values

000-031 and 127 are non-printing, control characters.

032	[blank]	051	3	070	F	089	Y	108	l
033	!	052	4	071	G	090	Z	109	m
034	"	053	5	072	H	091	[	110	n
035	#	054	6	073	I	092	\	111	o
036	$	055	7	074	J	093	]	112	p
037	%	056	8	075	K	094	^	113	q
038	&	057	9	076	L	095	_	114	r
039	’	058	:	077	M	096	‘	115	s
040	(	059	;	078	N	097	a	116	t
041	)	060	<	079	O	098	b	117	u
042	*	061	=	080	P	099	c	118	v
043	+	062	>	081	Q	100	d	119	w
044	,	063	?	082	R	101	e	120	x
045	–	064	@	083	S	102	f	121	y
046	.	065	A	084	T	103	g	122	z
047	/	066	B	085	U	104	h	123	{
048	0	067	C	086	V	105	i	124	|
049	1	068	D	087	W	106	j	125	}
050	2	069	E	088	X	107	k	126	~

Table 1

Thus, in BASIC[3], for example, two alphabetic items may be compared and ordered by the simple statement:

```
IF first.item$ > second.item$ THEN
    SWAP first.item$, second.item$
END IF
```

The ASCII equivalents of each item are used for the comparison, and the two items are interchanged if the first is greater, in the sense just explained, than the second.

The problem with ASCII coding

Suppose you need to write a computer sorting routine to alphabetize a list of items including the following: “`make`”, “`make work`” and “`makeshift`”. Depending on how many items (records) there are to be sorted in all, and depending on how much time you want to spend writing the code, and depending on whether you need a fast sort or can settle for a slower one, you choose from among a list of readily available, familiar, algorithms, for example, a bubble sort, a selection sort, a quick sort, a Shell sort or a fastsort.[4] If you are using IBM/MS DOS on your computer, you might choose the utility program SORT.EXE. The results of any of these sorts should be the same. The data will be alphabetized this way:

```
make
make work
makeshift
```

However, if you compare this latter list with the *Oxford English Dictionary*, for example, you will discover that these same items would be alphabetized differently:

```
make
makeshift
make work
```

The results will be even worse if any of your data is within quotation marks. Any of the aforementioned sorts would yield:

```
"ideas"
absolutism
idealism
```

If you are sorting proper names, any ordinary sort would produce:

```
D & J's Bakery
D J Allen Wholesale Groceries
Dinesen, Isak
Mabbett, Alice
MacGregor, F.
McCarthy, Richard
```

But telephone directories nowadays adopt a totally different convention and would list these same items in this fashion:

```
D J Allen Wholesale Groceries
D & J's Bakery
Dinesen, Isak
McCarthy, Richard
MacGregor, F.
Mabbett, Alice
```

In the latter case, among other peculiarities, "`&`" and the possessive ending "`'s`" are ignored in the alphabetizing, and "`Mac`" and "`Mc`" are treated as equivalent to "`Maa`".

The results will be more peculiar still if your data includes some records which begin in lowercase and others which are capitalized, for example,

```
acoustics
Abba
Zamfir, G.
zither
```

will be alphabetized this way:

```
Abba
Zamfir, G.
acoustics
zither
```

All capitalized words will be placed before any lowercase words. That is, ordinary computer sorting routines yield a so-called "Names" list, followed by a so-called "Subject" list.[5]

And finally, ASCII coding yields unacceptable results when it is used to alphabetize lengthy expressions, for example, the first lines of poems. Anyone who used IBM/MS DOS's

SORT.EXE program, even specifying ascending order, to alphabetize first lines of poems would be dismayed to get

```
"I've seen the Thousand Islands"
"Is there no secret place on the face of the Earth"
"Isn't it strange"
```

which is exactly the *reverse* order to the one desired.

Clearly, there is more to alphabetizing data than simply choosing the proper sorting algorithm.

Application-specific, non-ASCII coding

If ASCII coding is not suitable for your purposes, you will have to choose some other manner of mapping your data strings onto the whole numbers before they are compared in the sorting routine. Several questions must be attended to in choosing this mapping.

1. **Ignore characters** – Are any symbols to be totally ignored? You may, for example, want to disregard quotation marks when alphabetizing your data. Are blankspaces to be ignored (as they are in most modern dictionaries)? Are commas to be ignored? How, for example, do you want the following items alphabetized?

   ```
   Harvard College
   Harvard, John
   ```

   If commas (and blanks) are ignored, you will get the above ordering (since the code for "C" is "067" and for "J", "074"). If, on the other hand, blanks are omitted but commas are retained and treated as they are in ASCII (that is, represented by "044"), then you will get the reverse ordering.

2. **Character equivalence** – Are any single-symbols to be regarded as equivalent to other single-symbols? Does your data, for example, include foreign characters, `e-grave` [no. 138 in IBM PC; 143 in Mac], or `u-umlaut` [no. 129 in IBM PC; 159 in Mac]? If so, what English letters are to be regarded as their equivalents?

3. **Phrase equivalence** – Are any sequences of symbols to be regarded as equivalent to other sequences of symbols? Again, to use the example of phone directories, numerical street names ("`28th Ave.`") are alphabetized according to their English ("`Twenty-eighth Avenue`") equivalents, thus:

   ```
   Rosewood Circle
   28th Ave.
   Wellington St.
   ```

   The diphthongs "æ" and "Æ" are represented by single characters in micro computers (nos. 145 and 146 in IBM; 190 and 174 in Mac). You may want, instead, that these be treated not as single letters which occur later than "z" (122), but as sequences of two letters, "ae" (097 101) and "Ae" (065 101).

4. **Upper and lower case** – Data may be capitalized, e.g. "`Cuba`"; or may be in lowercase, "`cigars`"; or may be in uppercase, "`CIA`". Do you want to preserve the distinction between capitalized and lowercase items? That is, do you want a "Names" listing

followed by a "Subject" listing? Do you want to treat uppercase as merely capitalized, so that, for example, "`CIA`" occurs between "`California`" and "`Cuba`"? If not, ASCII coding will place "`CIA`" before "`California`".

Space constraints vs. time constraints

If your data requires that you use a non-ASCII mapping, you will have to restore your data to ASCII format after the records have been converted and sorted. Word processors, spreadsheets, databases and printers all assume standard ASCII coding for alphabetic data. Since any mapping you choose will almost certainly not be one-one, it will be impossible to undo the conversion by a reverse mapping. This means that the original data cannot simply be discarded after they are converted; quite the contrary: it will be essential that the original data be retained while the converted records are being sorted.

Since the original ASCII records must be retained, if you were to convert all of the data to a non-ASCII code before sorting, this would require a free working space in computer memory roughly equal to the amount of space your to-be-sorted data currently occupies. There is, thus, a substantial *space* overhead in converting all of the data wholesale.

If, on the other hand, you were to convert the data as items are being compared (and then discard the conversions after the comparison), there would be effectively little or no space overhead. However, every known sorting algorithm (indeed every theoretically possible sorting algorithm) requires that some at least of the data be used in a comparison test more than once. This means that individual items would have to be converted from ASCII more than once, some perhaps even a great number of times. Thus converting each record just prior to its being used in a comparison test, exacts a substantial *time* overhead.

Compromise method

There is, however, a compromise method you can use when space is tight and you do not want to pay the price of a high time overhead. In this compromise method, the converting of data occurs only once per record and the computer memory never has to hold both the original data and the converted data together. Briefly: the original data are saved to disk; the original data are then converted in place (that is, the converted records overwrite the original ones); the converted data are sorted, and a Re-allocation Table is constructed showing where each record has ended up; the sorted records are then discarded (erased); and, finally, the original data are read back into memory and are assigned new positions in accord with the pointers in the Re-allocation Table. This method optimizes the use of data space because the space needed to store the various pointers is usually far, far less (a few bytes per record) than the original data itself (alphabetic strings can be hundreds of bytes in length).

Step by step solution

In greater detail, the strategy for sorting alphabetic data using non-ASCII equivalents for the data items and using the compromise technique just described is this:

- Save the original data array to disk.
- Convert each item in computer memory using whatever non-ASCII mapping you need. These new, converted, records might be called "pseudo records", "proxy records",

"dummy records", "stand-in records", or some such thing. I will call them "pseudo records".

- Sort the pseudo records using a suitable algorithm, keeping track in a Tag Table where each pseudo record ends up. I.e. as records are swapped, swap their tags as well:

```
IF condition THEN
    SWAP ARRAY$(m), ARRAY$(n)
    SWAP TAG(m), TAG(n)
END IF
```

The effect would be, for example:

Before sorting		After sorting	
Record	Tag	Record	Tag
mango	1	apple	2
apple	2	grape	4
orange	3	mango	1
grape	4	orange	3

In this example, the Tag Table tells us that item #1 ("mango") has ended up in slot #3; item #2 ("apple") has ended up in slot #1; and so on.

- Create a Re-allocation Table, from the Tag Table, showing where each original item is to be placed:

Tag Table (from above)	Re-allocation Table
2	3
4	1
1	4
3	2

Since the Tag Table tells us that the original item #1 is to end up in the third position, we put a "3" in position #1 in the Re-allocation table. Since the original item #2 is to end up in the first position, we put a "1" in position #2 in the Re-allocation Table. Etc.

- Erase the pseudo records which have now served their purpose and are no longer needed. Then read back in the original array, item by item, and place each item into its new position in the array according to the pointers in the Re-allocation Table.

Implementation in BASIC

With the strategy mapped out, we can turn to implementing it in a program. The items to be alphabetized are to be stored in an array called, simply, "ARRAY$()". It, the TAG table and the REALLOCATE table must bear the same dimensions, let's say 1000. The MAIN program steps through the procedure just described, by calling subroutines to perform each of the required operations. These subroutines are listed at the end of this article.

```
DEFINT A-Z
OPTION BASE 0
DIM ARRAY$(1000),  TAG(1000),  REALLOCATE(1000)
DIM TRANSLATION.TABLE(255)
DIM OLD.PHRASE$(250),  NEW.PHRASE$(250)
NULL$ = CHR$(0)

MAIN :
    GOSUB READ.INPUT.DATA
    GOSUB CONSTRUCT.TAG.TABLE
    GOSUB SAVE.ORIGINAL.DATA
    GOSUB CREATE.PHRASE.LEXICON
    GOSUB CREATE.TRANSLATION.TABLE
    GOSUB CREATE.PSEUDO.RECORDS
    GOSUB SORT.PSEUDO.RECORDS
    GOSUB CREATE.REALLOCATION.TABLE
    GOSUB READ.AND.SORT.SAVED.DATA
    GOSUB SAVE.SORTED.DATA
END
```

Certain steps in this procedure merit special comment.

Case insensitivity

Let's begin with a relatively easy requirement. Suppose you want the final ordering to be case insensitive, that is, you want capitalized, lowercase and uppercase words to be interleaved irrespective of case. One way to do this is to create pseudo records entirely in uppercase. BASIC provides a function [UCASE$] to effect the conversion:

```
ARRAY$(n) = UCASE$( ARRAY$(n) )
```

The function UCASE$[6] provides a highly specific translation of each of the letters in a string: it converts lowercase letters to uppercase. But often we want other kinds of character-by-character translations and require a more general conversion routine.

Translation tables

There is a considerably more powerful method for effecting character-by-character translations, one which allows you to handle not just lowercase letters, but all 256 characters of the ASCII set. With a Translation Table (sometimes called a "lookup table"), the conversion routine matches each item in the data string against its corresponding entry in the table and converts it according to the assigned value of that entry. For example, we have seen that the ASCII value of "a" is 097. If we want to convert "a" to "A", we would place the value for "A" (viz. 065) in the 97th position of the Translation Table.[7]

A Translation Table can handle not just the converting of lowercase letters to uppercase; it can as well eliminate unwanted symbols, for example quotation marks; it can translate symbols, for example, it can replace the foreign lowercase u-umlaut with an uppercase English "U"; it can eliminate blankspaces; it can preserve commas; and so on.

To create a Translation Table, you will need to read a series of DATA statements.

```
CREATE.TRANSLATION.TABLE :
    RESTORE NON.ASCII.CODES
    FOR N = 0 TO 255  ' note: Option Base 0 required
        READ TRANSLATION.TABLE(N)
    NEXT N
RETURN
```

The DATA statements for the Translation Table consist of the new values to be assigned to each of the ASCII codes 000 through 255. For example, suppose you wanted to strip all control characters (001-031 → 000), wanted to convert all lowercase letters to uppercase (097-122 → 065-090), wanted to ignore blankspaces (032 → 000), wanted to preserve commas (044 → 044), and wanted to leave all numerals as they are (048-057 → 048-057), etc. You would write:

```
NON.ASCII.CODES :
    REM – ASCII codes 000-009 become:
    DATA  0, 0, 0, 0, 0, 0, 0, 0, 0, 0

    REM – ASCII codes 010-019 become:
    DATA  0, 0, 0, 0, 0, 0, 0, 0, 0, 0

    REM – ASCII codes 020-029 become:
    DATA  0, 0, 0, 0, 0, 0, 0, 0, 0, 0

    REM – ASCII codes 030-039 become:
    DATA  0, 0, 0, 0, 0, 0, 0, 0, 0, 0

    REM – ASCII codes 040-049 become:
    DATA  0, 0, 0, 0, 44, 0, 0, 0, 48, 49

    REM – ASCII codes 050-059 become:
    DATA  50, 51, 52, 53, 54, 55, 56, 57, 0, 0

    REM – ASCII codes 060-069 become:
    DATA  0, 0, 0, 0, 0, 65, 66, 67, 68, 69

    REM – ASCII codes 070-079 become:
    DATA  70, 71, 72, 73, 74, 75, 76, 77, 78, 79

    REM – ASCII codes 080-089 become:
    DATA  80, 81, 82, 83, 84, 85, 86, 87, 88, 89

    REM – ASCII codes 090-099 become:
    DATA  90, 0, 0, 0, 0, 0, 0, 65, 66, 67

    REM – ASCII codes 100-109 become:
    DATA  68, 69, 70, 71, 72, 73, 74, 75, 76, 77

    REM – ASCII codes 110-119 become:
    DATA  78, 79, 80, 81, 82, 83, 84, 85, 86, 87

    REM – ASCII codes 120-127 become:
    DATA  88, 89, 90, 0, 0, 0, 0, 0

    REM – etc. … …
```

(ASCII codes greater than 127 differ for different micro computers. You will have to consult your computer Manual to complete this DATA list.)

To translate a record, ARRAY$(I), using the data in this Translation Table, we would issue a CALL to the CHARACTER.SUBSTITUTION subprogram, e.g.

```
CALL CHARACTER.SUBSTITUTION( ARRAY$(I) )
```

The CHARACTER.SUBSTITUTION subprogram, in turn, looks like this[8]:

```
SUB CHARACTER.SUBSTITUTION( Q$ ) STATIC
    SHARED TRANSLATION.TABLE()
    FOR J = 1 TO LEN( Q$ )
        ASCII = ASC( MID$(Q$, J, 1) )
        MID$(Q$,J,1)=CHR$( TRANSLATION.TABLE( ASCII ) )
    NEXT J
END SUB
```

The result of this character-by-character translation is a set of pseudo records each of which may contain one or more ASCII nulls (character 000). For example, any blankspaces in the original record would have been converted to nulls (i.e. 032 → 000). It is necessary now to remove the nulls, since the sorting algorithm would otherwise treat them as letters 'earlier than' "A". To remove the nulls, we must do a snip and splice operation, where we save what is to the left of the null and to the right of the null, but discard the null itself.[9] We pass the string to the ELIMINATE.NULLS subprogram via a CALL statement:

```
CALL ELIMINATE.NULLS( ARRAY$(I) )
```

The ELIMINATE.NULLS subprogram looks like this (note: "NULL$" was defined prior to "MAIN:"):

```
SUB ELIMINATE.NULLS( Q$ ) STATIC
    SHARED NULL$
    SPOT = INSTR( Q$, NULL$ )
    DO WHILE SPOT <> 0
        Q$ = LEFT$(Q$, SPOT-1) + MID$(Q$, SPOT+1)
        SPOT = INSTR(SPOT, Q$, NULL$)
    LOOP
END SUB
```

If we perform a character-by-character translation, followed by a 'snip nulls' to the three records,

```
make
make work
makeshift
```

these subprograms would yield these pseudo records,

```
MAKE
MAKEWORK
MAKESHIFT
```

These records, in turn, would be alphabetized correctly (for example, in the manner of the *Oxford English Dictionary*) by any standard sorting algorithm. The original items (in lowercase and including blankspaces) could, then, at the final stage, be arranged in the same order as these stand-in pseudo records.

Phrase lexicons

The Translation Table might not, however, furnish all the conversions needed. Insofar as it provides a character-by-character translation, it is not sensitive to the *context* in which those characters occur. But sometimes the surrounding context makes a crucial difference. For example, for the purposes of alphabetizing, one may want to convert the letter "c" to "AA", if, for instance, the "c" occurs immediately after "M". That is, we want to convert "McCarthy" not to "MCCARTHY", as the Translation Table would do, but to "MAACARTHY". For these latter kinds of conversions, we will have to take recourse to what we might (for lack of a better name) call a "Phrase Lexicon". There are many such cases where a Phrase Lexicon is essential. For example, the Translation Table may have converted your original record which contained "28th" to "28TH", but what you really want for purposes of alphabetizing is "TWENTYEIGHTH". Similarly your data may contain "St.", and you may want this to be treated as "SAINT".[10]

One might think that all one has to do is to search the reformed records, in which, for example, "St." has been translated character-by-character to "ST", looking for "ST" and substituting "SAINT" wherever "ST" is found. This will not do. If you tried it this way, your original "style" would be converted to the unacceptable "SAINTYLE", and "mass transit" would be converted to "MASSAINTRANSIT". Similarly, if you thought it safe to substitute "MAA" for "MC" after a record had been reformed by the Translation Table, you would convert "camcorder" to "CAMAAORDER". Obviously, since important information about surrounding context is being discarded by the Translation Table, you want to have the substituting of 'phrases' occur *before* you call CHARACTER.SUBSTITUTION (see subroutine CREATE.PSEUDO.RECORDS).

You can set up a Phrase Lexicon as follows:

```
CREATE.PHRASE.LEXICON :
    RESTORE LEXICON
    N = 0
    A$ = ""
    DO WHILE A$ <> "END.PHRASES"
        READ A$,  B$
        IF A$ <> "END.PHRASES" THEN
            N = N + 1
            OLD.PHRASE$(N) = A$
            NEW.PHRASE$(N) = B$
        END IF
    LOOP
    NUMBER.OF.PHRASES = N
RETURN

LEXICON :
      DATA " 28th",    "TWENTYEIGHTH"
      DATA "2nd W W",  "SECONDWORLDWAR"
      DATA "St.",      "SAINT"
      DATA "æ",        "ae"
      DATA "'s",       ""
      DATA "Mc",       "MAA"
      DATA "MacG",     "MAAG"
                |
              etc.
                |
      DATA "END.PHRASES", "END.PHRASES"
```

Note the blankspace in the first DATA statement (between the left quotation mark and the "2"). If you were to omit that blankspace, the string "28th" would be found and replaced in the

string "128th", and you may not want it to be. And note that there is no blank between the second pair of quotation marks on the data entry for "'s", that is, "'s" is to be snipped out of any string in which it occurs.

To use the Phrase Lexicon, you proceed this way: search the string for each occurrence of the old phrase; as each is found, snip it from the string and replace it with the new phrase; continue searching and replacing to the right of each splice[11]; then repeat with the next old phrase, etc. ("_" signifies the continuation of the line.)

```
SUB PHRASE.SUBSTITUTION( Q$ ) STATIC
    SHARED NUMBER.OF.PHRASES
    SHARED OLD.PHRASE$(),  NEW.PHRASE$()
    FOR N = 1 TO NUMBER.OF.PHRASES
        A$ = OLD.PHRASE$(N)
        SPOT = INSTR(Q$, A$)
        DO WHILE SPOT <> 0
            Q$ = LEFT$(Q$, SPOT-1) + _
                NEW.PHRASE$(N) + MID$( Q$, SPOT + LEN(A$) )
            SPOT = INSTR(SPOT + LEN(NEW.PHRASE$(N)),Q$,A$)
        LOOP
    NEXT N
END SUB
```

If you will be alphabetizing many sets of data, needing a variety of Translation Tables and of Phrase Lexicons, you will want to write your program so that these tables can be constructed with a word processor and can be read into the program as companion files to the data files, rather than having the tables as permanent fixtures coded in DATA statements.

Results

What have we achieved to this point? *Table 2* shows incisively how different the results will be, applying, in the first instance, any standard sorting algorithm to a set of data, and applying, in the second, the program developed here.

The results of any standard sorting algorithm	The results of the above sorting program
“I’ve seen …	æsthetics
“Is there …	algorithm
“Isn’t it …	“Isn’t it …
“Shell” sort	“Is there …
2nd W W	“I’ve seen …
Mabbett, Alice	McCarthy, Richard
MacGregor, F.	MacGregor, F.
McCarthy, Richard	Mabbett, Alice
Mengleberg, H.	machine
Mæstro	Mæstro
algorithm	Mengleberg, H.
machine	mess hall
mess hall	2nd W W
zoology	“Shell” sort
æsthetics	zoology

Table 2

Case sensitivity

Suppose, finally, that you want to preserve the distinction between capitalized data and data in lowercase, that is, suppose you want to generate a “Names” listing followed by a “Subject” listing. The Translation Table, as discussed earlier, ignores *positions* of individual letters in the string: it translates all letters, including those in the first position, indiscriminately into capitals. To create a “Names” listing followed by a “Subject” listing, we will have to change both the Translation Table and the CHARACTER.SUBSTITUTION subprogram. First of all we will want the Translation Table *not* to convert case, e.g. lowercase “u” is to remain lowercase. Lowercase u-umlaut is to be translated as a lowercase “u”, while an uppercase U-umlaut is to be translated as an uppercase “U”. Once all the characters have been translated, we can then examine the first character of the string: if it is uppercase, we convert the entire string to uppercase, if it is lowercase, we convert the entire string to lowercase.

```
SUB CHARACTER.SUBSTITUTION( Q$ ) STATIC
    ' note: version #2 – modified to create
    ' a Names/Subject listing
    SHARED TRANSLATION.TABLE()
    FOR J = 1 TO LEN( Q$ )
        ASCII = ASC( MID$(Q$, J, 1) )
        MID$(Q$,J,1)=CHR$( TRANSLATION.TABLE( ASCII ) )
    NEXT J
    IF LEFT$( Q$, 1) => “a” THEN
                Q$ = LCASE$( Q$ )
        ELSE   Q$ = UCASE$( Q$ )
    END IF
END SUB
```

Letter-by-letter and Word-by-word sorting

Among professional indexers there has been a long-standing (and sometimes heated) debate about the relative merits of letter-by-letter and word-by-word sorting of data.[12] The difference may be readily illustrated:

Letter-by-letter:	Word-by-word:
`game`	`game`
`gamekeeper`	`game plan`
`game plan`	`game theory`
`gamete`	`gamekeeper`
`game theory`	`gamete`

To create letter-by-letter sorting, snip out blankspaces prior to sorting. That is, in the Translation Table, map the blankspace onto the null (i.e. 032 → 000). To create word-by-word sorting, map the blankspace onto a value greater than 000 but less than "A", e.g. map it onto itself (032 → 032).

Other kinds of data

For data which departs from the most commonplace, which, for example, contains both capitals and lowercase, or contains blankspaces, or contains foreign characters, or contains quotation marks, you cannot take a standard sorting routine off the shelf and expect to get results which mirror the conventions of ordinary dictionaries and the like. ASCII coding has severe limitations. For much ordinary data you will have to bully your data into a form suitable for sorting. A few techniques have been discussed here for some frequently occurring problems, but you will have to exercise some creative imagination for many other kinds of cases.

— Norman Swartz
Department of Philosophy
Simon Fraser University
Burnaby, B.C.
Canada V5A 1S6

Subroutines

```
READ.INPUT.DATA :
    OPEN "userfile.ext" FOR INPUT AS #1
    COUNT = 0
    DO WHILE EOF(1) = 0
        COUNT = COUNT + 1
        LINE INPUT #1,  ARRAY$(COUNT)
    LOOP
    CLOSE #1
RETURN
```

```
CONSTRUCT.TAG.TABLE :
    FOR N = 1 TO COUNT
        TAG(N) = N
    NEXT N
RETURN

SAVE.ORIGINAL.DATA :
    OPEN "savefile.$$$" FOR OUTPUT AS #1
    FOR N = 1 TO COUNT
        PRINT #1,  ARRAY$(N)
    NEXT N
    CLOSE #1
RETURN

CREATE.PHRASE.LEXICON :
     ' see text above for details
RETURN

CREATE.TRANSLATION.TABLE :
     ' see text above for details
RETURN

CREATE.PSEUDO.RECORDS :
    FOR I = 1 TO COUNT
         ' note: do not reverse the order of the following
         ' two lines.  See discussion in text.
        CALL PHRASE.SUBSTITUTION( ARRAY$(I) )
        CALL CHARACTER.SUBSTITUTION( ARRAY$(I) )
        CALL ELIMINATE.NULLS( ARRAY$(I) )
    NEXT I
RETURN

SORT.PSEUDO.RECORDS :  ' Shell sort used
    GUIDE = 1
    DO WHILE GUIDE < COUNT
        GUIDE = GUIDE * 2
    LOOP
    GUIDE = INT((GUIDE – 1) / 2)
    IF GUIDE = 0 THEN GUIDE = 1
    DO WHILE GUIDE > 0
        FOR I = 1 TO COUNT – GUIDE
            J = I
            DO WHILE J > 0
                K = J + GUIDE
                IF ARRAY$(J) > ARRAY$(K) THEN
                        SWAP ARRAY$(J), ARRAY$(K)
                        SWAP TAG(J), TAG(K)   ' see text
                        J = J – GUIDE
                    ELSE J = –1
                END IF
            LOOP
        NEXT I
        GUIDE = INT((GUIDE – 1) / 2)
    LOOP
RETURN
```

```
CREATE.REALLOCATION.TABLE :
    FOR N = 1 TO COUNT
        REALLOCATE( TAG(N) ) = N
    NEXT N
    ERASE TAG  ' discard the Tag Table
RETURN

READ.AND.SORT.SAVED.DATA :
    ERASE ARRAY$  ' erase the pseudo records
    OPEN "savefile.$$$" FOR INPUT AS #2
    FOR N = 1 TO COUNT
        LINE INPUT #2,  ARRAY$( REALLOCATE(N) )
    NEXT N
    CLOSE #2
RETURN

SAVE.SORTED.DATA :
    OPEN "sortdata.ext" FOR OUTPUT AS #1
    FOR N = 1 TO COUNT
        PRINT #1,  ARRAY$(N)
    NEXT N
    CLOSE #1
RETURN
```

Notes

1. ASCII code is standardized only for characters 000 through 127. In eight-bit bytes, another 128 characters may be defined, namely characters 128 through 255. Different manufacturers of micro computers have assigned different symbols to these so-called 'high-order' characters. Thus although characters 000-127 are identical on both the IBM PC and the Macintosh, the sets are remarkably different for the values 128 through 255.

2. Mainframe computers often use a different code, the EBCDIC (Extended Binary Coded Decimal Interchange Code). All our examples here, however, will assume ASCII coding.

3. The programming language used here for illustrative purposes is Microsoft's QuickBasic dialect of Basic. QuickBasic is used because it has superior string handling abilities, it is a structured language, it can handle strings up to 32K in length, and it is available for both IBM and Macintosh computers. The coding below, however, is so straightforward that it is easily convertible to Pascal, C, or any other programming language of choice.

4. There are dozens of books widely available illustrating many different sorting algorithms implemented in BASIC, PASCAL, C, etc. One such is Gabriel Cuellar's *Fancy Programming in IMB PC Basic*, Reston Computer Group, Reston, Virginia, 1984. In Chapter 3, Cuellar illustrates the bubble sort, the Shell sort, insertion sort, and the Quicksort, and provides comparative timing figures.

5. The SORT.EXE utility supplied with IBM/PC DOS 3.00 and later versions is case insensitive, for example, it will alphabetize "`Abba`" before "`acoustics`", and "`acoustics`" before "`Zamfir`".

6. The function UCASE$ was not available in IBM QuickBasic prior to version 4.0.

7. The initial position of the Translation Table is zero, not one, since the initial ASCII character is itself 000, not 001.

8. Persons programming in IBM QuickBasic can utilize a CALL to an assembly language subroutine XLATE in a library of programs, ProBas[tm], available from Hammerly Computer Services, 9309 Jasmine Court, Maryland 20707. The routine XLATE is at least three times faster than the BASIC coding in CHARACTER.SUBSTITUTION.

9. Warning: do not confuse the null character [CHR$(0)] with a string of null length. The two statements 'NULL$ = CHR$(0)' and ' NULL$ = "" ' are not equivalent.

10. You would have to write a pretty fancy subroutine if your data contains "`St.`" both as an abbreviation for "`Saint`" and for "`Street`", e.g. "`St. George St.`" (in Toronto). We will not attempt it here.

11. It is essential to safeguard against recursion at this point. Note in the coding for the subprogram PHRASE.SUBSTITUTION, the search – in the line immediately before LOOP – begins to the right of the inserted phrase.

12. See e.g. "On Sorting, Continued …", by Ruthanne Lowe, in *American Society of Indexers Newsletter*, no. 88 (Sept./Oct. 88), pp. 18-20.