Sorting

INTRODUCTION

Sorting, whether files or arrays, is a major operation in almost any computer application. Many uses of data require that the data are already in a sorted order, as we have seen with our control break programs. Furthermore, many applications are more useful or meaningful when the input data are sorted. A simple read/write report does not require that the input data be sorted. But if the report is a list of the finishers of the Boston Marathon, it will be much more useful if it is in sorted order.

Sorting requires arranging the individual elements of an array or file in order based on a specified field. If our list of Boston Marathon finishers is in order by their finishing time, we would say that time is the sort field, or key field. Some sorts use two or more key fields. The telephone book is sorted on last name. But when two or more people have the same last name, those people are sorted on first name. We would say that last name is the primary sort (or key) field and first name is the secondary sort field.

We will look at *array* sorting in this chapter because we have already discussed the means to manipulate array elements. Sorting records within a file uses similar algorithms, but the algorithms are applied to file manipulation techniques that we have not (and will not) cover in this text. Once you have a few sorts under your belt, it will not be difficult for you to apply them to any needed situation, including file sorts.

Because sorting is such a common and necessary application, there are many sort algorithms available and many means of accessing those algorithms. Sort utilities are frequently provided along with the operating system as part of a lease or purchase package. Some programming languages provide a sort command. And of course, we can design and code our own sort.

SORT UTILITIES

No matter what operating system you are using on a microcomputer, a mainframe, or anything in between, there is most likely at least one sort utility provided for your use. Exactly how you invoke that utility, what preparatory steps must be taken (i.e., the form the data must be in), and what sort algorithm will be used all depend on the utility. As a general rule, you will need to provide the utility with the name of the file to be sorted, the key field or fields on which it should be sorted, whether you want an ascending or descending sort, and where the result should go. Not all utilities will require each of these. Some utilities may assume an ascending sort, some may assume a standard output file, and some make make other assumptions.

Additional sort utilities may have been written "in house" or purchased separately from the operating system. Since some sort algorithms are more efficient on different kinds of data or different size lists or lists that begin in some partial order, it is reasonable to have multiple sort utilities. Each of these world be stored in ht e "library" of utilities available to programs. Much as you check a book out of a library for temporary use, you can *call* a sort utility from the subroutine library for temporary use by your program.

When we defined our flowchart symbols (light years ago), we defined a symbol called the External Process symbol. It is this symbol that we would use in a flowchart when calling a sort utility. In pseudocode we would continue to use the command DO but specify that it is an external module. External modules are not further defined within the design because the coder will not have to write them out.

The advantages of the utility is that it is already written, and we know it works so we don't have to worry about testing the sort. The disadvantage is that an all-purpose sort may not be the nest sort for our particular data. One-size-fits-all often means no one is fitted well.

From a design perspective there is not much to *using* a utility sort. You do need to know the requirements and limitations of the particular utility available. But you don't have to design the sort.

LANGUAGE SORTS

Some programming languages, in recognition of the importance of the sorting operation, include a sort command within the language. From a design perspective a SORT command can be treated much like a sort utility. We don't have to design the sort, but we do need to work within the requirements and limitations of the command. The languages that provide a sort command are primarily business, report-oriented languages like COBOL.

CODED SORTS

Because we're interested in designing algorithmic solutions, we are more interested in our own coded sorts than in predefined solutions like utility sorts or sort commands. You are probably asking yourself, "If I have a sort utility available, why would I *ever* want to go through the work of designing my own sort?' It's a reasonable question. It is possible, of course, that you won't have a sort utility available. But is also possible that the utility available is not the best one for the nature of your data. There are many, many algorithms available. Some sorts work best with small lists of data, some with very long lists. Some work best with data that are truly random; some are best with data that are skewed toward one extreme or the other; some are best with data that are already partially sorted. If you know something about your data and you also have a few sorts in your repertoire, you can probably design a more efficient sort for your particular need. If your program is going to be used repeatedly over a long period of time, the efficiency of the sort is important. If the program is going to be run once and then junked, it may not be worth the design and coding time required to include a custom sort, and the utility sort is a good compromise.

We will look at only two sorts, both of which are fairly simple (by comparison to other sort algorithms). Many of the sort algorithms, particularly those that work best for large files or for very skewed files, require a knowledge of other processing techniques or other ways to manipulate and store data that we have not discussed.

BUBBLE SORT

The bubble sort is frequently one of the first sorts taught to programming students. It requires repeated "passes" over the unsorted portion of the array. In each pass *each* adjoining pair of values is compared and switched if they are out of order. This process has the effect of moving the largest value in the array (or a portion of the array) to its appropriate position and "bubbling" the smaller values slowly up to the top. Since each pass puts one value (the largest value found) in its correct position, the unsorted portion of the array gets smaller by one element after each pass.

In the following algorithm *len* is the length of the array and *list* is the array itself. The flag *hadSwitch* is used to check if any pairs were out of order on the previous pass. If all the pairs were in order, then the array must be sorted and we can stop the sort. *PassCnt* is used to count the number of passes we make over the array checking for out of order pairs. At the most we will need *len-1* passes. *StopPt* marks the point in the array to stop checking the pairs during a given pass. Since the unsorted portion of the array keeps getting smaller, *stopPt* will keep moving up.