The Bubble Sort

In this exercise you will use the "bubble sort" algorithm to sort a list of numbers read from a file. You will also learn about "arrays," and how to read data from a file.

1. The Algorithm

The basic idea of the bubble sort is simple: you have your items to be sorted, such as a list of numbers, and you compare items which are next to each other and exchange them if they are out of order. You repeat the process over and over again until all items are in the proper order.

To be organized about it, you should start at the top of your list of numbers and compare the first two numbers. If they are out of order, exchange them. Then compare the second and third numbers, and exchange if necessary, then the third and fourth, and so on. At the end of one such "pass" through the list you will not have all numbers in order, but you will be closer to that goal. If you repeat such a "pass" through the list several times you will eventually have all the numbers in order.

Note that if you want to compare N numbers a pair at a time then you need only perform $N-1$ *comparisons* in a pass. Note also that on each pass a number which is not in it's proper position moves up the list by only *one* position. This is why the algorithm is called the "bubble" sort, because each number yet to be put in place slowly "bubbles up" toward it's final position. If you consider the case where the smallest number starts off at the end of the list you can easily convince yourself that to put all N numbers in their proper place you will need $N-1$ passes through the list.

2. Problem

Using the Bubble Sort algorithm just described, write a program which reads a list of numbers from a file into an array, sorts them (from lowest to highest), and then writes out the sorted list on the terminal.

The numbers you will read will be integers, and there will not be more than 2000 of them.

You should not use any algorithm other than the bubble sort, and you should not add any fancy improvements to the algorithm (that will come later).

Warning: the algorithm described in Etter's book, which uses the variable `SORTED`, is not the basic bubble sort, and so you should not copy this algorithm. Use the algorithm described above.

3. Input

Your program should read from the standard input (the terminal) the name of the file containing the numbers to be sorted. It should then open this file and read all the numbers to be sorted into an array. There will be one number per line. You should *not* echo the data you read from the file, but do remember to echo the file name.

The numbers you will read will be integers, and there will not be more than 2000 of them.

4. Output

Because there will be so many numbers in the sorted list it is important to organize the output. The sorted list of numbers should be written to the standard output with ten (10) numbers per line, using "`1X,I7`" to format each number.

In your sample run you should use the Unix "time" command to show how long your program takes to sort the list of numbers. You may want to try your program on different computers to see if you can detect a difference in speed.

5. Testing

To test your program you will need a file containing a list of numbers. You can copy such a file from one of my directories. The file `~myers/fortran/I256.d` contains 256 random integers, one per line. Also, you can create your own file containing a list of random numbers by copying the program `~myers/fortran/ranlist.f` to your own directory and compiling it and running it.

6. Reading

You will want to read in your book about "arrays" and how to use them.

You may find a discussion of the bubble sort in your book, or you may want to look at Section 5.2.2 of *Sorting and Searching* by Donald Knuth (this is Volume 3 of his series on *The Art of Computer Programming*), and in particular his Figure 14 may be helpful.

One way to get a good feeling for the algorithm is to write a short list of numbers on a chalk board or scratch paper and go through the steps "by hand."

You will need to read about how to read data from files rather than from the terminal. In particular, you will need to read about the `OPEN` and `CLOSE` statements, and learn what a "logical unit number" is and how it is used in a `READ` statement. You will also need to know how to detect when you are out of data. Pay particular attention to the `END=` clause of the `READ` statement.

You will need to read about `CHARACTER` variables in order to read the name of your data file. If you read the file name using a `FORMAT` statement (`A` format, in particular) then you will not need to enclose the name in quotation marks when you type it in.

To properly print your list of numbers you may find it useful to read about "implied" `DO` loops in a `PRINT` statement.

7. Fortran compiler options

As you may have noticed, Unix is rather terse. Commands are short and generally only lower case, as if designed for someone who doesn't type well. Unix commands are also generally silent unless something goes wrong. If you give a command to do something and you get nothing back but a command prompt, that usually means that your command was executed as you asked and nothing unusual was worth reporting.

Sometimes, however, you want more feedback, and you want to be able to type long commands with many options. Even then, there are some useful shortcuts.

For example, in this exercise you will be using Fortran arrays. The Fortran compiler normally does not check the array bounds. If you ask for the 407th element of an array for which only 100 elements were requested the program will gladly fetch for you the more or less random contents of the requested memory location, even though technically this is a serious error. But you can change this behavior by telling the compiler to warn you of such errors, with the command line option `-ffortran-bounds-check`, like so

```
% f77 -ffortran-bounds-check mavassar11.f
```

Now if you try to access an array element outside of the normal bounds (and remember, Fortran starts counting at one, not zero) then your program will terminate with an error message that should help you track down the problem.

Similarly, you might like to know when the compiler has implicitly set the type of a variable (one that you have not declared to be `REAL` or `INTEGER`). Often this will show you that you have made a typographical error. The command line option to request this is `-Wimplicit`, as in

```
% f77 -Wimplicit mavassar11.f
```

In fact, you might want to turn on all the warning messages you can, by using the compiler flag `-Wall`. Or you'd like to know if the code you are writing does not conform to the strict conventions of the Fortran standard, even though our compiler lets it pass by default. In that case, you could include the `-pedantic` flag in the `f77` command.

It may help you to know that all of these flags are used when your programs are tested for grading. The command used is something like

```
% f77 -pedantic -Wimplicit -ffortran-bounds-check -Wall mavassar11.f
```

If that fails, then the flags are removed and a simple `f77` command is used to try the program anyway, but the output from the more complicated compilation is used to see how well your program is written and to divine any hidden errors. So you may want to try compiling your assignment with all these options before you turn it in.

8. Unix command aliases

Typing the long compilation command given above is a lot of work, especially if you have to do it several times. Luckily Unix makes it easier, by letting you define your own aliases for commands. The syntax (using csh) is to give the command `alias` followed by the short-cut name you want to use, followed by the text of the shortcut. For example:

```
% alias f77-test f77 -pedantic -Wimplicit -ffortran-bounds-check -Wall
```

Then all you have to do to compile your program with all these options is give the much shorter command

```
% f77-test mavassar11.f
```

You can even put these definitions in the file `.cshrc` in your home directory so that they are automatically created whenever you log in or create a new command shell. In fact, you'll find that there are already some useful examples in your `.cshrc` file. You might want to try to figure out what the aliases for `ffind` or `psgrep` are supposed to do, and maybe even modify copies to do something equally useful.