ECE368 Programming Assignment #1

Due Friday, February 3, 2017, 11:59pm

Description:

This project is to be completed on your own. You will implement Shell sort using insertion sort and selection sort for sorting subarrays. You will use the following sequence for Shell sort:

$$\{1,2,3,4,6,\ldots,2^p3^q,\ldots\},$$

where every integer in the sequence is of the form 2^p3^q and is smaller than the size of the array to be sorted. Note that most of the integers in this sequence, except perhaps for some, can always be used to form a triangle, as shown in Lecture02. There may be incomplete rows of integers in the sequence below the triangle. For example, if there are 15 integers to be sorted, the corresponding sequence $\{1,2,3,4,6,9,8,12\}$ would be organized as follows, with an incomplete row containing the integers 8 and 12 in the sequence:

```
          1

       2     3

    4     6     9

 8   12
```

Functions you will have to write for `sorting.c`:

All mentioned functions and their support functions, if any, must reside in the program module `sorting.c`. The declarations of these functions should reside in the include file `sorting.h`. These functions should not call each other (unless you implement any such function as a recursive function). They should be called by the `main` function. `sorting.h` should not contain the declaration of other functions.

The first two functions `Load_From_File` and `Save_To_File`, are not for sorting, but are needed to transfer the `long` integers to be sorted from and to a file in **binary form**, respectively.

`long *Load_From_File(char *Filename, int *Size)`

The size of the binary file whose name is stored in the `char` array pointed to by `Filename` should determine the number of `long` integers in the file. The size of the **binary** file should be a multiple of `sizeof(long)`. You should allocate sufficient memory to store all `long` integers in the file into an array and assign to `*Size` the number of integers you have in the array. The function should return the address of the memory allocated for the `long` integers.

You may assume that all input files that we will use to evaluate your code will be of the correct format.

Note that we will not give you an input file that stores more than `INT_MAX` `long` integers (see `limits.h` for `INT_MAX`). If the input file is empty, an array of size 0 should still be created and `*Size` be assigned 0.

```
int Save_To_File(char *Filename, long *Array, int Size)
```

The function saves `Array` to an external file specified by `Filename` in **binary format**. The output file and the input file have the same format. The integer returned should be the number of `long` integers in the `Array` that have been successfully saved into the file.

If the size of the array is 0, an empty file should be created.

```
void Shell_Insertion_Sort(long *Array, int Size, double *N_Comp, double *N_Move)
```

```
void Shell_Selection_Sort(long *Array, int Size, double *N_Comp, double *N_Move)
```

Each of the two functions take in an `Array` of `long` integers and sort them. `Size` specifies the number of integers to be sorted, and `*N_Comp` and `*N_Move` should store the number of comparisons and the number of moves involving items in `Array` throughout the entire process of sorting. The function with the word "Insertion" in the name uses insertion sort to sort each subarray. The function with the word "Selection" in the name uses selection sort to sort each subarray.

You have to decide the order in which you use the integers in the sequence to perform Shell sort.

A comparison that involves an item in `Array`, e.g., `temp < Array[i]` or `Array[i] < temp`, corresponds to one comparison. A comparison that involves two items in `Array`, e.g., `Array[i] < Array[i-1]`, also corresponds to one comparison. A move is defined in a similar fashion. Therefore, a swap, which involves `temp = Array[i]; Array[i] = Array[j]; Array[j] = temp`, corresponds to three moves. Also note that a `memcpy` or `memmove` call involving *n* elements incurs *n* moves.

```
int Print_Seq(char *Filename, int Size)
```

For an array whose number of integers to be sorted is `Size`, this function prints the sequence to be used in Shell sort as defined earlier in the file whose name is stored in a `char` array, the address of which is stored in `Filename`. Each integer in the sequence should be printed to the file using the format `"%d\n"`. In other words, the output file is a **text file**. The sequence should be printed in the order from top to bottom, and from left to right in each row. For example, for the sequence given earlier, the output file should look like:

```
1
2
3
4
6
```

```
9
8
12
```

There should be only one integer per line. The function returns the number of integers in the sequence.

Note that it is not necessary to call this function before performing any form of Shell sort. This function should not be called by any of the Shell sort functions. It should only be called by the `main` function.

Also, note that when `Size` is 0 or 1, the output file should be empty because the sequence should be empty, and the return value should be 0.

You are allowed to write other helper functions in `sorting.c`. For any of these helper functions, the name should not start with an underscore '_' because the functions that the instructor writes to test your code will start with an underscore '_'. These helper functions should be declared and defined in `sorting.c`. It is best that these helper functions be declared as static functions.

Function `main`:

You have to write another file called `sorting_main.c` that would contain the `main` function to invoke the functions in `sorting.c`. You should be able to compile `sorting_main.c` and `sorting.c` with the following command (with an optimization flag -O3):

```
gcc -Werror -Wall -Wshadow -O3 sorting.c sorting_main.c -o proj1
```

If you want to debug your program, you should have the flag "-g" as well. When the following command is issued,

```
./proj1 i input.b seq.t output.b
```

the program should read in `input.b` to store the list of integers to be sorted in an array, run Shell sort with insertion sort on the array, print the sequence to file `seq.t`, and save the sorted integers in the array in `output.b`. For the purpose of this assignment, the suffix `b` indicates that the file is a binary file and the suffix `t` indicates that the file is a text file.

The program should also print to the standard output (i.e., a screen dump), the following information:

```
Number of long integers read: AAAA
Number of long integers stored: BBBB
Length of sequence: CCCC
Number of comparisons: DDDD
Number of moves: EEEE
I/O time: FFFF
Sorting time: GGGG
```

where `AAAA`, `BBBB`, and `CCCC`, all in `%d` format, and `DDDD`, `EEEE`, `FFFF`, and `GGGG`, all in `%le` format, report the statistics you have collected in your program. `AAAA` refers to the number of `long` integers loaded from the input file, BBBB refers to the number of long sorted integers saved to the output file, adn CCCC refers to the number of integers used in the sequence for Shell sort. DDDD and EEEE refer to *N_Comp and *N_Move, respectively. FFFF should report the time it takes to read from input.txt and to print to seq.txt and output.txt. GGGG should report the time it takes to sort.

The function that you may use to keep track of I/O time and Sorting time is clock(), which returns the number of clock ticks (of type clock_t). You can call clock() at two different locations of the program. The difference gives you the number of clock ticks that pass by between the two calls. You would have to divide the difference by CLOCKS_PER_SECOND to get the elapsed time in seconds. There are typically 1 million clock ticks in one second.

The following command will use Shell sorting with selection sort:

```
./proj1 s input.b seq.t output.b
```

Report you will have to write:

You should write a (brief, at most a page) report that contains the following items:

- An analysis of the time- and space-complexity of your algorithm to generate the sequence (not sorting).
- A tabulation of the run-time, number of comparisons, and number of moves obtained from running your code on some sample input files. You should comment on how the run-time, number of comparisons, and number of moves grow as the problem size increases, i.e., the time complexity of your routines.
- A summary of the space complexity of your sorting routines, i.e., the complexity of the additional memory required by your routines.

Each report should not be longer than 1 page and should be in PDF, postscript, or plain text format (using the textbox in the submission window). The report will account for 10% of the overall grade of this project.

Please note that Shell sort with selection sort will be quite inefficient if you do not optimize the selection sort. You may not be able to run your program on the largest test case. You should include in your report the reason(s) for the inefficiency of Shell sort with selection sort if your implementation is inefficient.

Submission and Grading:

The project requires the submission (electronically) of a zip file called proj1.zip. The zip file should contain the programs sorting.c and sorting_main.c and the header file sorting.h you have created through Blackboard. If the zip file contains a makefile, we will use that file to make your executable. The zip file should also contain the report.

The grade depends on the correctness of your program, the efficiency of your program, the clarity of your program documentation and report.

The two sorting functions will account for at least 50% and at most 70% of the entire grade. The other functions and the report will account for the remainder of the grade.

It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Memory leak will result in at least 50% penalty.

Given:

We provide sample input files (in `proj1_samples.zip`) for you to evaluate the runtimes, and numbers of comparisons and moves of your sorting algorithms. All ".b" files are binary files and all ".t" files are text files. The number in the name refers to the number of `long` integers the file is associated with. For example, `15.b` contains 15 `long` integers, `15s.b` contains 15 sorted `long` integers from `15.b`, `15_seq.t` contains the sequence used for sorting. In particular, `15s.b` and `15_seq.t` are created by `proj1` by the following command:

```
./proj1 i 15.b 15_seq.t 15s.b
```

My implementation of `proj1` prints the following output to the screen when the above command is issued:

```
Number of long integers read: 15
Number of long integers stored: 15
Length of sequence: 8
Number of comparisons: 6.800000e+01
Number of moves: 6.400000e+01
I/O time: 3.650000e-04
Sorting time: 9.000000e-06
```

My implementation of `proj1` also created `1000s.b` and `1000_seq.t` from `1000.b`. However, for the input files `10000.b`, `100000.b`, and `1000000.b`, the output files of my implementation of `proj1` are not included.

Your implementation should not try to match the number of comparisons and number of moves that my implementation reported. That is not the purpose of the assignment.

Getting started:

Copy over the files from the Blackboard website. Any updates to these instructions will be announced through Blackboard.

Given that the input files are in binary format, you probably want to write some helper functions to print the array of `long` integers before and after sorting for debugging purpose.

You also have to ask the question of whether you have performed (Shell) sorting correctly. If the array of `long` integers is in ascending order after sorting, have you sorted correctly?

Start sorting!