Lab 3: Sorting (for use in Project 1)

CPSC 1021 – Spring 2018

CLEMSON School of COMPUTING

Due Date: Friday, February 9, 2018 @ Midnight

Lab Objective

- Implement two basic sorting algorithms
- Process data of indeterminate length
- Use work with dynamic strings in C (sprintf())

Introduction

Sorting a list of numbers is a classical problem in computer science. Many different **algorithms** (a set of procedures we follow to solve some problem) have been developed to address sorting, each with different **strengths** and **weaknesses**. But why is sorting interesting, and why should you care? As an example, try to sort the following list in your head:

4 1 8 2 5

You were probably able to produce the sorted sequence, 1 2 4 5 8 in **almost no time at all**. From psychology, we know that the human brain is particularly good at identifying patterns and imposing order on the world around us. However, the **number** of items we can process in this way is surprisingly small: 7 plus or minus 2.

That means that if I were to give you another list:

4 1 -2 5 1 -13 7 9 3 8 11 -4 32

It will be noticeably more difficult to sort. We've effectively maxed out your working memory (brain RAM!). To sort this list, you probably tried to **identify the smallest number** (-13) and then searched for numbers of increasing value. How many times did you **scan** your head back and forth across the list to find the next number in the sequence? How many times did you have to check and make sure you didn't miss a number or negative sign? Did you notice right away that there are two 1's?

This process of scanning the list for numbers of increasing value has a name: selection sort. It is a simple and terribly **slow** algorithm. Our brains just aren't built to sort long lists of numbers… but computers are. In this lab we'll implement two other sorting algorithms (which will be useful in your project).

These will also be slow, but better than selection sort!

Resources

Bubble Sort: https://en.wikipedia.org/wiki/Bubble_sort

Comb Sort: https://en.wikipedia.org/wiki/Comb_sort

sprintf: http://www.cplusplus.com/reference/cstdio/sprintf/

fseek: http://www.cplusplus.com/reference/cstdio/fseek/

Assignment

Download the starter code from Canvas, which includes:

- **sorting.h:** declarations for sorting functions
- **large.dat:** a file containing many unsorted numbers
- **small.dat:** a file containing few unsorted numbers

Create **2** new files: **sorting.c** and **lab3.c**

Your goals are:

- **sorting.c:** implement the functions declared in sorting.h
  - both sorts should sort in **ascending order**, or least to greatest
  - there is no bound on how many numbers the passed array could contain
  - **note that bubble and comb sort are only about 8-12 lines of code**
- **lab3.c:** contains a main() function which should
  - take 2 command line arguments as input
    - the first is the name of the file to sort (i.e. large.dat)
    - the second is the name of the sort to use {bubble, comb}
  - read in the data from the input file into an integer array
  - use the correct sorting algorithm to sort the integers
  - print the sorted list to an output file
    - the file **must** be named as follows**: <name of sort>_<# of ints>.dat**

For example, if given the input file below (newline after each number):

```
12
3
4
7
1
```

And your program was executed as follows:

```
./SORT small.dat comb
```

I expect the output file to be named **comb_5.dat** and it should look like (newline after each number):

```
1
3
4
7
12
```

Bubble vs. Comb

As stated in the resource links, both Bubble and Comb sort work by scanning from left → right in a list and swapping large elements with their smaller neighbors. The sorting stops when you pass through the whole array without making any swaps at all.

This should be a hint that you will need to **keep track of how many swaps you perform each pass**, stopping once the swap count is 0 (a for loop nested inside a while loop!).

Comb sort is almost identical to bubble sort, but instead of comparing adjacent elements to each other in the list, we start with a larger comparison gap. This allows us to move large numbers (sometimes called rabbits) in big leaps towards the right end. Slower moving medium range numbers (turtles) won't be dealt with until we decrease the gap size. The Wikipedia article is pretty good, as is the pseudo code.

In future classes, you will talk about time complexity to measure how fast an algorithm truly is. Instead of going there today, I'll simply note that for a list of 500 random numbers, my bubble sort required **65,000** swaps while the comb sort required **2000**. That's approx. **33x** less swaps for the comb sort!

Reading a file of indeterminate length:

You might have noticed that I did not specify how many integers the files will contain. Your program should be able to handle any input file which is formatted correctly. To achieve this, we need to first scan the file and **count** the number of integers it contains. I'll give you the code to accomplish this:

```
// Count numbers in file
int data_size = 0;
int num;
while (fscanf(input, "%d ", &num) == 1) {
  data_size++;
}
```

After this code executes, we will know how much memory we need to store all the integers in the file.

The trick is, how can we reread the file? Using the **fseek** function, you can reset a file pointer to begin reading from the file again. See the link above. This time, you should read the numbers into an array which we can pass to our sorting functions.

Calling Your Sort and Writing Output

The links in the resources above do an excellent job of explaining how Bubble Sort/Comb Sort work. Please review them carefully (notice that there's even pseudo-code for both algorithms). Use the strcmp() function as seen in Lab 2 to determine which sort your program should use, either bubble or comb. **I will only ever call your program with bubble or comb as arguments!**

```
./SORT small.dat bubble
./SORT large.dat comb
```

Finally, you need to write the output to a custom named output file.

You can safely assume that no output file name will be **longer than 50 characters**. Use the sprintf() function from the resources to insert the data_size counter into the file name. Again, if the file contained 432 integers and was sorted using comb sort, the output file name should be:

```
comb_432.dat
```

Tips and Tricks

Really this is just an excuse for you to get a head start on your project! Sorting algorithms, especially the **fast** ones, will be a big topic of discussion in later courses, so it's also good that we are covering them now.

For comb sort, you need to calculate an integer gap by dividing by the floating point shrink value. Remember that in C, if I divide an integer by a float, any value after the decimal point is removed. This is great because it will work as the "floor" function described in the comb pseudo code. However, we never want the gap to be 0, so we can make use of the **ternary operator** to bound the gap value to be at least 1:

```
int gap = …
double shrink = …
…
gap = gap / shrink;
gap = gap < 1 ? 1 : gap;
```

See this link for more info on the ternary operator, it is very useful.

I have provided two **Python** scripts along with the source starter code which can be used to generate lists of random integers as well as check a given file to determine if it is sorted or not.

- i**s_sorted.py:** check a given file to see if it is sorted or not

```
python is_sorted.py <file_name>
```

- **make_randoms.py:** generate a list of random numbers

```
python make_randoms.py <output_name> <# of integers>
```

Use these scripts if you'd like to make additional input files for testing or to validate your results for large data sets. Feel free to look at the code if you'd like, they're just simple Python scripts.

Submission

Submit all C source files: **lab3.c, sorting.c, sorting.h**

Do not submit any other files.

Compile and Execute

Use GCC to compile your code as follows:

```
gcc lab3.c sorting.c -Wall -o SORT
```

Execute the program

```
./SORT small.dat bubble
```

Formatting

Your program should be well documented!

1. Each file should include a header.

2. Your program should consist of proper and consistent indention
3. No lines of code should be more than 80 characters

5 – 10 points will be deducted for each of the above formatting infractions.

Submission Instructions

- Test your program on the School of Computing server prior to submitting.
- Use the tar utility to tar.gz all source files. **Do not tar an entire directory!**
  When I decompress your archive, I should see all of the files you included, not a top level directory!
  Failure to correctly tar may result in up to a 25 point penalty!

```
tar -czvf nglyder-lab3.tar.gz *.c *.h
```

- Name your tarred file **<username>-lab<#>.tar.gz** (ex. nglyder-lab3.tar.gz)
- Use handin (http://handin.cs.clemson.edu) to submit your archive