ECE 368: Project 1

This project requires a milestone and a final submission
The deadlines will be 11:59pm ET on the due dates
See the syllabus for due dates

- If you are wondering sample performance numbers see here.
- Performance leaderboard: compare your performance numbers with others! See here
- **Big data challenge** (optional) here. See more details below.
- Backup your code often! "Deleting my code by mistake" does not warrant a late submission.

Description:
This project is to be completed on your own. The goal of this project is to understand how Shell Sort improves the performance of Insertion Sort and to apply a similar optimization to Bubble Sort. In order to successfully complete this project, you will need to implement and compare the performance of Shell Sort (using Insertion Sort) and your Improved Sort (using Bubble Sort).

As we saw in the lecture, Shell Sort creates logical sub-arrays of elements that are spaced "*gap*" positions from each other in the given array. It then sorts each of these sub-arrays using Insertion Sort. The entire procedure is repeated using smaller and smaller values of "*gap*" till eventually `gap=1`, which will result in a sorted array. The chosen values of *gap* (called the *gap sequence*) are a very important aspect of Shell Sort. For this project, you will use the following sequence for implementing Shell sort:

- $\{2^{p'}3^{q'}, \ldots, 2^{p}3^{q}, \ldots, 16, 12, 9, 8, 6, 4, 3, 2, 1\}$, where $2^{p'}3^{q'}$ is the largest integer smaller than `N` (the size of the array to be sorted). We shall refer to this sequence as `Seq1`.
  Think about how to generate `Seq1`. See here for more details. (mirror)

You can envision a similar modification to improve Bubble Sort. Your task is to design and implement this modification. For your improved version of Bubble Sort, use the following gap sequence:

- $\{N_1=N/1.3,\ N_2=N_1/1.3,\ N_3=N_2/1.3, \ldots, 1\}$, where `N` is the size of the array to be sorted. In other words, the next value in the gap sequence is the **floor** value of the previous integer gap divided by 1.3. However, if any value in this sequence becomes 9 or 10, it is replaced by a value of 11. We shall refer to this sequence as `Seq2`. Note that since the gap value has to be an integer, you will need to take the floor of the gap value computed above.

Functions you will have to write:
All the functions mentioned below and their support functions, if any, must reside in a program file named `sorting.c`. The first two functions *Load_File* and *Save_File*, are not for sorting, but are needed to transfer the integers to be sorted to and from the disk.

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

The input file contains $N+1$ integers, one in each line. The first line of the file indicates the number of integers to be sorted, *i.e.,* N. `*Size` should be assigned the value of N at the end of the routine. The subsequent lines contain (long) integers.

```
int Save_File (char *Filename, long *Array, int Size)
```
The function saves `Array` to an external file specified by `Filename`. The output file should have the same format as the input file. Return the count of integers that have been successfully saved; that is, if everything goes well, the return value should equal the argument Size.

```
void Shell_Insertion_Sort (long *Array, int Size, double *NComp, double *NMove)

void Improved_Bubble_Sort (long *Array, int Size, double *NComp, double *NMove)
```

These functions take in an array of long integers (stored in `Array`) and sort them in **ascending order**. `Size` specifies the number of integers to be sorted, and `*NComp` and `*NMove` contain the number of comparisons and the number of moves involving items in `Array`. The Shell Insertion Sort function should use the sequence `Seq1`, and the Improved Bubble Sort should use the sequence `Seq2`.

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.

```
void Save_Seq1 (char *Filename, int N)
```

You implement this function to allow the grader to check whether you can generate the right `Seq1`.
Your code does not have to directly call this function.

The function should generate `Seq1` that is identical to the one used in your sorting code, and write `Seq1`, in ascending order, to an external file specified by `Filename`. The format of the file should follow what has been used in Save_File(). `N` is the size of the array to be sorted, NOT the size of `Seq1`.

```
void Save_Seq2 (char *Filename, int N)
```
Same as the above function, but for `Seq2`.

Big data challenge with large input (optional):

For fun and to compete for a position in the "ECE368 Hall of fame", consider to take the challenge in which we will stress your code with very large input, e.g. a few hundred MB.

This will NOT be counted into the project grade. So finish basic requirements first.

To participate, sign up here.

Submission
We use the ECN shay server for submission. To log in,

```
ssh userid@shay.ecn.purdue.edu
```

Milestone1

Write a few paragraphs (around 400 words in total) in a .**pdf** file to explain your overall approach to the problem. Note that the course name is 'ece368', NOT 'ee368'.

```
turnin -c ece368 -p proj1ms1 ms1.pdf
```

Final submission

The project requires the submission (electronically) of three files, `sorting.c`, `sorting.h`, and `report.pdf` using the `turnin` command invoked as follows:

```
turnin -c ece368 -p proj1 sorting.c sorting.h report.pdf
```

Report:

You should write a report that contains the following items:

1. A concise description of your algorithms, the structure of your program, and any optimizations that you have performed.
2. An analysis of the time- and space- complexity of your algorithm to generate the two sequences.
3. A tabulation of the run-time, number of comparisons, and number of moves obtained from running your code on a few sample input files for both Shell Sort and Improved Bubble Sort. You should comment on how the run-time, number of comparisons, and number of moves appear to grow as the problem size increases.
4. A summary of the space complexity of your sorting routines, i.e., the complexity of the additional memory required by your routines.

The report should not be longer than 1 page and should be in PDF format only.

We will grade your program only on the machine `shay.ecn.purdue.edu`. Your total grade depends on the correctness of your program, the efficiency of your program, and the clarity of your program documentation and report.

Grading rubrics (tentative, to be finalized soon)

Correctness (75 points):

- Program doesn't crash on any inputs, up to the max number of 100,000 numbers. (10 test cases, 1.5 points each)
- Gives the correct output. (10 test cases, 6 points each)

Performance (10 points):

- Speed (5 points): Is the speed within 120% of the provided binary.
  - If the provided binary takes 10 seconds, yours must run in 12 seconds. This will be tested on the largest test case
- Memory usage (5 points): Is memory usage within 120% of the provided binary.
  - If the provided binary uses 1,000 bytes on the program, yours can take a maximum of 1,200.

- Will be tested using Valgrind total heap usage. This will be tested on the largest test case.

Report (15 points):

- PDF format (5 points)
- Clarity and quality of the report (10 points)

Given to you:
We provide you the header file `sorting.h`. We also provide a main function in `sorting_main.c` that allows you to begin coding the sorting algorithms and other above-mentioned functions. You are not allowed to modify the file `sorting_main.c`. Create a new file, `sorting.c`, that will contain the functions you write. To compile, use the following command (add optimization flag `-O3`)[1]:

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

We also provide sample input files for you to evaluate the runtimes, and the number of comparisons and moves of your sorting algorithms. We also provide an executable `random_file` that can generate a file of randomly generated numbers. To generate a sample input file, use this command:

```
random_file output_filename num_of_integers random_seed
```

Getting started:
Download the relevant files from Shay:

```
git clone https://github.com/fxlin/ece368-proj1.git
```

Monitor the syllabus regularly for any updates to these instructions. Use TA/instructor office hours and Piazza for any general questions or clarifications. *Have fun!*

[1] If you use gcc>4.4 (which is NOT the case of Shay), you may need to add -Wunused-result to suppress warnings.