CSC 301 Project 1: Sorting Fall 2025

Code Due: Friday, September 26th

Report Due: Friday, October 3rd

In this project, you will implement a number of sorting algorithms that we have discussed in class. You will be required to implement your solutions in their respective locations in the provided `project1.cpp` file. **Please do not change the name of any provided files.**

Contents

1 Pair Programming

You are required to work in assigned pairs for this project.

- You should submit only one version of your final code and report. Have one partner upload the code and report through Canvas and associate both partners with the submission. Make sure that both partner's names are clearly indicated at the top of all files.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating').
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

2 Readable Code

You are expected to produce high quality code for this project, so the code you turn in must be well commented and readable. A reasonable user ought to be able to read through your provided code and be able to understand what it is you have done, and how your functions work. For these sorting algorithms, this means that a grader should be able to read over your code and tell that your algorithms are implemented correctly.

The guidelines for style in this project:

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code.

3 Getting Setup

3.1 Short C++ Guide and Documentation

A useful quick guide to C++, told with some comparissons to C, is from learn x in y minutes (https://learnxinyminutes.com/docs/c++). You will also find extensive documentation at the C Library Reference (https://cplusplus.com/reference/clibrary).

3.2 Running on MathLAN

All of the provided code will run on MathLAN, so I would encourage you to use that resource for this project in particular. Some of the calculations performed require a call to `lapack`, a C wrapper of an old Fortran library for linear algebra. Meanwhile, the plotting code written in python requires `matplotlib`. Both of these libraries would need to be installed and properly linked on a personal computer.

Useful guides to getting setup on MathLAN can be found at:

- Professor Perlmutter's CSC 213 Page (https://perlmutter.cs.grinnell.edu/teaching/2023F/CSC_213_02/reference/). You should follow the relevant link(s) under the "Remote Development" header.

On MathLAN, place the files `project1.cpp`, `project1.hpp`, `project1tests.cpp`, `project1tests.hpp`, and `Makefile` in the same directory. You can compile the code with the '`make`' command, run the tests with '`./project1tests`', and run the python code by calling '`python project1plotting.py`'.

4 Problem Statement

You will implement five algorithms for sorting an array. We define this sorting task as:

- The input array will have a positive integer length.
  - Will not be asked to sort an empty array.
  - Could be asked to sort a singleton array.
- The elements need not be distinct (i.e. repeats allowed).
- The elements can be any (positive or negative) real number.

The goal is to sort the elements from the input array into ascending order so that the smallest element is at index 0, i.e.,

$$A[0] \leq A[1] \leq \cdots \leq A[k-1] \leq A[k] \leq A[k+1] \leq \cdots \leq A[end]$$

You will implement the following five algorithms as discussed in class:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort

A large component of your grade for this project will be the correctness of these algorithms.

4.1 Runtime Comparisons

In the provided code is the function `measureTime`. This function can be used to time your implementations and ultimately obtain plots of the runtime versus input size. The provided `main` function in `project1tests.cpp` will run this `measureTime` function for each of your sorting algorithms, and for each, it will:

- Consider a range of increasing n values, where n is the length of the array to sort.
- Run 30 trials for each n value, timing each trial, and averaging over the trials at the end of computation.
- Write the n values and associated average runtimes to a specified file.
- In some cases, it will also calculate the log-log slope of the runtime vs n plot (more on this below).
- For all algorithms, it will do this process for both unsorted and sorted input. Recall that several of these algorithms behave differently for unsorted and sorted input.

4.2 Log-Log Runtime

The function `measureTime` considers a plot of $\log(runtime)$ vs $\log n$, and attempts to fit a line to these log-log plots, outputting the fitted slope.

Looking at log-log plots of the runtime versus the input size is a very common method used to interpret the runtime of a polynomial time algorithm. To understand why we do this, consider the following hypothetical:

Let's assume that we have an algorithm that runs in $\Theta(n^k)$ time, where k is some positive integer. Once we have implemented this algorithm, how can we determine if it really has obtained the promised runtime complexity?

The claim that the runtime is $\Theta(n^k)$ means that

$$T(n) \sim n^k \quad \Longrightarrow \quad T(n) = a\, n^k, \quad a \in \mathbb{R}.$$

Now take the log of both sides of this equation. This gives us

$$\log T = \log(a\, n^k) = \log(n^k) + \log a = k \log n + \log a.$$

$$\Longrightarrow \quad \log T = k \log n + \log a.$$

So if we set $y = \log T$ and $x = \log n$, with $\log a = c$ because it is just a constant, we get an equation of the form

$$y = kx + c.$$

When we plot $\log T$ versus $\log n$, we should see a straight line! Moreover the slope of that line should be the value of k.

This means that if you want to verify that an algorithm runs in $\Theta(n^2)$ time, you should plot the runtime versus the input size on a log-log scale. If you fit the resulting log-log plot to a straight line, the slope should be 2.

4.3 Questions

Along with your code, you should submit a short report (2-3 pages *not counting figures*) that addresses the following questions. Your report can include any of the generated plots that *you deem relevant*. You are not required to include any of the images. Note that you should not directly answer these questions one at a time, but rather your report should discuss their answers with details/evidence obtained from the results of running your code.

Make sure to discuss all of the questions.

- Do your algorithms behave as expected for both unsorted and sorted input arrays?
- Which sorting algorithm was the best (in your opinion)? Which was the worst? Why do you think that is?

- Why do we report theoretical runtimes for asymptotically large values of n?
- What happens to the runtime for smaller values of n? Why do you think this is?
- Why do we average the runtime across multiple trials? What happens if you use only one trial?
- What happens if you time your code while performing a computationally expensive task in the background (i.e., opening an internet browser during execution)?
- Why do we analyze theoretical runtimes for algorithms instead of implementing them and reporting actual/experimental runtimes? Are there times when theoretical runtimes provide more useful comparisons? Are their times when experimental runtimes provide more useful comparisons?

5 Provided Code

The file `project1.cpp` (and its associated `.hpp`) have pre-defined functions listed below:

- `selectionSort(arrayToSort)`: to contain your implementation for Selection Sort.
- `insertionSort(arrayToSort)`: to contain your implementation for Insertion Sort.
- `bubbleSort(arrayToSort)`: to contain your implementation for Bubble Sort.
- `mergeSort(arrayToSort)`: to contain your implementation for Merge Sort.
- `quickSortHelper(arrayToSort, i, j)`: to contain your implementation for Quick Sort. Note that the values of the indices `i` and `j` are present to help you recursively call the function on the partitions you create (so it will be in-place). Note that `j` must point *one past* the end of the array. All testing with this function will be done with the call `quickSort(arrayToSort)`.
- `quickSort(arrayToSort)`: this is a fully implemented function that calls `quickSortHelper(arrayToSort, 0, arrayToSort.size())`.

The following functions are found in `project1tests.cpp` (and its associated `.hpp`):

- `testingSuite`: runs a number of tests on the input algorithm. This is not an exhaustive list of tests by any means, but covers the edge cases for your sorting algorithms. This function will print info about passed/failed tests.
- `calcLogLogSlope`: used by the `measureTime` function to calculate the slope of the log-log plots of runtime vs n. It makes a call to the Lapack library, which is a C wrapper for a well known linear algebra library written in very old Fortran '77.
- `measureTime`: runs the input algorithm on randomized/sorted arrays of varying sizes while tracking the runtime. It writes the runtime and n data to a specified file. It can also print info about the log-log slope.

The plotting script is in the file `project1plotting.py`. A number of `.dat` files will get generated once you have run the provided `project1tests`. The python script will read those files and create and save a series of plots of runtime vs n on both linear and log-log scales.

You are welcome to change anything in the `main` function of your local version of the testing code if you think it will help with the images generated for your report.

You have been provided with a `Makefile` which will compile the code correctly on MathLAN using the `make` command. The code can be cleaned (i.e., the executables deleted) with the command `make clean`. If you are working on a personal machine, you may need to edit this file to properly link the required lapack libraries. You can learn more about makefiles in the GNU guide (https://www.gnu.org/software/make/manual/make.html). Good luck.

6 Compiling and Running the Code

As mentioned before, you can compile the code with the `make` command, and can clean with the command `make clean`.

Once you have compiled the code, you can run it with the command `./project1tests`. This will call the provided testing code, print the results of the tests, and generate the various runtime plots.

7 Submission

You **must** submit your `project1.cpp` and `project1.hpp` code and your report online on the course Gradescope. Since you are working with a partner, only submit one version of your completed project, associate both partners with the submission, and indicate clearly the names of both partners at the top of all submitted files. Attribute help in the `.cpp` header.