ECE 2400 Computer Systems Programming, Fall 2017
Programming Assignment 3: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2017-10-14-07-54

1. Introduction

The third programming assignment is designed to give you experience working across four important sorting *algorithms* in computer systems programming: insertion sort, selection sort, merge sort, and quick sort. In this assignment, you will leverage many of the concepts from lecture including types, pointers, arrays, and complexity analysis. Algorithms together with data structures provide the basis of all software programs. In particular, learning to analyze and compare different algorithms that solve the same class of problems is a fundamental skill and will be tremendously useful as you continue to work with software programs.

Sorting algorithms are a particularly useful class of algorithms, and although we focus on integer sorting in this assignment, the same algorithms can often be used to sort other types of elements as well. Sorting the elements in a container can often simplify and reduce the complexity of future operations (e.g., being able to run binary search on a sorted vector), and therefore, sorting algorithms have been intensely studied to maximize their performance and work efficiency. In this assignment, you will implement four functions corresponding to four sorting algorithms: *insertion sort*, *selection sort*, *merge sort*, and *quick sort*. You will evaluate the impact of scaling the problem size (i.e., the number of integers to sort) as well as the impact of different patterns of inputs (e.g., random, almost sorted, reversed) on the performance of your implementations. This will provide you enough data to demonstrate the impact of algorithmic changes on performance and work efficiency of an implementation. As in the previous assignments, design your code to be both *maintainable* and *robust*. We will continue to leverage the Criterion framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your sorting implementations are functional and verified, you will write a 2–4 page design report that describes your design of each implementation, discusses your testing strategy, and evaluates the performance and other trade-offs between the four algorithms. **The final code and report are due at the end of the assignment, but for this assignment you must also meet an incremental milestone of pushing your implementations of the two simpler sorting algorithms, insertion sort and selection sort, to GitHub on the date specified by the instructors. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed.**

This handout assumes that you have read and understand the course tutorials. To get started, log in to an `ecelinux` machine, source the setup script, and clone your individual repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/<netid>
```

You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository. If you have already cloned your individual remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece2400/<netid>
% git pull --rebase
% mkdir -p pa3-algo/build
% cd pa3-algo/build
% ../configure
% make check
```

All of the tests should fail since you have not implemented the programming assignment yet. For this assignment, you will work in the `pa3-algo` subproject, which includes the following files:

- `configure` – Configuration script to generate Makefile
- `Makefile.in` – Makefile template to run tests and eval
- `src/insertion-sort.h` – Header file for `insertion_sort`
- `src/insertion-sort.c` – Source code for `insertion_sort`
- `src/insertion-sort-test.c` – Test cases for `insertion_sort`
- `src/selection-sort.h` – Header file for `selection_sort`
- `src/selection-sort.c` – Source code for `selection_sort`
- `src/selection-sort-test.c` – Test cases for `selection_sort`
- `src/merge-sort.h` – Header file for `merge_sort`
- `src/merge-sort.c` – Source code for `merge_sort`
- `src/merge-sort-test.c` – Test cases for `merge_sort`
- `src/quick-sort.h` – Header file for `quick_sort`
- `src/quick-sort.c` – Source code for `quick_sort`
- `src/quick-sort-test.c` – Test cases for `quick_sort`
- `src/utils.h` – Header file for useful utilities
- `src/utils.c` – Source code for useful utilities
- `src/sort-eval.c` – Evaluation program for all sorting implementations
- `scripts` – Scripts for the build system and generating datasets

2. Design Description: Sorting Algorithm

You are responsible for implementing the following four sorting algorithms:

- `void insertion_sort ( int arr[], size_t size );`
- `void selection_sort ( int arr[], size_t size );`
- `void merge_sort ( int arr[], size_t size );`
- `void quick_sort ( int arr[], size_t size );`

Each function takes as input an integer array `arr` with length `size` and conducts an in-place sort. Notice that all of these sorting algorithm implementations have the same *interface*. This means that any of these implementations can be "dropped in" to replace any other implementation as needed!

In general, sorting implementations can be conducted out-of-place or in-place. Typically, out-of-place implementations are more intuitive and easier to understand but also result in additional space com-

plexity. The in-place implementations are typically slightly more complex but have far more efficient space complexity. In this assignment, we require that each sorting algorithm be implemented *in-place*. Sorting algorithm implementations are used widely and frequently, and space complexity makes a significant impact on the efficiency of larger software programs. **Note that because the merge sort algorithm is not easily implemented in-place, you are permitted to write an out-of-place implementation of merge sort followed by an element-by-element array copy back into the original array**.

Previous assignments explored the tradeoffs between iterative and recursive solutions for the same problem. **In this assignment, the merge sort and quick sort implementations are required to be implemented using recursion**. Think critically about what the base cases must be and what the recursive cases must be in order to avoid infinite recursion and to ensure correct functionality. We encourage you to enumerate the base and recursive cases and draw out your thoughts visually on a piece of paper before you begin writing any code!

Because the interface has separated interface from implementation, many implementation decisions are left for you to decide. For example, one critical design implementation for quick sort involves choosing a pivot. We do not require any particular pivot selection mechanism. Please choose your own pivot and justify your decision in the report. Students can also consider using a hybrid variant of merge sort and/or quick sort as discussed in lecture to further improve the performance of these algorithms. Students are not allowed to use pointer arithmetic and should instead only use array indexing.

Sorting algorithm implementations often require a swap operation, and we have provided a `swap()` function for you in `src/utils.c`. It is also very convenient in this assignment to print out the contents of an array. We have provided the `print_array()` utility function for you as well. These two utility functions have the following function signatures:

```
void print_array ( int arr[], size_t size );
void swap        ( int arr[], size_t i, size_t j );
```

Here is an example program that demonstrates how these utility functions can be used:

```
#include "utils.h"
#include <stdio.h>

int main( void )
{
  size_t size   = 4;                // Initialize size
  int    arr[4] = {10, 11, 12, 13}; // Initialize the array

  print_array( arr, size );         // Output: "10, 11, 12, 13"
  swap( arr, 1, 3 );                // Call swap on arr[1] and arr[3]
  print_array( arr, size );         // Output: "10, 13, 12, 11"

  return 0;
}
```

Write your implementation of insertion sort inside `src/insertion-sort.c`, selection sort inside of `src/selection-sort.c`, merge sort inside of `src/merge-sort.c`, and quick sort inside of `src/quick-sort.c`. Please make sure to comment your code. Your goal should be to comment well enough that if you

see this code again in one year, you will be able to quickly read through the comments and understand your own implementation. Others should also be able to read your comments and understand your approach.

Note that you report should discuss insertion and selection sort as your "first implementation" and merge and quick sort as your "second implementation."

3. Testing Strategy

As in the previous programming assignments, you will develop an effective testing strategy and write tests systematically so that you can give a compelling argument for the robustness of your code. We have provided basic tests for each of the sorting algorithm functions. You will need to add directed tests and random tests to provide enough evidence that your code works as intended.

The basic tests provided to you take into consideration the fact that the sorting algorithm implementations are done in-place. Setting up a test therefore requires an unsorted array (to be passed to your sorting function) in addition to the golden model sorted array which will be compared against. Please follow this approach for all of your testing.

Design your directed tests to stress different cases that you as a programmer suspect may be challenging for your functions to handle. For example, what happens if you call sort on a zero-sized array? Are there any special cases where a design decision (e.g., choice of pivot) can break the functionality of your sort? Convince yourself that your sorting algorith implementations are robust by carefully developing a testing strategy.

Random testing will be useful in this programming assignment to stress test your sorting algorithm implementations with large amounts of data. Ensure that your random tests are repeatable by calling the `srand` function once at the top of your test case with a constant seed (e.g., `srand(0)`). You may use the `qsort()` C standard library function *for random testing only*. This function has the following function signature:

```
void qsort( void* base, size_t num, size_t size,
            int (*compare)( const void*,const void* ) );
```

The required `compare()` function has been provided in `src/utils.h`. Here is an example program that uses the standard library `qsort` to sort an array of four integers:

```
#include "utils.h"
#include <stdio.h>

int main( void )
{
  size_t size   = 4;                        // Initialize size
  int    arr[4] = {14, 13, 12, 11};         // Initialize array contents

  qsort( arr, size, sizeof(int), compare ); // Call qsort
  print_array( arr, size );                 // Output: "11, 12, 13, 14"

  return 0;
}
```

4. Evaluation

Once you have verified the functionality of the sorting algorithm implementations, you can evaluate their performance across a range of input datasets. We provide you a performance analysis harness for this purpose. You can build and run these evaluation programs like this:

```
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa3-algo/build-eval
% cd pa3-algo/build-eval
% ../configure --enable-eval
% make eval
```

We are working in a separate `build-eval` build directory, and we are using the `--enable-eval` command line option to the `configure` script. This tells the build system to create optimized executables without any extra debugging information.

The evaluation program runs each of your functions with input datasets of size 10,000 in different patterns: random, reversed, almost sorted, and few unique. There are also two random datasets of size 50,000 and 100,000 that will be useful in evaluating for large N.

We have provided summary tables as part of the evaluation program that summarize the raw data for each sorting algorithm implementation with each input dataset, as well as visual summary tables that use small 'o' symbols to represent relative time. For example, an entry with ten 'o' symbols is running ten times longer than an entry with a single 'o' symbol. The visual summary tables correspond directly to the raw data, but are much easier to understand and will be useful in your report.

This harness will enable you to compare the performance across all sorting algorithm implementations and see evidence for the differences in their time complexities. Which algorithm do you expect to perform best for small dataset sizes? For large dataset sizes? For an almost sorted dataset? For a reverse sorted dataset? How do best case, average case, and worst case compare? Use the evaluation results to compare your sorting algorithms in your report.

The evaluation programs also ensure that your implementations are producing the correct results, however, you should not use the evaluation programs for testing. If your implementations fail during the evaluation, then your testing strategy is insufficient. You must add more unit tests to effectively test your program before returning to performance evaluation.

Acknowledgments

This programming assignment was created by Christopher Batten and Christopher Torng as part of the ECE 2400 Computer Systems Programming course at Cornell University.