1. Introduction

The third programming assignment is designed to give you experience working across four important sorting algorithms in computer systems programming: selection sort, merge sort, quick sort, and radix 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 (except radix sort) 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: selection sort, merge sort, quick sort, and radix 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 CMake/CTest 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 four page design report that describes each design as well as your implementation, qualitatively discusses trade-offs in each design, 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 in this PA. Specific requirements for this milestone are described later in this handout. 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
% cmake ..
% 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:

- `CMakeLists.txt` - CMake configuration script to generate Makefile
- `src/selection-sort.h` - Header file for `selection_sort`
- `src/selection-sort.c` - Source code for `selection_sort`
- `src/selection-sort-main.c` - Ad-hoc test program for `selection_sort`
- `src/merge-sort.h` - Header file for `merge_sort`
- `src/merge-sort.c` - Source code for `merge_sort`
- `src/merge-sort-main.c` - Ad-hoc test program for `merge_sort`
- `src/quick-sort.h` - Header file for `quick_sort`
- `src/quick-sort.c` - Source code for `quick_sort`
- `src/quick-sort-main.c` - Ad-hoc test program for `quick_sort`
- `src/radix-sort.h` - Header file for `radix_sort`
- `src/radix-sort.c` - Source code for `radix_sort`
- `src/radix-sort-main.c` - Ad-hoc test program for `radix_sort`
- `tests/selection-sort-basic-tests.c` - Basic test cases for `selection_sort`
- `tests/selection-sort-directed-tests.c` - Directed test cases for `selection_sort`
- `tests/selection-sort-random-tests.c` - Random test cases for `selection_sort`
- `tests/merge-sort-basic-tests.c` - Basic test cases for `merge_sort`
- `tests/merge-sort-directed-tests.c` - Directed test cases for `merge_sort`
- `tests/merge-sort-random-tests.c` - Random test cases for `merge_sort`
- `tests/quick-sort-basic-tests.c` - Basic test cases for `quick_sort`
- `tests/quick-sort-directed-tests.c` - Directed test cases for `quick_sort`
- `tests/quick-sort-random-tests.c` - Random test cases for `quick_sort`
- `tests/radix-sort-basic-tests.c` - Basic test cases for `radix_sort`
- `tests/radix-sort-directed-tests.c` - Directed test cases for `radix_sort`
- `tests/radix-sort-random-tests.c` - Random test cases for `radix_sort`
- `src/sort-eval.c` - Evaluation program for all implementations
- `src/sort.dat` - Input dataset for the evaluation
- `src/utils.h` - Header file for useful utilities
- `src/utils.c` - Source code for useful utilities
- `scripts` - Scripts for the build system and generating datasets
2. Implementation Specifications

You are responsible for implementing the following four sorting algorithms:

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

Each function takes as input an integer array \( arr \) with length \( size \) and sorts numbers in the array in an ascending order. 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 this PA, integers in \( arr \) have a limited range between 0 and 99999 (inclusive). Students do not need to test and evaluate their implementations with input values out of this range. Students are strongly encouraged to use array indexing instead of pointer arithmetic.

Write your implementation of selection sort inside of \texttt{src/selection-sort.c}, merge sort inside of \texttt{src/merge-sort.c}, quick sort inside of \texttt{src/quick-sort.c}, and radix sort inside of \texttt{src/radix-sort.c}.

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

\[
\begin{align*}
\text{void print_array ( int arr[], size_t size );} \\
\text{void swap ( int arr[], size_t i, size_t j );}
\end{align*}
\]

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

```c
#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;
}
```

Students are required to complete at least one implementation of each sorting algorithm. For each algorithm, we describe a basic implementation, and we also sketch one or more advanced implementations. Most students are encouraged to start by implementing the basic implementation before potentially exploring the advanced implementations. More advanced students may wish to start by exploring the more advanced implementations. If a student explores multiple implementations of the same sorting algorithm, they should be sure to evaluate all of these implementations and record the corresponding results to enable a deeper discussion in the evaluation section of the report. Students should discuss these implementations they explored even if they only submit one. Students
can receive full credit for coding functionality with either basic or advanced implementations as long
as these implementations pass all of the student and staff tests.

2.1. Selection Sort

For the basic implementation, students can implement an “out-of-place” selection sort that moves
elements from the given array into a temporary array and then copies this temporary array back into
the given array. A more advanced implementation of selection sort can be done “in-place”, meaning
that it does not require a temporary array. Typically, out-of-place implementations are more intuitive
and easier to understand than their in-place counterparts, but at the cost of increased space usage.

2.2. Merge Sort

For the basic implementation, students can implement an out-of-place merge sort using recursion.
Think critically about the base and recursive cases to ensure correct functionality and avoid infinite
recursion. We encourage students to enumerate the base and recursive cases and draw out your
thoughts visually on a piece of paper before you begin writing any code! A more advanced “hybrid”
implementation would start with the recursive merge sort algorithm, but then switch to a different
sorting algorithm (e.g., selection sort) when the size of the partition is relatively small. Students will
need to quantitatively experiment to determine when it makes sense to switch sorting algorithms.
In-place versions of merge sort are possible but can be particularly complex to implement.

2.3. Quick Sort

For the basic implementation, students will implement an in-place quick sort using recursion. Sim-
ilar to merge sort, students should carefully consider the base and recursive cases. For the basic
implementation, students might want to simply use the first element in a partition as the pivot. A
more advanced implementation might choose the pivot more carefully so as to improve the typi-
cal case execution time. As with merge sort, a more advanced “hybrid” implementation would start
with the recursive quick sort algorithm but then switch to a different sorting algorithm (e.g., selection
sort) when the size of partition is relatively small. Students will need to quantitatively experiment to
determine when it makes sense to switch sorting algorithms.

2.4. Radix Sort

For the basic implementation, students will implement a pseudo-radix sort algorithm. Assume the
numbers to be sorted are represented in base $R$ ($R$ is also called the “radix”). Assume the maximum
number of digits in the representation is $K$. For example, if the radix is 10 and all numbers are in the
range between 0 and 99999, $R$ is 10 and $K$ is 5. Here is the pseudo-code for the pseudo-radix sort
algorithm.

1. make $R$ bins
2. for each number in arr:
3. get the $(K-1)$-th digit of the number
4. put the number into a corresponding bin based on the $(K-1)$-th digit
5. for each bin:
6. use a regular comparative sort algorithm to sort the numbers in a bin
7. concatenate all bins together in an ascending order

Assume $R$ is 10 and $K$ is 5. Then for the number 12345 the $(K-1)$-th digit is 1, and for the number 345
the $(K-1)$-th digit is 0 (i.e., 345 is really 00345). We provide a helper function called get_digit_at
in src/radix_sort.c to get the i-th digit of a given number represented in a given radix. Read our comments in the code for how to use the function. For the basic implementation, we recommend starting with a radix of 10.

A “real” radix sort implementation, will use a non-comparative sort for each digit and will also sort each digit incrementally (either starting from the most or least significant digit). Students might want to consider a radix other than 10. Students will need to consult the reference material posted on Piazza to learn more about real radix sort implementations.

2.5. Qualitative Comparisons

Before testing and evaluating your implementations, you should think qualitatively about your implementations. In your report, you should include an explicit subsection at the end of Section 2 to discuss this qualitative comparison. For each implementation, what is its time and space complexity in terms of input size $N$? For the radix sort, how do $R$ and/or $K$ impact the time and space complexity? If you experimented with more than one implementation for a given algorithm, what are the trade-offs between these implementations? Why do you think your implementation would be better than others? When should one sorting algorithm be used instead of the others?

3. Testing Strategy

You are responsible for developing an effective testing strategy to ensure all implementations are correct. Writing tests is one of the most important and challenging aspects of software programming. Software engineers often spend far more time implementing tests than they do implementing the actual program.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation in src/selection-sort-main.c, src/merge-sort-main.c, src/quick-sort-main.c, and src/radix-sort-main.c. Students are strongly encouraged to start compiling and running these ad-hoc test programs directly in the src/ directory without using any build-automation tool (e.g., CMake and Make).

You can build and run the given ad-hoc test program for selection sort like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo/src
% gcc -Wall -o selection-sort-main selection-sort.c selection-sort-main.c utils.c
% ./selection-sort-main
```

The -Wall command line option will ensure that gcc reports all warnings.

3.2. Systematic Unit Testing

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 algorithms. 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 implementation) in addition to a sorted array which will be used as a reference. 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 implementations 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 implementation? Convince yourself that your sorting algorithm 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:

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

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

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

    return 0;
}
```

3.3. Code Coverage

After your implementations pass all unit tests, you can evaluate how effective your test suite is by measuring its code coverage. The code coverage will tell you how much of your source code your test suite executed during your unit testing. The higher the code coverage is, the less likely some bugs have not been detected.

You can run the code coverage like this:

```bash
% cd ${HOME}/ece2400/netid/pa3-algo
% mkdir -p build
% cd build
% cmake ..
% make coverage
```
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
% cmake .. -DCMAKE_BUILD_TYPE=RELEASE
% make eval
```

Notice that you will work in a separate build-eval directory and use -DCMAKE_BUILD_TYPE=RELEASE flag in cmake to create optimized executables without any extra debugging information.

The evaluation program runs each of your implementations 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.

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.

5. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require you to complete an incremental milestone, push your code to GitHub and then use the ece2400-pa-admin script to submit your milestone by the date specified by the instructors. In this PA, you will need to submit your implementations of selection sort and merge sort, your directed tests, and your random tests for these two implementations.

Acknowledgments

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