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 bucket sort. While this programming assignment is more algorithm centric, you will still need to leverage your knowledge of data structures. You will be adding a new sorting function to the vector data structure you developed in the previous programming assignment. 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 be used to sort other types of elements as well. Sorting the elements in a data structure 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 first implement three functions corresponding to three sorting algorithms: selection sort, merge sort, and quick sort. For each sorting algorithm you will need to implement a helper function first, and then you will use this helper function in the implementation of the sorting algorithm. You will then add a new sorting function to the vector data structure from the previous programming assignment. Finally, you will implement bucket sort by using a set of vectors. This final example will illustrate the interplay between algorithms and data structures; an algorithm (bucket sort) will use a data structure (vector) which uses another algorithm (one of the other sorting algorithms) on a simple data structure (array). 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, sorted forward, sorted reverse) on the execution time and space usage of the various algorithms. As in the previous assignments, we will leverage the CMake framework for building programs, the CTest framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your algorithms are functional and verified, you will write a three-page report that provides a detailed deep-dive discussion on one implementation, qualitatively evaluates trade-offs across both implementations, and quantitatively evaluates the performance across both implementations. While the final code and report are all due at the end of the assignment, we also require meeting 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. For this PA, we require you to discuss your bucket sort implementation (including whatever algorithm you choose to use in your vector sort function) for the implementation deep dive in the report.

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
% cd ${HOME}/ece2400/netid/pa3-algo
% tree

Where netid should be replaced with your NetID. You should never for 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 working on your programming assignment.

% cd ${HOME}/ece2400/netid
% git pull
% cd ${HOME}/ece2400/netid/pa3-algo
% tree

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-adhoc.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-adhoc.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-adhoc.c – Ad-hoc test program for quick_sort
- src/vector.h – Header file for vector
- src/vector.c – Source code for vector
- src/vector-adhoc.c – Ad-hoc test program for vector
- src/bucket-sort.h – Header file for bucket_sort
- src/bucket-sort.c – Source code for bucket_sort
- src/bucket-sort-adhoc.c – Ad-hoc test program for bucket_sort
- test/selection-sort-directed-test.c – Directed test cases for selection_sort
- test/selection-sort-random-test.c – Random test cases for selection_sort
- test/merge-sort-directed-test.c – Directed test cases for merge_sort
- test/merge-sort-random-test.c – Random test cases for merge_sort
- test/quick-sort-directed-test.c – Directed test cases for quick_sort
- test/quick-sort-random-test.c – Random test cases for quick_sort
- test/bucket-sort-directed-test.c – Directed test cases for bucket_sort
- test/bucket-sort-random-test.c – Random test cases for bucket_sort
- test/utst.h – Helper macros and functions used for unit testing
- eval/sort.dat – Input dataset for the evaluation
- eval/selection-sort-eval.c – Evaluation program for selection_sort
- eval/merge-sort-eval.c – Evaluation program for merge_sort
2. Implementation Specifications

The high-level goal for this programming assignment is to implement four different sorting algorithms and to also add a new sorting function to the vector data structure you developed in the previous programming assignment. The programming assignment is divided into eight steps. Implement and test each step before moving on to the next step.

- Step 1. Implement and test `find_min` helper function
- Step 2. Implement and test `selection_sort` using `find_min`
- Step 3. Implement and test `merge` helper function
- Step 4. Implement and test `merge_sort` using `merge`
- Step 5. Implement and test `partition` helper function
- Step 6. Implement and test `quick_sort` using `partition`
- Step 7. Implement and test `vector_int_sort` using one of the sort functions
- Step 8. Implement and test `bucket_sort` using `vector_int_sort`

Implementing and testing a helper function before implementing and testing the sorting function is an example of effective procedural programming. Procedural programming involves carefully organizing your program into “procedures” (i.e., functions) to help mitigate design complexity. The helper function helps us focus on a smaller part of the implementation, before working on the complete implementation of the sorting algorithm.

2.1. Step 1. Implement `find_min` Helper Function

The `find_min` helper function has the following interface:

- `size_t find_min( int* a, size_t begin, size_t end );`

This function should find the minimum value in given array from the index `begin` to the index `end-1`. Note that the range is inclusive of `begin` and exclusive of `end`. So if we have an array of eight elements and we want to find the minimum of the entire array we would set `begin` to 0 and `end` to 8. The function should return the index of where the minimum value is stored in the given array. Write your implementation in `src/selection-sort.c`.

2.2. Step 2. Implement `selection_sort` Using `find_min`

The `selection_sort` function has the following interface:

- `void selection_sort( int* a, size_t size );`

This function takes as input an integer array `a` with length `size` and sorts numbers in the array in an ascending order. Your implementation must make use of the `find_min` helper function from the previous step. You can use either an out-of-place or an in-place implementation. Write your implementation in `src/selection-sort.c`.

2.3. Step 3. Implement `merge` Helper Function

The `merge` helper function has the following interface:
• void merge( int* dst, int* src0, size_t begin0, size_t end0,
  int* src1, size_t begin1, size_t end1 )

This function should merge the given src0 and src1 range of values and write the result into
the given dst array. The function can assume that the src0 and src1 range of values are already sorted
and should ensure that the resulting dst array is also sorted. The src ranges to be merged are from
the index begin to the index end-1. Note that the range is inclusive of begin and exclusive of end.
This function can assume that the dst array size is at least equal to the sum of the two src ranges
(i.e., (end0 − begin0) + (end1 − begin1)). Write your implementation in src/merge-sort.c.

2.4. Step 4. Implement merge_sort Using merge

The merge_sort function has the following interface:

• void merge_sort( int* a, size_t size );

This function takes as input an integer array a with length size and sorts numbers in the array
in an ascending order. Your implementation must make use of the merge helper function from
the previous step. You should use an out-of-place implementation. Write your implementation in
src/merge-sort.c.

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 basic implementation
will only use the recursive merge sort algorithm. 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 might want to quantitatively
experiment to determine when it makes sense to switch sorting algorithms. Note that a basic is
perfectly fine and can receive full credit.

2.5. Step 5. Implement partition Helper Function

The partition helper function has the following interface:

• size_t partition( int* a, size_t begin, size_t end )

This function should partition the given array from the index begin to the index end-1. Note that
the range is inclusive of begin and exclusive of end. So if we have an array of eight elements and
we want to partition the entire array we would set begin to 0 and end to 8. The partition should
be based on a pivot which is chosen from the elements in the given range. All elements less than
the pivot should be in the left partition and all elements greater than the pivot should be in the
right partition. The pivot should be moved in between these two partitions, and the index of where
this pivot is located should be returned from the function. You can use either an out-of-place or an
in-place implementation. Write your implementation in src/quick-sort.c.

Think critically about how to choose an effective pivot which will result in a good average case par-
tition where the pivot ends up in the middle of the given range. A basic implementation might want
to simply use the last element in the range as the pivot. A more advanced implementation might
use the median element, an approximately median element, or a pseudo-random (but repeatable)
 element. Students might want to quantitatively experiment to determine an effective pivot. Note
that a basic is perfectly fine and can receive full credit.
2.6. **Step 6. Implement** quick_sort Using partition

The quick_sort function has the following interface:

- void quick_sort( int* a, size_t size );

This function takes as input an integer array a with length size and sorts numbers in the array in an ascending order. Your implementation must make use of the partition helper function from the previous step. You should use an in-place implementation. Write your implementation in src/quick-sort.c.

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. As in 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 the partition is relatively small. Students might want to quantitatively experiment to determine when it makes sense to switch sorting algorithms. Note that a basic is perfectly fine and can receive full credit.

2.7. **Step 7. Implement** vector_int_sort Using One of the Sort Functions

Start by copying your implementation of the vector data structure from the previous programming assignment into src/vector.c. We now want to add a new function to your vector data structure with the following interface:

- void vector_int_sort( vector_int_t* this );

This function should directly call one of your previously developed sorting functions. Do not copy and paste the code! You should #include the appropriate header and simply call one of these three functions:

- void selection_sort( int* a, size_t size );
- void merge_sort( int* a, size_t size );
- void quick_sort( int* a, size_t size );

You will pass in the internal array and size managed by the vector data structure. Which sort function you use is up to you, but choose a sort function that will perform well in the general case.

2.8. **Step 8. Implement** bucket_sort using vector_int_sort

The bucket_sort function has the following interface:

- void bucket_sort( int* a, size_t size );

This function takes as input an integer array a with length size and sorts numbers in the array in an ascending order. The high-level idea for a bucket sort is to divide the input array into K buckets, sort each bucket using a previously developed sort function, and then concatenate the sorted buckets to produce the final fully sorted array. More specifically, your bucket sort should work as follows: (1) scan through the array to find the minimum and maximum values; (2) determine the range for each bucket as ((max - min) / K); (3) construct K vectors; (4) scan through the array again and push back each element into the appropriate vector based on its value; (5) call the vector_int_sort function developed in the previous step; (6) scan through each bucket writing the sorted values into the original array. Write your implementation in src/bucket-sort.c.
Note that \( K \) is a compile time constant. Think critically about a good value for \( K \). Students might want to quantitatively experiment to determine what value of \( K \) performs best.

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-adhoc.c`, `src/merge-sort-adhoc.c`, `src/quick-sort-adhoc.c`, and `src/radix-sort-adhoc.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:

```bash
% cd ${HOME}/ece2400/netid/pa3-algo/src
% gcc -Wall -Wextra -pedantic-errors \
    -o selection-sort-adhoc ece2400-stdlib.c selection-sort-adhoc.c selection-sort.c
% ./selection-sort-adhoc
```

The `-Wall`, `-Wextra`, `-pedantic-errors` flags will ensure that `gcc` reports all warnings.

3.2. Systematic Unit Testing

While ad-hoc test programs help you quickly see results of your implementations, they are often too simple to cover most scenarios. We need a systematic unit testing strategy to hopefully test all possible scenarios efficiently.

In this course, we are using CMake/CTest as a build and test automation tool. For each implementation, we provide a directed test program that should include several test cases to target different categories and a random test program that should test that your implementation works for random inputs. **We only provide a very few directed tests and no random tests. You must add many more directed and random tests to thoroughly test your implementations!**

Note that you should definitely test your helper functions, but these helper functions are *not* meant to be used in any other context besides the corresponding sorting algorithm. So testing the helper function is really a form of **white box** testing. The staff tests will do some “light” testing of your helper functions to ensure they meet the basic specification described in this document, but most of our testing will be for the sort functions.

The tests for the sort functions 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:

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

This interface is a little complicated and different from the interface we are using for the sort functions in this programming assignment. So we have provided a wrapper in ‘ece2400-stdlib.h’ which you can use like this:

```c
#include "ece2400-stdlib.h"
#include <stdio.h>

int main( void )
{
    size_t size = 4; // Initialize size
    int a[4] = { 14, 11, 12, 13 }; // Initialize array contents
    ece2400_sort( a, size ); // Call C standard library qsort
    ece2400_print_array( a, size ); // Output: "11, 12, 13, 14"
    return 0;
}
```

### 3.3. Memory Leaks

Students are also responsible for making sure that their program contains no memory leaks or other issues with dynamic allocation. We have added a new makefile target called memcheck which runs all of the test programs with Valgrind. Valgrind will force the test to fail if it detects any kind of memory leak or other issues with dynamic allocation.

You can check memory leaks and other issues with dynamic memory allocation for all your test programs like this:

```sh
% cd ${HOME}/ece2400/netid/pa3-algo/build
% make memcheck
```

You can just check one test program (e.g. selection-sort-directed-test) like this:

```sh
% cd ${HOME}/ece2400/netid/pa3-algo/build
% make selection-sort-directed-test
% valgrind --trace-children=yes --leak-check=full --error-exitcode=1 --undef-value-errors=no ./selection-sort-directed-test
```

Those are quite a few command line options to Valgrind, so we have created a ‘ece2400-valgrid’ script. This script is just a simple wrapper which calls Valgrind with the right options.

```sh
% cd ${HOME}/ece2400/netid/pa3-algo/build
```
3.4. 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
% rm -rf build-coverage
% mkdir -p build-coverage
% cd build-coverage
% cmake ..
% make check
% make coverage
% elinks coverage-html/index.html
```

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 with a set of evaluation programs for each of the four sorting algorithms as well as for the C standard library’s `qsort` implementation. You can build these evaluation programs like this:

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

Note how we are working in a separate `build-eval` build directory, and that we are using the `-DCMAKE_BUILD_TYPE=eval` command line option to the `cmake` script. This tells the build system to create optimized executable without any extra debugging information. You must do your quantitative evaluation using an eval build. Using a debug build for evaluation produces meaningless results.

To run an evaluation for a specific implementation, you simply specify the input pattern and the size of input array on the command line. For example, the following runs an evaluation with uniform-random input of 100 elements using the selection sort implementation.

```bash
% cd ${HOME}/ece2400/netid/pa3-algo/build-eval
% make selection-sort-eval
% ./selection-sort-eval urandom 100
```

Available input patterns are:

- `urandom` – Input data is uniform-randomly distributed
- `sorted-asc` – Input data is already sorted in ascending order
- `sorted-desc` – Input data is already sorted in descending order
The evaluation programs measure the execution time as well as the memory (heap) usage. This will enable you to compare the performance and memory usage between your sorting implementations. The evaluation programs also verify 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 evaluation.

You should quantitatively evaluate your implementations with input size from 1 to 50000. You do not need to further increase the input size if the average execution time for a trial exceeds 1.5 seconds. So you will likely reach the 1.5 second for much smaller input sizes when evaluating selection sort compared to the other sorting implementations. Record all performance and memory usage data and create three plots. The first plot should have the size of the input on the x-axis and execution time for urandom pattern on the y-axis. Plot a line for each of the five sorting functions (selection-sort, merge-sort, quick-sort, bucket-sort, and qsort). The second plot should have the size of the input on the x-axis and execution time of quick-sort for the three different patterns on the y-axis. Plot a line for each of the three patterns (urandom, sorted-asc, sorted-desc). The third plot should have the size of the input on the x-axis and the peak heap usage on the y-axis. Plot a line for merge-sort and a line for bucket-sort. If you have any other out-of-place implementations, include them in the third plot as well. **Students are required to include these three plots in their report, and to discuss these results in the quantitative evaluation section.** Ensure your plots are easy to read with a legend, reasonable font sizes, and appropriate colors/markers for black and white printing.

### 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 and push your code to GitHub on the date specified by the instructor. In this PA, you will need to submit your implementations of find_min, selection_sort, merge, and merge_sort, along with your tests for these implementations.

### Acknowledgments

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