ECE 2400 Computer Systems Programming, Fall 2021
PA3: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-24-20-38

1. Introduction

The third programming assignment will 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, GitHub Actions for continuous integration testing, and lcov for code coverage analysis.

After your algorithms are functional and tested, you will write a four-page report that includes your complexity analysis and a quantitative evaluation of the performance across all implementations. **You should consult the programming assignment logistics document for more information about the expectations for all programming assignments and how they will be assessed. 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. Requirements specific to this PA for the incremental milestone and the final report are described at the end of this handout.**

This handout assumes that you have read and understand the course tutorials and that you have attended the discussion sections. To get started, log in to an `ecelinux` server, source the setup script, and clone your individual remote 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 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 working on your programming assignment.

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

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/ece2400-stdlib.h` – Header file for course standard library
- `src/ece2400-stdlib.c` – Source code for course standard library
- `src/selection-sort-int.h` – Header file for selection sort
- `src/selection-sort-int.c` – Source code for selection sort
- `src/selection-sort-int-adhoc.c` – Ad-hoc test program for selection sort
- `src/merge-sort-int.h` – Header file for merge sort
- `src/merge-sort-int.c` – Source code for merge sort
- `src/merge-sort-int-adhoc.c` – Ad-hoc test program for merge sort
- `src/quick-sort-int.h` – Header file for quick sort
- `src/quick-sort-int.c` – Source code for quick sort
- `src/quick-sort-int-adhoc.c` – Ad-hoc test program for quick sort
- `src/vector-int.h` – Header file for `vector-int`
- `src/vector-int.c` – Source code for `vector-int`
- `src/vector-int-adhoc.c` – Ad-hoc test program for `vector-int`
- `src/bucket-sort-int.h` – Header file for bucket sort
- `src/bucket-sort-int.c` – Source code for bucket sort
- `src/bucket-sort-int-adhoc.c` – Ad-hoc test program for bucket sort
- `test/selection-sort-int-directed-test.c` – Directed test cases for selection sort
- `test/selection-sort-int-random-test.c` – Random test cases for selection sort
- `test/selection-sort-int-helper-test.c` – Whitebox test cases for selection sort
- `test/merge-sort-int-directed-test.c` – Directed test cases for merge sort
- `test/merge-sort-int-random-test.c` – Random test cases for merge sort
- `test/merge-sort-int-helper-test.c` – Whitebox test cases for merge sort
- `test/quick-sort-int-directed-test.c` – Directed test cases for quick sort
- `test/quick-sort-int-random-test.c` – Random test cases for quick sort
- `test/quick-sort-int-helper-test.c` – Whitebox test cases for quick sort
- `test/vector-int-directed-test.c` – Directed test cases for `vector_int_t`
- `test/vector-int-random-test.c` – Random test cases for `vector_int_t`

- `test/bucket-sort-int-directed-test.c` – Directed test cases for bucket sort
- `test/bucket-sort-int-random-test.c` – Random test cases for bucket sort

- `eval/sort.dat` – Input dataset for the evaluation
- `eval/selection-sort-int-eval.c` – Evaluation program for selection sort
- `eval/merge-sort-int-eval.c` – Evaluation program for merge sort
- `eval/quick-sort-int-eval.c` – Evaluation program for quick sort
- `eval/bucket-sort-int-eval.c` – Evaluation program for bucket sort
- `eval/std-sort-eval.c` – Evaluation program for standard C library sort

The programming assignment is divided into the following steps. Complete 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_int` using `find_min`
- Step 3. Implement and test `merge` helper function
- Step 4. Implement and test `merge_sort_int` using `merge`
- Step 5. Implement and test `partition` helper function
- Step 6. Implement and test `quick_sort_int` using `partition`
- Step 7. Implement and test `vector_int_sort` using one of the sort functions
- Step 8. Implement and test `bucket_sort_int` using `vector_int_sort`

We cannot stress enough how important it is to take an incremental design approach! You really must implement *and test* each helper function before trying to implement the sorting function. This means more than just adhoc testing. You must do thorough directed testing of your helper function (in the `-helper-test.c` test program) *before* implementing the sorting function.

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. 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.

Note that your implementations cannot use anything from the Standard C library except for the `printf` function defined in `stdio.h`, the MIN/MAX macros defined in `limits.h`, the `NULL` macro defined in `stddef.h`, and the `assert` macro defined in `assert.h`. You should not use `malloc` and `free` functions directly, but should instead be using `ece2400_malloc` and `ece2400_free`.

2.1. `find_min` Helper Function

The `find_min` helper function has the following interface:

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

This function should find the minimum value in given array `a` 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. This is a helper function, so you are responsible for determining if and how you want to handle corner cases based on how you plan to use this helper function in `selection_sort_int`. Write your implementation in `src/selection-sort-int.c`.

2.2. `selection_sort_int` Using `find_min`

The `selection_sort_int` function has the following interface:

- `void selection_sort_int( int* a, int 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-int.c`. You can assume that `size` correctly reflects the size of the input array. Your algorithm must work correctly if `size` is zero, which means the input array pointer `a` may be a NULL pointer.

2.3. `merge` Helper Function

The `merge` helper function has the following interface:

```
void merge( int* dst, int* src0, int begin0, int end0,
                      int* src1, int begin1, int 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 should assume that the `dst` array size is at least equal to the sum of the two `src` ranges (i.e., (`end0` − `begin0`) + (`end1` − `begin1`)). This is a helper function, so you are responsible for determining if and how you want to handle corner cases based on how you plan to use this helper function in `merge_sort_int`. Write your implementation in `src/merge-sort-int.c`.

2.4. `merge_sort_int` Using `merge`

The `merge_sort_int` function has the following interface:

- `void merge_sort_int( int* a, int 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-int.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 implementation is perfectly fine and can receive full credit. Your algorithm must work correctly if `size` is zero, which means the input array pointer `a` may be a NULL pointer.

2.5. `partition` Helper Function

The `partition` helper function has the following interface:

- `int partition( int* a, int begin, int 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. This is a helper function, so you are responsible for determining if and how you want to handle corner cases based on how you plan to use this helper function in `quick_sort_int`. Write your implementation in `src/quick-sort-int.c`.

Think critically about how to choose an effective pivot which will result in a good average case partition 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 algorithm is perfectly fine and can receive full credit.

2.6. `quick_sort_int` Using `partition`

The `quick_sort_int` function has the following interface:

- `void quick_sort_int( int* a, int 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-int.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 algorithm is perfectly fine and can receive full credit. Your algorithm must work correctly if `size` is zero, which means the input array pointer `a` may be a NULL pointer.

2.7. `vector_int_sort` Using One of the Sort Functions

You will need to implement a resizable vector with the following interface.

- `void vector_int_construct( vector_int_t* this );`
- `void vector_int_destruct( vector_int_t* this );`
- `void vector_int_push_back( vector_int_t* this, int value );`
- `int vector_int_size( vector_int_t* this );`

- `int vector_int_at( vector_int_t* this, int idx );`
- `int vector_int_contains( vector_int_t* this, int value );`
- `void vector_int_sort( vector_int_t* this );`
- `void vector_int_print( vector_int_t* this );`

Start by copying your implementation of the vector data structure from the previous programming assignment into `src/vector-int.c`. Note that in this programming assignment we only have a single `vector_int_push_back` function. We recommend choosing the faster implementation. So to be explicit, you should choose either v1 or v2 and rename that function to be `vector_int_push_back` and the remove the other implementation.

Notice that we also want to add a new `vector_int_sort` function to your vector data structure. 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( int* a, int size );`
- `void merge_sort_int( int* a, int size );`
- `void quick_sort_int( int* a, int 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 (i.e., for both small and large arrays with many different data values).

2.8. `bucket_sort_int` **using** `vector_int_sort`

The `bucket_sort_int` function has the following interface:

- `void bucket_sort_int( int* a, int 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 N/K buckets (where N is `size` and K is a key parameter), 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) calculate the number of buckets (M) using $M = N/K$; (2) scan through the array to find the minimum and maximum values; (3) determine the range for each bucket as $((\max - \min)/M)$; (4) construct M vectors; (5) scan through the array again and push back each element into the appropriate vector based on its value; (6) call the `vector_int_sort` function developed in the previous step; (7) scan through each bucket writing the sorted values into the original array. Write your bucket sort implementation in `src/bucket-sort-int.c`.

In general, calculating $(\max - \min)$ can lead to integer overflows. For this PA, it is undefined behavior if the maximum difference between two integers in the array cannot be stored in an `int` (i.e., we will not test this behavior). You can simply perform the $(\max - \min)$ calculation and assume it does not overflow. Your algorithm must work correctly if `size` is zero, which means the input array pointer `a` may be a NULL pointer.

Note that we will be using *Batten's bucket sort* which is a variation on the *generic bucket sort* you may find if you do some independent reading. In the generic bucket sort, K is often used to indicate the number of buckets where the number of buckets are assumed to be close to N or even larger than N, such that each bucket only includes a few elements. This is why the generic bucket sort often uses a $O(N^2)$ sorting algorithm to sort each bucket. The exact relationship between N and and the number of buckets is often somewhat vague in the generic bucket sort. In Batten's bucket sort, we define K

to be the expected number of elements in each bucket assuming a uniform random distribution of the input data. K is a compile time constant that is chosen such that there are fewer buckets and more elements per bucket than in the generic bucket sort. You should set $K = 100$ for all of your primary experiments (i.e., there will be on average about 100 elements in each bucket). Students can optionally explore how varying K impacts performance, but this is not required. **Note that since Batten's bucket sort is not quite the same as the generic bucket sort, you cannot simply reuse the complexity analysis you find in your independent reading. You must think more critically about this specific variation of bucket sort.**

2.9. ECE 2400 Malloc and Free

As in the previous programming assignment, instead of using the `malloc` function directly for dynamic memory allocation in the vector data structure, we provide you a pair of wrapper functions called `ece2400_malloc` and `ece2400_free`. You should also use these functions if need to allocate a temporary array for an out-of-place sorting algorithm. These functions are declared inside `src/ece2400-stdlib.h`. These two functions internally call `malloc` and `free`, but they also keep track of how much heap memory your program has allocated so far.

- `void* ece2400_malloc( size_t mem_size );`

  Dynamically allocate a memory space of size `mem_size` on the heap. The function returns a pointer to the newly allocated space. If the allocation fails, a `NULL` is returned. Note that just like `malloc`, this function has a parameter of type `size_t`. Because we use the `-Wconversion` flag to tell the compiler to warn of us of any potentially unsafe implicit type conversions, this means we need to explicitly cast any variables of time `int` to `size_t` when calling this function. See an example below.

- `void ece2400_free( void* ptr );`

  Deallocate the memory space pointed by `ptr` in the heap. If `ptr` is `NULL`, no action occurs. Note that this function must be used in pair with `ece2400_malloc`, i.e., `ptr` must be a pointer returned by `ece2400_malloc`. Using this function on a pointer returned by normal `malloc` is undefined and may result in a segmentation fault

For reference, here is a simple function that allocates an array of N integers on the heap.

```
int main( void )
{
  int  N    = 32;
  int* data = ece2400_malloc( (size_t) N * sizeof(int) );

  // ... do something with data ...

  ece2400_free( data );
  return 0;
}
```

Notice the need to use an explicitly cast the variable N to `size_t`. Technically this means if N is negative you will allocate a huge amount of memory on the heap, so you should ensure that N is not negative.

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.

Note that while there are limitations on what you can use from the Standard C library in your *implementations* there are no limitations on what you can use from the Standard C library in your *testing strategy*. You should feel free to use the Standard C library in your golden reference models and/or for random testing.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation in `src/selection-sort-int-adhoc.c`, `src/merge-sort-int-adhoc.c`, `src/quick-sort-int-adhoc.c`, and `src/bucket-sort-int-adhoc.c`. The ad-hoc tests for selection, merge, and quick sorts only test the helper functions for these algorithms. Students can use these ad-hoc test programs by compiling and running them directly in the `src/` directory without using any build-automation tool (e.g., CMake and Make). However, we encourage you to transition to the automated test suite as early as possible in your development process.

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

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

The `-Wall` flag will ensure that `gcc` reports most 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 not test your helper functions. All of our testing will be *black-box testing* of the sort functions.

We cannot stress enough how important it is to take an incremental design approach! You really must implement *and test* each helper function before trying to implement the sorting function. This means more than just adhoc testing. You must do thorough directed testing of your helper function (in the `*-helper-test.c` test program) *before* implementing the sorting function.

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 C standard library sorting 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* ) );
```

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:

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

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

  ece2400_sort( a, size );          // Call C standard library sort
  ece2400_print_array( a, size );    // Output: "11, 12, 13, 14"

  return 0;
}
```

Also make sure to copy over your tests in `test/vector-int-directed-test.c` and `test/vector-int-random-test.c`, and to modify them to only test `vector_int_push_back` (and not `vector_int_push_back_v1` and `vector_int_push_back_v2`).

As in the previous programming assignment, we provide you a testing framework you should use for your directed and random testing. See the provided test programs in the `test` subdirectory for how to use this framework. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following macros you should use to check the correctness of your implementations:

- `ECE2400_CHECK_FAIL()` – check program does not reach this point
- `ECE2400_CHECK_TRUE( expr_ )` – check `expr_` is always true
- `ECE2400_CHECK_FALSE( expr_ )` – check `expr_` is always false
- `ECE2400_CHECK_INT_EQ( expr0_, expr1_ )` – check `expr0_` equals `expr1_`

Before running the tests you need to create a separate `build` directory and use `cmake` to create the `Makefile` like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo
% mkdir -p build
% cd build
% cmake ..
```

Now you can build and run all unit tests for all implementations like this:

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

If you are failing a test program, then you can "zoom in" and run all of the unit tests for a single test program (e.g., directed tests for `list`) like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo/build
% make selection-sort-int-directed-test
% ./selection-sort-int-directed-test
```

You can then "zoom in" to a specific test case by passing in the index of that test case like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo/build
% make selection-sort-int-directed-test
% ./selection-sort-int-directed-test 1
% ./selection-sort-int-directed-test 2
```

3.3. Test-Case Crowd Sourcing

While a comprehensive test suite provides strong evidence that your implementation has the correct functionality, it is particularly challenging to write high-quality test cases for all of your implementations. **Students can use test-case crowd-sourcing after the milestone to reduce the workload of constructing a comprehensive test suite.** Test-case crowd-sourcing will use a Canvas discussion page; students cannot see any of the currently posted test cases until they post one of their own. Focus on uploading one or two very strong directed or random test cases. Do not upload more than two test cases. Avoid uploading simple directed test cases since students will have already developed such test cases for the milestone. Posting the basic test case provided by the course instructors, posting an obviously too simple test case, and/or posting something which is obviously meant to "game" the system is not allowed. Let's all work together to crowd-source a great test suite that every student can take advantage of!

You can use test cases posted in the Canvas discussion page in your test programs as long as you acknowledge the author, so be sure to include the comment in your source code which describes the test case and includes the author's name. You will need to renumber the test cases and call them correctly from `main()`. **Make sure you understand the test case and that you feel it is testing correct behavior before including it in your test suite!**

3.4. Memory Leaks

Students are also responsible for making sure that their program contains no memory leaks or other issues with dynamic allocation. We have included a make 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:

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

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

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

```
% cd ${HOME}/ece2400/netid/pa3-algo/build
% make selection-sort-int-directed-test
% ece2400-valgrind ./selection-sort-int-directed-test
```

3.5. 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:

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

Note that these code coverage results will reflect *all* prior runs of the test and evaluation programs in the build directory. That is why in the above example, we do a fresh build in a separate `build-coverage` build directory.

If you want to drill down and explore the coverage of each line in a program you use use the elinks web browser like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo/build-coverage
% elinks coverage-html/index.html
```

Code coverage is just one more piece of evidence you can use to make a compelling case for the correct functionality of your implementations. It is not required that students achieve 100% code coverage. It is far more important that students simply use code coverage as a way to guide their test-driven design than to overly focus on the specific code coverage number.

4. Evaluation

Once you have tested 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 sorting function. You should not need to modify the aevaluation programs. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following functions that are used in the evaluation programs to measure the execution time and heap space usage.

- `ece2400_timer_reset()` – reset global timer
- `ece2400_timer_get_elapsed()` – return elapsed time in seconds since last reset
- `ece2400_mem_reset()` – reset global memory usage counter
- `ece2400_mem_get_aux_usage()` – return max heap space allocated in bytes since last reset

You can build these evaluation programs like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo
% mkdir -p build-eval
% cd 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.

```
% cd ${HOME}/ece2400/netid/pa3-algo/build-eval
% make selection-sort-int-eval
% ./selection-sort-int-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 auxiliary heap space usage. This will enable you to compare the performance and space 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. You will likely reach the 1.5 second for much smaller input sizes when evaluating sorting algorithms with time complexity of $O(N^2)$. Record all of this performance data.

5. Milestone and Report

This section includes critical information about the incremental milestone, final code submission, and the final report specific to this PA. **The programming assignment logistics document provides general details about the requirements for the milestone and final submission.** You must actually read the document to ensure you know how we will access your milestone and final submission.

5.1. 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 by the date specified by the instructor. In this PA, to meet the incremental milestone, you are expected to:

- Complete the implementation of `find_min`
- Complete the implementation of `selection_sort_int`
- Complete the implementation of `merge`
- Complete the implementation of `merge_sort_int`
- Pass all given directed tests for these implementations
- Test these implementations by adding your own helper, directed, and random tests

Here is how we will be testing your milestone:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa3-algo
% mkdir -p build
% cd build
% cmake ..
% make check-milestone
```

5.2. Final Code Submission

Your code quality score will be based on the way you format the text in your source files, proper use of comments, deletion of instructor comments, and uploading the correct files to GitHub (only source files should be uploaded, no generated build files). To assist you in formatting your code correctly, we have created a make target that will autoformat the code for you. You can use it like this:

```
% cd ${HOME}/ece2400/netid/pa3-algo
% mkdir -p build
% cd build
% cmake ..
% make autoformat
% git diff
# ... check all changes ...
% git commit -a -m "autoformat"
```

Note that the `autoformat` target will only work if you have already committed all of your work. This way you can easily use `git diff` to view the changes made by the autoformatting and commit those changes when you are happy with them. Since we provide students an automated way to format their code correctly, students have no excuse for not following the course coding conventions!

Note that students must remove unnecessary comments that are provided by instructors in the code distributed to students. Students must not commit executable binaries or any other unnecessary files. The `autoformat` target will not take care of these issues for you.

To submit your code you simply upload it to GitHub. Your code will be assessed both in terms of functionality and code quality. Your functionality score will be determined by running your code against a series of tests developed by the instructors to test its correctness. Here is how we will be testing your final code submission:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa3-algo
% mkdir -p build
% cd build
% cmake ..
% make check
% make memcheck
% make eval
# ... run the eval programs ...
```

5.3. Final Report

The final report must be uploaded to Canvas. The date you upload your report will indicate how many slip days you are using for the assignment. For this PA, we require you to include four sections: introduction, complexity analysis, quantitative evaluation, and conclusion.

The complexity analysis section of your report must include a table that summarizes the execution time, time complexity (in big-O notation), and space complexity (in big-O notation) of several functions (see Table 1 for a template). For the execution time, you should discuss (at a high level) your chosen units (e.g., a critical operator, critical loop iteration). $T_K(N)$ for `bucket_sort_int` must be a function of both N *and* K. For space complexity analysis, you need to analyze the *auxiliary* heap space usage of *just* that function (i.e., do not include the heap memory usage before calling the function). The input parameter is N where N is the number of elements stored in the input array. This means your time and space complexity analysis should capture the trend as we call the function on larger and larger input arrays. Worst case complexity analysis should consider worst case values stored within the input array. Average complexity analysis should consider an input array of size N whose values follow a uniform random distribution. Note that you don't need to explicitly discuss all six steps of complexity analysis and we are not looking for a rigorously formal proof, but you do need to be clear about the assumptions you made during analysis and provide some kind of compelling high-level argument. Average case analysis for `quick_sort_int` is particularly challenging, so simply noting that you are using the result from lecture is fine.

The quantitative evaluation section of your report must include three plots of execution time and auxiliary heap usage which you then discuss in the quantitative evaluation section of your report. You should create the plots using the data recorded from your quantitative evaluation. The first plot should have the size of the input on the x-axis and execution time for `urandom` pattern on

Algorithm	Analysis	Execution Time $T_K(N)$	Time Complexity big-O	Measured Execution Time (s)	Aux Heap Space Complexity big-O	Measured Aux Heap Space Usage (B)
`selection`	worst					
`merge`	worst					
`quick`	average					
`bucket`	average					

(Not all fields required; see description!)

Table 1: Template for Complexity Analysis and Measured Equations Table

the y-axis. Plot a line for each of the five sorting functions (`selection_sort_int`, `merge_sort_int`, `quick_sort_int`, `bucket_sort_int`, and `stdsort`). The second plot should have the size of the input on the x-axis and execution time of `quick_sort_int` 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 auxiliary heap space usage for `urandom` on the y-axis. Plot a line for each of the four sorting functions `selection_sort_int`, `merge_sort_int`, `quick_sort_int`, and `bucket_sort_int`. Ensure your plots are easy to read with a legend, reasonable font sizes, and appropriate colors/markers for black-and-white printing. The quantitative evaluation section of your report must describe how you collected this data and what conclusions can be drawn from this data.

The quantitative evaluation section of your report must also include a table reporting a best-fit polynomial equation as a function of N for each data series determined using a tool of your choice (see Table 1 for a template). The equation should be in units of seconds for execution time and in units of bytes for auxiliary heap space usage. You should use your complexity analysis to choose the most appropriate degree when doing your polynomial regression. If your complexity analysis suggests the measured data should be $O(1)$ then you should use a 0th degree polynomial fit (i.e., just take the average). If complexity analysis suggests the measured data should be $O(N)$ then you should use a 1st degree polynomial fit (i.e., just use linear regression). If complexity analysis suggests the measured data should be $O(N^2)$ then you should use a 2nd degree polynomial fit, and so on. You should also verify either qualitatively or quantitatively that this produces a good fit. **Note that if your complexity analysis suggests the measured data should be $O(N \cdot \log(N))$ then performing a regression is optional. Also note that performing a regression when measured space usage is "particularly complicated" is optional (it will be obvious what this means after you plot your data!). The quantitative evaluation section of your report must discuss the connection between your theoretical complexity analysis and your experimental data as captured by these best-fit polynomial equations.**

Acknowledgments

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