COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 1

Dr. Joe Anderson

Due: 5 September

1 Objectives

In this lab you will focus on the following objectives:

1. Review dynamic memory and array usage in `c++`
2. Explore empirical tests for program efficiency
3. Compare theoretical algorithm analysis with practical implementations of recursive and iterative sorting algorithms

2 Tasks

1. Put your code in a folder called "Lab-1". This folder will be zipped and turned in at the end.
2. Implement Bubble Sort, Insertion Sort, and Selection Sort as functions which take an array pointer and length, then sorts the array in-place. The one-line descriptions of each sorting routine are below, but you may refer to other sources for details (try not to!).
   (a) Bubble Sort: swap adjacent out-of-order elements until no swaps need to be made
   (b) Insertion Sort: take the "next" element and "insert" it into the proper place in the current prefix by swapping left with larger elements
   (c) Selection Sort: find the smallest element of the current suffix and move it to the end of the current prefix.
3. Test your sorting algorithms on different sized arrays. Use three copies of each test array, passing one to each different sort routine, since the algorithms do the swapping in-place.
   (a) Use some small (≈ 100 elements) and some large ($\approx 1,000,000$ elements) arrays, and various sizes in between
   (b) Use some arrays that are already sorted
   (c) Use some arrays that are sorted backwards
   (d) Use some arrays that contain many duplicate elements
   (e) Use some arrays that are randomly generated
4. Use a function called `isSorted` to validate that a given integer array is in sorted order to verify the correctness of your code.
5. Include the following in the output:

(a) The number of swaps that are made during the sorting process

(b) The time it takes for the sorting to happen using the standard library utilities (requires compiler argument `-std=c++11`). One example of how to do this:

```
#include<chrono>

...

// The "auto" type determines the correct type at compile-time
auto start = std::chrono::system_clock::now();

myFunction(); // replace with your sorting alorithm

auto end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end-start;
std::time_t end_time = std::chrono::system_clock::to_time_t(end);
std::cout << "finished at " << std::ctime(&end_time)
          << "elapsed time: " << elapsed_seconds.count() << "s\n";
```

6. Include a `Makefile` to build your code.

7. Include a `README` file to document your code, any interesting design choices you made, and answer the following questions:

   (a) What is the theoretical time complexity of your sorting algorithms (best and worst case), in terms of the array size?

   (b) How does the absolute timing scale with the number of elements in the array? The size of the elements? Can you use the data collected to rectify this with the theoretical time complexity?

   (c) Aggregate your data into a graph of the complexity for the various array sizes, for example with a spreadsheet program like LibreOffice Calc or Microsoft Word.

   (d) How do the algorithms perform in different cases? What is the best and worst case, according to your own test results?

   (e) How could the code be improved in terms of usability, efficiency, and robustness?

3 Submission

All submitted labs must compile with `g++` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.