COSC 220 - Computer Science II
Lab 6

Dr. Joe Anderson

Due: 15 October 2019

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-6". 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 pseudocode for each algorithm is below:

```
function BubbleSort(A, length)
    swapped = true
    while swapped do
        // if the array is sorted, will remain false
        swapped = false
        for i = 0 to length − 1 (inclusive) do
            if A[i] > A[i + 1] then
                // Found an inversion. Fix and remember
                Swap(A[i], A[i + 1])
                swapped = true
            end if
        end for
    end while
end function
```

Selection sort works as follows:

```
function SelectionSort(A, length)
    for i = 0 to length − 2 (inclusive) do
        // Find the min among A[i, . . . , length]
        min = i // tracks the min index
        for j = i + 1 to length − 1 (inclusive) do
            if A[j] < A[min] then
```

```
                min = j
            end if
        end for
        // Move the min to spot A[i]
        Swap(A[i], A[min])
    end for
end function
```

Insertion sort works as follows:

```
function InsertionSort(A,length)
    // Loop invariant: A[0,...,i-1] is sorted
    for i = 1 to length - 1 (inclusive) do
        j = i
        // Insert A[i] in the correct location among A[0,...,i]
        while j > 0 and A[j] < A[j-1] do
            Swap(A[j], A[j-1])
            j = j - 1
        end while
    end for
end function
```

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. Use some small (≈ 100 elements) and some large ($\approx 1,000,000$ elements) arrays, and various sizes in between.
   (a) Use some arrays that are sorted in ascending order
   (b) Use some arrays that are sorted backwards
   (c) Use some arrays that are randomly generated
      i. Write a function to dynamically allocate an array of a specified length that also assigns each element to a random integer
      ii. To use the native random number libraries, you can use `#include<stdio.h>` and also `#include<time.h>`
      iii. To seed the random number generator to use new values every time, add the instruction `srand(time(NULL))` to your `main` function. Otherwise, it may be the same every time you run your program.
      iv. To generate a random number between 1 and n (inclusively), use `rand() % n + 1`
         A. `rand()` generates a random number from 0 to the largest possible integer. Using the `%` operator reduces that value between 0 and $n - 1$.

4. Write 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. You can use a global counter, or an extra parameter to your sort algorithms.
   (b) The absolute 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.txt` 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? For example: if you double the size of an array, does Selection Sort take four times longer?
   (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?

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.