COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 2

Dr. Joe Anderson

Due: 11 September 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-2". This folder will be zipped and turned in at the end.
2. You should use the testing/benchmarking framework from the previous lab.
3. You may find it clean/helpful to implement a function `swap(int&,int&)` that exchanges the values of two integers.
4. Implement the Quicksort algorithm to sort an array of integers.
   (a) The prototype of the function should look like `void quicksort(int* arr,int start,int end)`.
   (b) Use a subroutine: `int partition(int* arr, int start, int end)` to choose a pivot and perform the work of moving the pivot to the correct location of the array, and returning the *index* of where it ends up in the array. **Make sure this subroutine only needs $\Theta(n)$ time!**

Algorithm 1 Partition($A[m, \ldots, n]$)

```
$pivot := A[n]$ // value of the pivot
$loc := m - 1$ // tracks current position the pivot needs to be
for $i = m$ to $n - 1$ do
    if $A[i] \leq pivot$ then
        // we know $A[i]$ needs to be on the left of the pivot
        $loc = loc + 1$
        swap($A[loc]$, $A[i]$)
    end if
end for
swap($A[loc + 1]$, $A[n]$) // put the pivot in the correct location
```

   (c) Your sort should follow the basic scheme:
      i. If `start` >= `end`, do nothing.
      ii. Otherwise, call `partition` to get the location of a pivot point
      iii. Call `quicksort(arr, start, pivot)`
      iv. Call `quicksort(arr, pivot+1, end)`

5. Implement the Mergesort algorithm to sort an array of integers.
   (a) Record the number of *comparisons* that are made during the sorting process
   (b) Record the absolute time it takes for the sorting to happen using the standard library utilities, as in Lab 1.
   (c) Use a subroutine to complete the merge operation of for integer arrays of length n and m respectively, which takes $O(n + m)$ time.
6. Test your sorting algorithms on different sized arrays.
   (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
7. Use a function to validate that a given integer array is in sorted order to verify the correctness of your code.
8. Include a `Makefile` to build your code.
9. 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) How do the algorithms perform in different cases? What is the best and worst case, according to your own test results?
   (d) Do your observations confirm the difference in the best and worst case for Quicksort? How does Mergesort handle these cases?
   (e) How do this algorithms compare to your implementation of Bubble sort from Lab 1?
   (f) 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.