COMP 2611 – Data Structures

Group Assignment on Sorting Algorithms

Date Due: *Monday November 30, 2020 @ 11.55 pm*

Overview

This assignment requires you to time and analyse several sorting algorithms. You must also do some experiments with some of the algorithms to answer a few questions. You are given the code for all except one of the algorithms. In order to time the algorithms, data sets of increasing sizes must be randomly generated and sorted. You must observe the effect of the increasing size of the data set on performance of the sort algorithms and present your observations in a document using appropriate tools such as charts and tables.

Instructions for Groups

You must work on groups of 3 for this assignment. You are free to choose your own groups. The mark obtained by the group will be awarded to each member of the group. If you are unable to join a group by November 22, 2020, please contact the lecturer of the course.

Code Given

The file, `GroupAssignment.zip`, contains all the files needed for this assignment. The project file is `GroupAssignment.dev`. Unzip the files in a folder and open `GroupAssignment.dev`. Code to test the sort algorithms should be written in the *main* function in *Sorting.cpp*. You may need to modify the algorithms in *SortAlgorithms.cpp* depending on the experiments you are conducting.

Sort Algorithms to be Analyzed

The following sort algorithms must be timed and analyzed:

- Selection sort
- Bubble sort
- Insertion sort
- Quicksort
- Mergesort
- Heapsort
- An algorithm of your choice comparable to quicksort, mergesort, or heapsort

Requirements

The general requirements of the assignments are as follows:

(1) Generate different sets of random integers in the range 0 to 1 billion. Depending on the performance of the sort algorithms, the size of the different sets of data (and corresponding arrays) will vary. You should start will small sizes (e.g., 10000) and keep increasing the size until the performance of the algorithm starts to degrade. Decide on the most effective way to present the performance data. You should stop experiments with a given sort algorithm or data set if it is taking more than 3 minutes to sort the data set.

(2) Given data sets of different sizes and the recorded time to sort the data sets, use regression to predict how long the next data set will take to execute. Determine if the prediction is valid. You can use the regression feature in Microsoft Excel to find the values of the coefficient and *y*-intercept of the regression line.

(3) Do a quick review of sorting techniques (by referring to material on the Internet) and write code to implement a sorting algorithm that is known to perform well. This algorithm must also be timed and compared with quicksort, heapsort, and mergesort.

Two Questions to Answer

(4) How does the size of the elements to be sorted in an array affect the performance of the sort algorithms? The given sort algorithms all operate on integers. Modify about half of the algorithms to time the algorithms when the elements to be sorted is a struct where the key is a randomly generated integer. The struct can contain one other field, e.g., a character array of 100, 500, or 1000 characters. You can use the same data in all the elements of the array, except the key (the data must be physically copied to each element).

(5) How do the sort algorithms perform if the data is already sorted?

What to Submit / Mark Scheme

You must submit a document, no more than 10 pages, which describes all the experiments you conducted, the results obtained, and explanations for the results.

Section 1 [35 marks]

Describe all the experiments conducted to time the algorithms and show the results obtained for the different data sets. Use charts, tables, and other tools to present your results in a convincing manner. Discuss how effective were the regression predictions.

Section 2 [15 marks]

Give suggestions for the vast difference in performance between the selection sort, bubble sort and insertion sort algorithms, and the others. Also, explain the difference in performance between the quicksort, mergesort, and heapsort algorithms. If you wish, you can refer to material on the performance of sort algorithms (e.g., O (n^2), O ($n$log$n$)). However, you can base your explanations on a study of the inner workings of the different algorithms.

Section 3 [10 marks]

Study the mergesort algorithm. This algorithm is heavily dependent on the merge algorithm which creates two arrays, *B* and *C*, each time the merge is executed. Instead of creating the arrays each time, experiment with creating the array once (globally). Describe the experiments you conducted. Did you get different results? What conclusions can you draw?

Experiment with improving the performance of quicksort. Did you get different results?

Section 4 [10 marks]

Describe the sort algorithm you implemented and show how it compares to the other algorithms.

Section 5 [20 marks]

Describe the experiments you conducted to answer the two questions. What are your answers to the questions posed?

Section 6 [5 marks]

Summarize what you have learned from these experiments involving timing and analysis of the different sort algorithms.

Appendix [0 marks]

On a separate page (not included in the limit of 10), list the team members and describe the configuration of the computer/s on which the majority of the tests were conducted (e.g., type of CPU, CPU speed, memory size, size of hard disk or solid state memory).

Overall [5 marks]

Marks will be awarded for the quality of the overall document, the effectiveness of the analyses conducted, and the rigour of the accompanying explanations.

Next Page: Programming Guidelines

Programming Guidelines

1. Timing a Segment of Code

(a) Insert the following line at the top of your code:

```
#include <ctime>
```

(b) Declare variables of type *clock_t*, e.g.,

```
clock_t start, end;
```

(c) When you need to start timing a segment of code, record the start time as follows:

```
start = clock();
```

Similarly, when you need to stop timing a segment of code, record the end time as follows:

```
end = clock();
```

The time elapsed can be calculated as follows:

```
double elapsed;
elapsed = double (end - start)/CLOCKS_PER_SEC;
```

2. Runtime Problems with Declaration of Large Arrays

You may get problems declaring an array beyond a certain size using a declaration statement such as:

```
int A [1000000];
```

If you get problems, you can create the array using dynamic memory as follows:

```
int * A = new int [1000000];
```

Once the array is created like this, it can be used exactly as a "normal" array (e.g., passed as a parameter, etc.).