University of Calgary
Department of Electrical and Computer Engineering
Summer 2020
ENSF 594

Final Project

Due: Before 11:59 on Saturday August 15

Exercise A: Implementation and Comparison of Sorting Algorithms

Computers are frequently used to sort lists of items, especially when these lists are long. Many sorting techniques are available, some more efficient than others. In this projct, you will implement four different sorting algorithms in a Java language program and compare their performance. The sorting algorithms you will implement are:

1. Bubble Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort

You can base your own code for these algorithms on the code discussed in lectures, or on code found from other sources. Be sure to cite the source for any code you borrow or adapt, putting the citation in comments in your program.

Your program will create an array of integers to be sorted. The user will be able to specify whether the array is filled with integers in random order, ascending order, or descending order. The length of the array is arbitrary and will be specified at run time. The array may contain duplicate numbers. Your program will take this array and sort it into ascending order, outputting the sorted list to a text file, one item per line. The sorting algorithm to use and the name of the output file will also be specified at run time using command-line arguments.

Your program will time how long it takes to sort the array. Be sure to time only the sorting function itself, and not the time taken to fill the array with numbers or do input or output. Your program will print the time in seconds to the screen (standard output). You will use this data to help compare the efficiency of the four algorithms.

Write a program in the Java programming language to implement the above requirements. Your program will be invoked from the command line as follows:

```
java Exercise1 order size algorithm outputfile
```

where *Exercise1* is the name of the file containing executable bytecode for your program, *order* is the order of the integers in the input array (use one of the following strings: *ascending*, *descending*, *random*), *size* is the number of items in the integer array to be sorted, *algorithm* specifies the sorting technique to use (use one of the following strings: *bubble*, *insertion*, *merge*, *quick*), and *outputfile* is the name of the output file where the sorted list will be written to. Be sure your program detects any command-line input errors (for example, a negative number for *size*), printing out an error message, and aborting the program.

Experiments and Data Collection

You will use your program to perform a series of experiments that help you to compare the efficiency of the four sorting algorithms in the best case (input array already sorted into ascending order), worst case (input array in descending order), and average case (input array in random order). At the very least, use the following sizes for the array length: 10, 100, 1000, 10,000, 100,000, and 1,000,000. Determine the timings for all four algorithms using these inputs.

Complexity Analysis

Using the techniques shown in class, do a complexity analysis of each of the four algorithms. Be sure you show all the steps for this, from counting operations to the final big-O characterization of the algorithm.

Report

Create a formal PDF written report that describes at least the following:

1. The experimental method used.
2. The data collected (use tables and graphs to help illustrate this).
3. Data analysis. What does the data tell us about the algorithms?
4. The complexity analysis (show all steps for this).
5. Interpretation. What does the empirical data and complexity analysis tell us about the algorithms? How do the algorithms compare with each other? Does the order of input matter? How do algorithms within the same big-O classification compare to each other?
6. Conclusions. What algorithms are appropriate for practical use when sorting either small or large amounts of data? Are there any limitations with any of the algorithms? How does your analysis support these conclusions?

You can add more to this, if necessary. Be sure the report is complete, organized, and well formatted.

Submit electronically via D2L:

1. Your formal PDF written report.
2. Your source code files. Your TA will run your program to verify that it works correctly. Make sure your Java program compiles and runs without issues.

Exercise 2: Linked Lists and Sorting

The goal of this assignment is to write a Java program that arranges a list of words into separate lists of anagrams. The input is a text file that contains a list of words, each word on its own line. The number of words in the input is arbitrary and could be very large.

The program should print to an output file the lists of anagrams in the following way:

- All the words that are anagrams of each other are displayed together on a single line; any word without any corresponding anagrams is displayed alone on a line.
- The words on each line should be in alphabetic order.
- Lines of words are sorted into ascending alphabetic order according to the first word of a line.
- Words in a line should be separated by a single space.

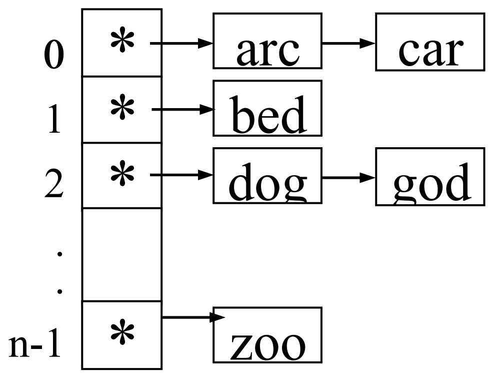
For example, this input:

```
car
dog
bed
stop
god
pots
arc
tops
```

should yield the output:

```
arc car
bed
dog god
pots stop tops
```

You must use linked lists to deal with the arbitrary number of anagrams in a line and use an array of references to keep track of all the linked lists. For example:



An acceptable alternative to using an array is to use a vector, provided you program your own vector class (i.e. you cannot use a class such as Vector or ArrayList from the Java libraries for this).

Adapt the insertion sort given in class so that it works on the items in the linked lists, and adapt the quick sort so that it sorts the array of references. Be sure to cite your sources of information. You must write your own implementation of linked lists and sorting algorithms rather than using calls to a Java library.

Your program should read and store all the words in the file to your data structure first, and then sort as a second step, applying the insertion sort to each row of words, and then the quick sort to the array, using the first word in the list as the sort key. An acceptable alternative to the insertion sort is to insert each word in its row in ascending order as you read it from file. Your program must also print out to the screen the time in seconds it takes to process an input file.

One way to determine if two words are anagrams is to sort the letters in both words. If the sorted words are the same, then the original two words are anagrams of each other. For example, “dog” and “god” are both sorted into “dgo”, so they are anagrams.

Write a program in Java to implement the above requirements. Your program will be invoked from the command line as follows:

```
java Exercise2 inputfile outputfile
```

where *Exercise2* is the name of the file containing executable bytecode for your program, *inputfile* is the name of the input file, and *outputfile* is the name of the output file.

Complexity Analysis

1. What is the worst-case complexity of your algorithm when checking if two words are anagrams of each other? Express this using big-O notation, and use the variable k to represent the number of letters in each word. Support this with a theoretical analysis of your code.
2. Let N be the number of words in the input word list, and L be the maximum length of any word. What is the big-O running time of your program? Justify your answer using both a theoretical analysis and experimental data (i.e. timing data).

Submit electronically via D2L:

1. Your formal PDF written report.
2. The *readme* file.
3. Your source code files. Your TA will run your program to verify that it works correctly.