Sorting and Algorithm Analysis

This is a team assignment. You may only discuss the project with your team, the instructor, and the tutors / STEM Guides.

Purpose

The purpose of this assignment is to gain hands-on experience with big-O runtime complexity analysis by examining three well known sorting algorithms.

Background

This assignment requires you to implement Selection Sort, Insertion Sort, and Quick Sort (usually styled 'quicksort'). Your algorithms will be sorting instances of the Student class. The Student class contains the data values: int id, string name, and string email.

For this assignment you will create a new class to handle all comparisons between student objects. This class will be called the Comparer class. Your algorithms will all be required to use an instance of Comparer to do all comparisons in your sorting algorithms.

Starter Code

The starter code provides you with the following files.

student_data.txt: A text file containing test data. The first line of the file gives the number of student records. **DO NOT MODIFY**.

Students.hpp: This provides the implementation for the Student class. **DO NOT MODIFY**.

StudentList.hpp: This provides a **partial** implementation of the StudentList data structure with some parts missing or only given as stubs. You will be completing this for your project.

Comparer.hpp: This class is a stub with **nearly all code missing** other than method signatures. You will be completing this class also.

main.cpp: This is the driver of the program. The tests are defined here. **You will only modify the three average functions in this file**. You may wish to comment out some tests as you debug your program.

Task

Complete the following tasks.

IMPORTANT NOTE: *It is important that you closely follow these requirements. Read the entire document before you begin. Your project will be graded based on tests written in C++. If your classes do not meet the requirements, they will fail the tests and you will lose points. This is especially true of file names as well as method and function names.*

For this project, assume that all sorts are smallest to largest or increasing order. This is known as **ascending** order.

Header Comment: before you get started, place a header comment in the Comparer.hpp, StudentList.hpp, and main.cpp file that gives:

- The names of all group members.
- The name of your group: Come up with a cool name that all the team members can agree on. This should be no more than 26 characters. **Example**: Team: Flight of the Concords

Implement the Comparer class. Your Comparer class must meet the following requirements.

- The class **MUST** be defined in a file called Comparer.hpp.
- Comparer must have the following private variables:
    - int compares: this will track the number of compares executed.
    - std::string field: This string will identify the field to be used for sorting.
- Give Comparer the following methods:
    - A default constructor:
        - The default constructor should initialize compares to 0.
        - The default constructor should initialize the string variable 'field' to "id".
    - int compareId(Student *s1, Student* s2):
        - compareId should return -1 if the id of s1 is less than the id of s2.
        - Return 1 if s1->getId() > s2->getId().
        - Return 0 otherwise.
        - Inside this method, add 1 to the 'compares' variable anytime it is called.
    - int compareName(Student *s1, Student* s2):
        - Implement compareName similarly to compareId, but use getName rather than getId to access the Student's sort field.
        - Inside this method, add 1 to the 'compares' variable anytime it is called.
    - int compareEmail(Student *s1, Student* s2):
        - Implement compareEmail similarly to compareId, but use getEmail rather than getId to access the Student's sort field.
        - Inside this method, add 1 to the 'compares' variable anytime it is called.
    - int compare(Student* s1, Student* s2):
        - This function will use Comparer's field variable to choose which compare to use.
        - For example, if field is 'name' the compare functions should return the result of compareName(s1, s2).
        - **DO NOT** add 1 to the 'compares' variable in this method. The counting is handled by the specific field compare functions.
    - int getCompares():
        - This method will return the current value of the 'compares' variable.
    - void reset():
        - This method will reset the 'compares' variable to 0.
    - std::string getField():
        - The method should return the current field for sorting.
    - void setField(std::string sortField):
        - this method will set the 'field' variable to the new values of sortField.
        - If sortField is not "email" or "name", set the 'field' variable to 'id'.

Complete the StudentList class. Your Comparer class must meet the following requirements. Note that many of the functions/methods below will need a Comparer passed by reference to do the comparisons.

- **Implement selectionSort** as a class method of StudentList by writing the following functions. The method should sort the 'students' array in ascending order.
    - int findMinIndex(int start, Comparer &comp):
        - This method will use the comparer to find the **index** minimum element of the student array starting at the 'start' index position.
    - void selectionSort(Comparer &comp):
        - This method will use the Comparer and the findMinIndex function to complete selection sort. The exchange method is already given and does not need a Comparer instance.
- **Implement Insertion Sort** using the Comparer class to do the compares. The method should sort the 'students' array in ascending order. Write the following method:
    - insertionSort(Comparer &comp): This method should sort the elements in the array using the insertion sort algorithm.
- **Implement Quick Sort** using the Comparer class to do the compares. The method should sort the 'students' array in ascending order. Write the following method:
    - int partition(int start, int end, Comparer &comp): This method chooses the array element at the 'start' position and designates it the pivot.
        - Once the partition function finishes, the pivot should be in its final position in the array.
        - Any value less than the pivot should be to the left of the pivot. Any value greater than the pivot should be on the right of the pivot.
        - The partition function should return the index of the pivot at its final location in the sorted array.
    - void quicksort(int start, int end, Comparer &comp): This method is used to recursively quicksort the left and right subarrays after calling partition.
    - void quicksort(Comparer &comp): This method is a wrapper for calling quicksort with the appropriate indexes to sort the entire array.
- **Implement isSorted()**. This function should return true if all students in the array are in the proper ascending order.
    - bool isSorted(): have this method return true if all elements are in sorted order and false otherwise.
        - The array is sorted when all consecutive pairs of elements are sorted. This means compare(students[i], students[i+1]) < 0 for all valid i and i+1 indexes.
- **Implement randomize()**: This function should randomly exchange different student pointers in your array.
    - void randomize():
        - This method should loop size/2 times and randomly generate two indexes. Have each index be between 0 and size-1.
        - You should use the rand() % size to generate the indexes.

        - Finally, call exchange for the two generated indexes to move the pointers within the array.

With the previous tools build, you are ready to test your algorithms. Compare them to the Example output that is given at the end of this document.

<u>**Implement average calculating functions**</u> to explore the behavior of the sorting algorithms. In **main.cpp**, you will complete the following functions:

- double averageSelectionSort(int n): This function will run selection sort n times and calculate the average number of compares used each time. In this method do the following:
    - Create a StudentList instance by reading the data file. This is the same method used in the tests.
    - Create a Comparer instance. These **should not** be a pointer but a regular instance that is a local variable of the function.
    - Declare an int count variable to count the number of compares done each time the list is sorted.
    - In a loop, repeat the process below n times:
        1. Sort the list using list.selectionSort(comp). This will pass the Comparer instance 'comp' to the sort function.
        2. Once the objects are sorted, use the Comparer variable's getCompares and add it to the current value of count.
        3. Reset the comparer using the reset() method.
        4. Call list.randomize() to randomize the data in the list for the next work.
    - Once the loop finishes, the 'count' variable will hold the total number of comparisons used for each sort.
    - Finally, return count/n to give the average. Be aware that you must 'cast' one of the integers to a double to give a proper average with factionaly values. Otherwise, you will encounter integer division and only see the whole number portion of the average.
- Repeat this process for both:
    - double averageInsertionSort(int n), and
    - double averageQuickSort(int n)
    - These functions will be largely the same with only the sort function changed.

Write a Summary: Once you have completed the project, provide a 1-page written summary, named **summary.docx** in Microsoft Word format, that includes your answers to the following questions:

1. What is the worst case and average case big-O of the three sorting algorithms?
2. Which algorithm was the easiest to implement? Which algorithm was the most difficult to implement?
3. What was your experience using the Comparer class for sorting? Did it make the implementations harder or easier?
4. What advantages did using the Comparer class give?
5. Based on this assignment, in which cases would you use Selction Sort, Insertion sort, or quicksort. Give some example of when you would choose one over the other.

Submission

Have one group member of your team submit a single zipped file containing the **StudentList.hpp**, **Comparer.hpp**, **main.cpp**, and **summery.docx** to Moodle.

Evaluation

/2	Header
/3	Program Compiles without error
/2	Comparer: compareId implementation
/2	Comparer: compareName implementation
/2	Comparer: compareEmail implementation
/4	Comparer: compare implemented correctly using field.
/5	Comparer: compare counts correct with getting and reset.
/2	Comparer: get and set fields methods are correct.
/3	Sorting: Selection sort implementation
/5	Sorting: Insertion sort implementation
/5	Sorting: quicksort implementation.
/2	Sorting: isSorted implementation.
/2	Sorting: randomize implementation.
/6	Average calculators: functions are correct and in valid range.
/5	Written Summary

50 points total

Example Output

```
#####|========== Id Sorting ==========|#####
===== Selection Sort =====
comparing by: id
compares: 528
isSorted reports:Sorted!
----------

===== Insertion Sort =====
comparing by: id
compares: 295
isSorted reports:Sorted!
----------

===== Quick Sort =====
comparing by: id
compares: 114
isSorted reports:Sorted!
----------

#####|========== Name Sorting ==========|#####
```

```
===== Selection Sort =====
comparing by: name
compares: 528
isSorted reports:Sorted!
----------

===== Insertion Sort =====
comparing by: name
compares: 298
isSorted reports:Sorted!
----------

===== Quick Sort =====
comparing by: name
compares: 120
isSorted reports:Sorted!
----------

#####|========== Email Soring ==========|#####
===== Selection Sort =====
comparing by: email
compares: 528
isSorted reports:Sorted!
----------

===== Insertion Sort =====
comparing by: email
compares: 302
isSorted reports:Sorted!
----------

===== Quick Sort =====
comparing by: email
compares: 133
isSorted reports:Sorted!
----------

#####|========== Average Compares Testing ==========|#####
===== Selection Sort average =====
Average Test Passed: Estimate in valid range
===== Insertion Sort average =====
Average Test Passed: Estimate in valid range
===== Quicksort average =====
Average Test Passed: Estimate in valid range
```