CSCI 335 Fall 2022

HW 4: Sorting and Heaps

100 points (77 points deliverables, 15 points for design, 8 points Heaps)

EC: 10 points

Due 11pm, December 1, 2022

This is an individual assignment

Instructions:

1. **Read and follow the contents of Fall 2022 335 Programming Rules document on the blackboard (Course Information Section).**
2. **Read the assignment described below.**
3. **Submit only the files requested in the deliverables at the bottom of this description to Gradescope by the deadline.**
4. **Academic Integrity Policy will be strictly followed.**

Learning Outcome: The goal of this assignment is to use and compare various sorting algorithms.

In this assignment, you are going to compare various sorting algorithms. You will also modify the algorithms in order for a `Comparator` class to be used for comparisons. You will then further experiment with algorithmic variations. **For this assignment, there are 10 visible tests and 8 invisible/hidden tests.**

Files provided:

- `Sort.h` (Chapter 7)
- `test_sorting_algorithms.cc`

Q1: Sorting Total: 77 points

Q1: Part 1 (55 Points)

******** **Step 1** ********* (10 Points)

Task 1.1 You should write a function that verifies that a collection is in sorted order.

```
template<typename Comparable, typename Comparator>
bool VerifyOrder(const vector<Comparable> &input, Comparator less_than)
```

The above function should return true if and only if the input is in sorted order according to the Comparator. For example, in order to check whether a vector of integers (`vector<int> input_vector`) is sorted from smaller to larger, you need to call:

```
VerifyOrder(input_vector, less<int>{});
```

If you want to check whether the vector is sorted from larger to smaller you need to call

```
VerifyOrder(input_vector, greater<int>{});
```

This function should be placed inside `test_sorting_algorithms.cc` `All deliverables are described at the end of the file.`

Task 1.2 Next, you should write two functions, one that generates a *random vector* and another that generates a *sorted vector*. The sorted vector should generate a vector of increasing or decreasing values based on bool `smaller_to_larger`. You will use both of these for your own testing purposes.

The function signatures should be as follows.

1) `vector<int> GenerateRandomVector(size_t size_of_vector)`
2) `vector<int> GenerateSortedVector(size_t size_of_vector, bool smaller_to_larger)`

Task 1.3 Next, write a function that computes the duration given a start time and stop time in nanoseconds. Hint: Look at the `TestTiming` function given to you in `test_sorting_algorithms.cc`:

The function signature should be as follows.

```
auto ComputeDuration(chrono::high_resolution_clock::time_point start_time,
chrono::high_resolution_clock::time_point end_time)
```

These functions should be placed inside `test_sorting_algorithms.cc` `All deliverables are described at the end of this document.`

******** Step 2 ********* (45 Points)

You will now modify several sorting algorithms provided to you in `Sort.h`. You will modify: **`Heapsort, Quicksort, and Mergesort.`**

Task 2.1. You should modify these algorithms so that they each take a Comparator with their input.

The signatures for these sorts should be:

```
template <typename Comparable, typename Comparator>
void HeapSort(vector<Comparable> &a, Comparator less_than)

template <typename Comparable, typename Comparator>
void QuickSort(vector<Comparable> &a, Comparator less_than)

template <typename Comparable, typename Comparator>
void MergeSort(vector<Comparable> &a, Comparator less_than)
```

You **will** have to modify multiple functions, helpers and wrappers to make this fully operational without error.

These functions should be modified and kept inside `Sort.h`
`All deliverables are described at the end of this document.`

******** Step 3 ********* (Grade for Step 2 and Step 3 are combined)

Now that those two steps are finished, you will move on to testing.

Task 3.1 You should now create a driver program within **`test_sorting_algorithms.cc`** that will test each of your modified sorts with different inputs.

The program will be executed as follows:

```
./test_sorting_algorithms <input_type> <input_size> <comparison_type>
```

where **`<input_type>`** can be **`random`**, **`sorted_small_to_large`**, or **`sorted_large_to_small`**, **`<input_size>`** is the number of elements of the input, and **`<comparison_type>`** is either `less` or `greater`.

For example, you should be able to run

```
./test_sorting_algorithms random 20000 less
```

The above should produce a random vector of 20000 integers and apply all three algorithms using the **`less<int>{}`** Comparator.

You can also run the following:

```
./test_sorting_algorithms sorted 10000 greater
```

The above will produce the vector of integers containing 1 through 10000 in that order and will test the three algorithms using the **`greater<int>{}`** Comparator.

This driver should be implemented inside the **`sortTestingWrapper()`** function.
The formatting for driver output is shown at the bottom of the file.

Note: *The format presented is an example of how you should test your functions. It serves as a good base for understanding how the different sorts will vary in runtime with different types of inputs. You will not be constrained (or graded exactly on how) you implement this step, but doing so would help you and us verify the accuracy of your work.* ***(You still must create a driver that functions as the one described, but it will not be autograded for formatting, it will be manually looked at.)***

Task 3.2 (Optional Extra credit: 9 points) Modify the shellsort sorting algorithms provided to you in `Sort.h`. We should also be able to test it as you did the other sorting algorithms in Step 3.1.

You should modify this algorithm so that they each take a Comparator with their input.

The signature should be:

```
template <typename Comparable, typename Comparator>
void ShellSort(vector<Comparable> &a, Comparator less_than)
```

Q1: Part 2 (22 points)

Task 4 In this part, you will implement variations of the quicksort algorithm. You will investigate the following pivot selection procedures.

1. Median of three (**already implemented in part 2)**
2. Middle pivot (always select the middle item in the array)

3. First pivot (always select the first item in the array)

Although median of three (1) is already implemented in the file, you will use it for further comparisons in this part.

The following two quicksort implementations, **middle pivot**, and **first pivot**, should have wrappers with the following signatures that then call the full implementations.

```
//Middle Pivot Wrapper
template <typename Comparable, typename Comparator>
void QuickSort2(vector<Comparable> &a, Comparator less_than)

//First Pivot Wrapper
template <typename Comparable, typename Comparator>
void QuickSort3(vector<Comparable> &a, Comparator less_than)
```

Note: *these are just the wrappers. You have to write the actual quicksort functionality in another function called by these (similar to the original quicksort provided).*

In order to test these functions, you will add to the output of the driver described in Step 3. The full format is shown below deliverables. You should also validate the input and provide error messages such as "Invalid input" or "Invalid size" when necessary.

Deliverables: You should submit these files:

- `README` file
- `Sort.h` (modified)
  - All sort modifications and additions should be kept within this file.
- `test_sorting_algorithms.cc` (modified)
  - `VerifyOrder()`
  - `GenerateRandomVector()`
  - `GenerateSortedVector()`
  - `ComputeDuration()`
  - `sortTestingWrapper()`

****** Important Note: A large amount of this assignment will be manually checked and graded. We will run your sorts and implemented functions in the autograder, but the sort modifications will be verified manually. *******

Driver Formatting

The full driver format should be as follows: (example shown with function call **`./test_sorting_algorithms random 20000 less`*)*** *Note:* The number output next to "Verified" is the boolean output of the function `VerifyOrder()`. If that value is 0, your sort did not work as intended.

```
Running sorting algorithms: random 20000 less

Heapsort
Runtime: <X> ns
```

```
Verified: 1

MergeSort
Runtime: <X> ns
Verified: 1

QuickSort
Runtime: <X> ns
Verified: 1

Testing Quicksort Pivot Implementations

Median of Three
Runtime: <X> ns
Verified: 1

Middle
Runtime: <X> ns
Verified: 1

First
Runtime: <X> ns
Verified: 1
```

Q2: Heaps — Total: 8 points

For this question, match the question with one of the answers given in the “Options” list. Please fill in your answers electronically using the space provided, save as pdf with the file name “**HW4-Q2_Heaps.pdf”** and submit in the respective Gradescope submission.

A complete binary tree of N elements uses array positions 1 to N. Suppose we try to use an array representation of a binary tree that is not complete. Determine how large the array (in terms of N, last 3 answers are in terms of big-Oh) must be for the following (fill in the answers in the right column below)

A binary tree that has two extra levels (that is, it is slightly unbalanced)	4N
A binary tree that has a deepest node at depth 2 logN	O(N^2)
A binary tree that has a deepest node at depth 4.1 log N	O(N^4.1)
The worst-case binary tree	O(2^N)

Options: N, $N^{1.5}$, N^2, $N^{4.1}$, 4N, $N^{2.5}$, 2^N, $N^{0.5}$, 4^N