CPSC223
Assignment 3
Sorting

Goals:
To gain a better understanding of n^2 sorting algorithms, empirically test what we have learned in theory, make use of inheritance in C++, and build a basis for an ongoing study of more advanced sorts.

Task:
Implement Bubble Sort (2 versions: normal and “short-circuited”), Insertion Sort, and Selection Sort. Test each of these sorts through empirical testing methods and plot the results on a graph for comparison. Also to be included in the comparison is the built-in sort std::sort() which is provided.

You are provided with some starter code, and a Makefile to aid in your benchmarking. It is set up in such a way that it runs each sort 5 times, then takes the average of those 5 runs and produces a time.

Your job is to trace through the starter code and implement the missing functions. You will then create a vector of sort pointers to run a suite of benchmarks on your implementation.

Collect the output data from **only the random number dataset** for each sort and plot them on a single graph. You should end up with a single graph with 5 curves, one for each sort. Use appropriate axis scales to give a good visual representation of the behavior as n grows larger and larger.

Some recommended tools for graphing are Excel, Desmos, or gnuplot. You may add your own helper function to output the results in an easier to plot way, either by redirecting the console output to a file in linux using the >> command, or by writing to a file directly within your code. **Do not write to a file for your final submission.**

Depending on the speed of your system you may want to experiment to find a good size for your vectors. You want to get at least three good data points for each sort so you can try to graph them to showcase their “Big-O” behavior. The more points you plot, the more accurately you can determine a trendline, but beware the curse of dimensionality.

Below is a screenshot of some test runs using the built-in std::sort() on my m1 mac. I needed to run fairly large n values to start seeing larger runtimes. Feel free to add your own n values as needed to better time your runs based on your system (they don't necessarily have to be powers of 10).

```
./main 40000
--------------------------------------------------------------------------------
Sort type: std::sort
std::sort, input data: Ascending, n=40000, average time: 0.0030 secs
std::sort, input data: Descending, n=40000, average time: 0.0024 secs
std::sort, input data: Random, n=40000, average time: 0.0066 secs
--------------------------------------------------------------------------------
./main 80000
--------------------------------------------------------------------------------
Sort type: std::sort
std::sort, input data: Ascending, n=80000, average time: 0.0063 secs
std::sort, input data: Descending, n=80000, average time: 0.0050 secs
std::sort, input data: Random, n=80000, average time: 0.0140 secs
--------------------------------------------------------------------------------
./main 100000
--------------------------------------------------------------------------------
Sort type: std::sort
std::sort, input data: Ascending, n=100000, average time: 0.0077 secs
std::sort, input data: Descending, n=100000, average time: 0.0060 secs
std::sort, input data: Random, n=100000, average time: 0.0177 secs
--------------------------------------------------------------------------------
./main 1000000
--------------------------------------------------------------------------------
Sort type: std::sort
std::sort, input data: Ascending, n=1000000, average time: 0.0871 secs
std::sort, input data: Descending, n=1000000, average time: 0.0673 secs
std::sort, input data: Random, n=1000000, average time: 0.2089 secs
--------------------------------------------------------------------------------
./main 10000000
--------------------------------------------------------------------------------
Sort type: std::sort
std::sort, input data: Ascending, n=10000000, average time: 1.0726 secs
std::sort, input data: Descending, n=10000000, average time: 0.8095 secs
std::sort, input data: Random, n=10000000, average time: 2.4885 secs
--------------------------------------------------------------------------------
```

Submission:

- Your program should compile and run without issues.
- There is a built-in sort checker that will throw an assertion if a sort isn't working properly
- By default when we type "make" it will run the suite of sorts with n=1000
- You may modify your program to write to a file for the data collection task but for your final submission make sure its outputting to the console as mentioned in the above bullet point.

Report:

Include a professional pdf report with your results, include your graph in the report.

Answer the following questions in narrative style:

1. Were the empirical test results surprising in any way, or did they conform to your expectations based on what you know about the sorts?
2. Which sort performed the fastest on the random dataset? Why?
3. Which sort performed the slowest on the random dataset, and why?
4. What is a pure virtual function in C++, and why did we choose to use it (doSort) in this assignment?
5. What is the difference between a pure virtual function and a virtual function?
6. Using what you see from the empirical tests, and from your own research (internet searching) what is the underlying sort for the built-in std::sort() in c++?
7. Among bubble sort, selection sort, and insertion sort, describe when would you want to use each sort over the others? Consider things like the underlying data structure, the data types, how the computer interacts with memory, and complexity to implement / understand each sort.
8. (reflection) After completing the assignment, describe:
    a. What was the most challenging aspect
    b. What was the least challenging aspect
    c. If you were to do the assignment again, what would you do differently and why?
9. Include a screenshot of your output from valgrind showing no memory leaks.

Notes:

There is an assert statement to check if the list is sorted, if you see assertion failed message it means there may be a problem with your sorting implementation.

Your program should not have any memory leaks. Use valgrind to ensure that all memory is freed.

$ valgrind ./main

Rubric:

100 Points total

Code compiles	10 points (0 points for non compiling code)
Bubble Sort	10 points
Bubble Sort Optimized	10 points
Selection Sort	15 points
Insertion Sort	15 points
Report	25 points
Output is correct	5 points
Code commenting and style	5 points
Valgrind screenshot - no memory leaks	5 points