Lab Exercise #8
Due Date: Apr 19th
Sorting

Description:
For this lab exercise, you will implement the Insertion Sort Algorithm and the Radix Sort Algorithm. Insertion sort repeatedly takes one element at a time, placing it in its correct position among already sorted elements, efficient for small datasets. Radix sort sorts integers by grouping digits, starting from the least significant to the most significant, ensuring stable sorting.

starter.zip (Starter Code):
The following is a brief description of the starter code provided to you:

- **sort.cpp**
    - **void radixSort(int * arr, int n):** Given an input (unsorted) array ***arr***, and an integer **n** representing the size of the array, you must implement the radix sort algorithm.
    - **void insertion_sort(int * arr, int n):** Given an input (unsorted) array ***arr***, and an integer **n** representing the size of the array, you must implement the insertion sort algorithm.
    - For both of these functions, the arr passed in as the parameter must contain the final sorted sequence of numbers.
- **SortedPriorityQueue.h**
    - You must use the SortedPriorityQueue class to implement the insertion sort algorithm. Make sure that your pq_delete() function runs in O(1) time. You can use the startIndex member variable in the SortedPriorityQueue class to keep track of the start of the array to avoid an O(N) pq_delete().
- **main.cpp**
    - You can use this file to test each of your sorting functions.

Contract (TASK):
Your task is to implement the insertion sort and radix sort functions. You have been provided with their non-functional definitions. **You must implement these function definitions in the given sort.cpp and SortedPriorityQueue.h files**, by replacing the current non-functional definitions with your functional implementations.

[Advisory]
- **You can use your implementation of the SortedPriorityQueue class from PA3 (incorporating the change mentioned above if you've not already done so).**

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<LE8>.zip** you must submit the following file(s) to Canvas:

- **SortedPriorityQueue.h** (containing the functional method definitions of the SortedPriorityQueue class)

- **sort.cpp** (containing the functional definitions of the insertion sort and radix sort)

Do not use angular brackets in the name of the submission folder.

Grading:
Coding (20 points)
The insertion sort and radix sort will be tested using an automatic testing infrastructure. You will be provided with the testing infrastructure which you can use to evaluate your implementations.

The grading rubric is as follows:
- Insertion Sort
  - 10 points
- Radix Sort
  - 10 points

Each of these functions will be timed to ensure you implement them in the right manner. Insertion sort should be completed in O(N^2) time (with SortedPriorityQueue.h - pq_insert() being O(N) and pq_delete() being O(1)). The time complexity of radix sort must be O(N * D), where N is the number of elements and D is the number of digits in the largest number. If one of your sorting functions does not meet the appropriate timing expectations, you will be penalized 5 points for that sorting function. The recorded timings on our machines are as follows:
- Radix Sort
  - Mac M1: 0.036s
  - WSL: 0.23s
- Insertion Sort
  - Mac M1: 15.37s
  - WSL: 46s

testInfrastructure.zip (Test Methodology):
In order to familiarize you with the auto-grading environment on our end we are providing you with an automated test infrastructure that can be used to check the correctness of your code. The testing infrastructure is constructed as follows:
- **test_sort.cpp**: This cpp file defines all the test functions that have been mentioned in the grading rubric above. It also runs these tests and outputs your final score.
- **test.py**: This file compiles test_sort.cpp in order to produce the executable. It then runs this executable which will indicate your score (as well as the test cases you have passed or failed).

Note: There is a possibility that one or more of your methods causes a segmentation fault which in turn causes the test script to crash when calling that particular method. In such a case, we will not be able to calculate your score for the assignment and you will be assigned a default grade of 0. After the grades are posted, you will be given 1 week to rectify your error(s) and resubmit the assignment. You will be subject to a 25%

penalty on the grade obtained from the resulting resubmission of the assignment.

How to test your program natively (run your own test cases):
You can use main.cpp provided in the starter.zip folder to test your sorting functions. You can then test the functionality of your methods by calling these methods and checking whether what they return is what you expect.
To compile your own tests in main.cpp, **you can use the makefile provided within starter.zip to compile your program.** In your terminal:

- Run **make** on the terminal (this will create an executable named main)
- Next, run **./main** on your terminal to run your native tests

How to use testInfrastructure:
To check your score using testInfrastructure, you can follow either of the two options mentioned below:

- **Option 1:** Move your sort.cpp and SortedPriorityQueue.h files into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move your sort.cpp and SortedPriorityQueue.h files into the testInfrastructure directory. **Ensure that the makefile in this directory is the one given to you originally in the testInfrastructure.zip folder and not in the starter.zip folder.** In your terminal:
    - Run **make** in order to compile the program (which will compile test_sort.cpp and create an executable named ./test_sort).
    - Next, run **./test_sort** on your terminal to actually run the tests and output your score.