Lab Exercise #9
Due Date: Apr 21st
Merge Sort Using Threads

Description:
For this lab exercise, you will implement the merge sort algorithm using threads and compare its performance against the single-threaded configuration. Merge Sort with threads divides the array, sorts each half *concurrently*, then merges sorted halves; it utilizes multithreading for faster performance on large datasets.

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

- **merge_sort.cpp**: You have been provided with the **void mergeSort(int* arr, int start, int end, int level)** function. The only difference between this function and the merge sort function you will implement in PA5 is the "level" parameter.
  - In the context of the provided code, the "level" variable represents the depth of recursion in the merge sort algorithm. Each recursive call to the merge sort function divides the array into halves, and the "level" parameter decrements by one.
  - When the "level" variable reaches zero, the recursion stops and no more threads are spawned for further divisions of the array. This allows controlling the depth at which threads are created, effectively limiting the parallelism to a certain depth of the recursion tree.
  - Adjusting the "level" variable enables fine-tuning the balance between parallelism and overhead associated with thread creation.
- **main.cpp**
  - You can use this file to test your merge sort function.

Contract (TASK):
Your task is to implement the merge sort function using a multi-threaded approach. You have been provided with the non-functional definition. **You must implement this function definition in the given merge_sort.cpp file,** by replacing the current non-functional definition with your functional implementation.

[Advisory]

- **You may use your (default single threaded) implementation of the merge sort function from PA5 to get started.**

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<LE9>.zip** you must submit the following file(s) to Canvas:

- **merge_sort.cpp** (containing the functional method definitions of merge sort)

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

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

The grading rubric is as follows:

- **Merge Sort Single Threaded:** 5 points
  - The test script will first call the mergeSort function with level 0 indicating that you should not use threads in the algorithm at all (i.e. a single threaded merge sort)
- **Merge Sort Multi Threaded:** 15 points
  - The test script will then make a call to mergeSort with the level parameter set to **4**. This means that your implementation of the merge sort algorithm must spawn threads to perform sorting until it reaches the fourth level of recursion.

The single-threaded and multi-threaded merge sort will be timed in order to check that you are correctly implementing the multi-threading (over 1,000,000 randomized input elements). If you do not meet the timing requirements, you will be penalized 20 points. The following times have been measured on our machines:

- Single Threaded Merge Sort (called with level = 0)
  - Mac M1: 0.156 seconds
  - WSL: 0.43 seconds
- Multi Threaded Merge Sort (called with level = 4)
  - Mac M1: 0.04 seconds
  - WSL: 0.25 seconds

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:

- **merge_test.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 merge_test.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 merge_sort.cpp file into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move your merge_sort.cpp file 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 merge_test.cpp and create an executable named ./merge_test).
    - Next, run **./merge_test** on your terminal to actually run the tests and output your score.