# CSC 140: Advanced Algorithms
# Spring 2024
# Third Assignment: Sorting and Divide-and-Conquer

In this assignment you will implement and experiment with four different sorting algorithms using actual timing. The algorithms are InsertionSort, MergeSort, QuickSort and LibSort (the sorting algorithm in your language's standard library). Your job will be translating the pseudocode given in class into actual code, and then experimenting with the different algorithms using different input types and sizes. Follow the pseudocode presented in class as closely as possible. Note though that since each student will have his/her own way of implementing the details, every student's code should look different.

This assignment consists of two parts. In the first part, you will be comparing the performance of the four algorithms mentioned above. In the second part, you will be doing a focused experimental study of Quicksort. In the second part, you can earn up to 30% of extra credit if you do the extra work described below.

The attached source file sort.cpp/Sorting.java contains a program that reads two inputs: a letter specifying which sorting algorithm to use (I for InsertionSort, M for MergeSort, Q for Quicksort, L for LibSort) and an integer specifying the input size. The program will automatically generate four different types of input arrays (sorted, nearly sorted, reversely sorted and random) with the specified size and apply the specified sorting algorithm to each array. The program will then report the execution time in milliseconds for each run.

The first step in this assignment is writing the code to implement each sorting algorithm (see where it says "Write your code here" in sort.cpp/java). After adding your code to the source file, compile it. Before you do the production runs using the large input sizes listed below, do four test runs (one for each sorting algorithm) using a small input size of 10 to verify that your program correctly sorts all kinds of input data.

**Part 1: Algorithm Comparison**

Once your small-input tests work correctly, change the value of the global variable OUTPUT_DATA to *false* so that it does not ridiculously print the contents of the huge arrays used in the production runs. Also, it is ***highly recommended that you enable compiler optimizations when you compile your code for the production runs***.

Now, run each algorithm (InsertionSort, MergeSort, QuickSort, LibSort) using the following three input sizes:
10000
100000
1000000

For Quicksort, first try the basic implementation that always uses the last element as a pivot (no randomization). That should result in a "***stack overflow" error***, why? Make sure you comment on this point in your report, but ***don't present any numerical results for this implementation***. After verifying that this implementation indeed causes a stack overflow, fix this problem using the randomized version of Quicksort studied in class (pick an index at random and swap it with the element in the last position, and then use the element in the last position as a pivot). List all of your results in the tables provided in the attached report template.

Important Note: After running the algorithms using an input size of 10000, calculate the expected running times for the other input sizes so that you have an estimate of the time that you will need to wait. If the expected running time is too long, run it overnight or let it run while you are working on something else.

**Part 2: QuickSort Focused Study**

After completing the above comparative study of the four algorithms, you will need to conduct a focused study of QuickSort. First, modify the QuickSort code such that it computes and prints the recursion depth. Then run QuickSort using the following implementations. In this focused study, use random input of size 1000000. You may comment out the code that runs the other input types.

1. The simple randomized pivot selection scheme discussed in class (pick an index at random and swap it with the element in the last position, and then use the element in the last position as a pivot).
2. The median-of-three method for selecting the pivot, which can be described as follows. Pick three numbers in the array at random and then use their median as the pivot (swap it with the element in the last position and then proceed as usual).
3. Use the simple randomized pivot selection scheme described in Part 1, but when the array size is 40 or smaller, call InsertionSort instead of recursively calling QuickSort.

Similar to Part 1, list your results in the table provided in the attached report template, and then discuss them.

**Extra Work**

Try to come up with your own ideas for speeding up Quicksort by minimizing the recursion depth or possibly making other enhancements. Implement these ideas and experiment with them using a random input of size 1000000. Use the library sorting algorithm as a reference for comparison. Try to match its performance if you can. Present and discuss your results in your report. If you do this part, you will get up to 30% of extra credit. The amount of extra credit that you will get will depend on the quality of your work and the depth of your discussion.

**Submission**

Submit both the source code (sort.cpp/sort.java with your added code in it) and your completed report on Canvas.

Notes:

1. Your code must compile and run successfully in the standard ECS computing environment, or you may not get any credit.
2. It is very important that you write the report in your own words. Similarities between two reports will be considered cheating.