Motivation:

For many years it was believed that the fastest algorithm to bring the elements of an array into nondecreasing order had asymptotic growth proportional to the n2, where n is the size of the data to be sorted. However, people have now invented sorting algorithms that have asymptotic growth proportional to n\*log(n), such as Merge sort and Quick sort. This doesn’t mean that we should never use n2 sorting algorithms. n2 algorithms are more intuitive and are actually faster than n\*log(n) algorithms for small enough data sizes. We will see an example of how large the data must be for n\*log(n) algorithms to be faster than n2 algorithms in this experiment. Other than size, the state of the data determines which algorithm is optimal. Each algorithm has a best case and a worst case input state, but the average case is the same for all algorithms that solve the same problem. Some algorithms will change only a little for extreme cases, but some will lose or gain functionality for extreme cases. The average case is randomly generated data for all sorting algorithms. For the three sorting algorithms in this experiment the best case is data that is already sorted (nondecreasing) and the worst case is data that is sorted in the reverse order (nonincreasing). However, for some sorting algorithms these will not be the best and worst cases. For example, one version of Quick sort picks the middle element as a pivot. Reverse sorted data would take it n\*log(n) complexity, but if the data decreased from the middle outward in both directions then this Quick sort algorithm would have n2 complexity, so the worst case for this version of Quick sort is not nonincreasing data.

We will empirically compare the running time of one n\*log(n) algorithm, Merge sort, and two n2 algorithms, Insertion sort and Bubble sort. We will count CPU cycles between the beginning and end of each algorithm execution for many different values of n and compare the data on a graph. CPU cycles are an approximation to the comparisons and assignments in the algorithms, since each comparison and assignment takes a number of CPU cycles. CPU cycles are more relevant than the actual number of comparisons and assignments, since the whole purpose of algorithms is to run computers. In a world where data sizes are growing exponentially, using the faster algorithm is critical.

Background:

Merge sort is a sorting algorithm that merges sorted subarrays together until the entire array is sorted. Merge sort starts with subarrays of size 1, which are already sorted. It merges sorted subarrays into larger sorted subarrays until it merges two sorted subarrays into the entire array.

Insertion sort inserts each element into a sorted array, starting with the left and moving right. Insertion sort uses a linear search from the right to find the correct position at which to insert an element moving each element one space to the right in the array to make a space to the element being inserted.

Bubble sort traverses the array n times, swapping each pair of adjacent elements if they are in reverse (decreasing) order. Each traversal results in the largest element not previously sorted being added to the sorted end of the array after each traversal.

Insertion sort and Bubble sort are simpler and more intuitive than Merge sort, but we will see that Merge sort is much more efficient than Insertion sort or Bubble sort.

Procedure:

A program was written in the language C++. The program empirically compared Insertion sort, Bubble sort, and Merge sort. The program tested Merge sort, Insertion sort, and Bubble sort on data sizes that are the closest whole numbers to the powers of the square root of two (sqrt(2)). The intent of using powers of the sqrt(2) was to see if powers of 2 affect merge sort. Every other data size is a power of two, so if Merge sort is significantly affected by powers of 2 it will show up in the graph. However, we will find that Merge sort is not significantly affected by powers of 2. The efficiency of each algorithm was compared via the rdtsc CPU cycle counter. The C++ program outputted the data through STDOUT. The program was compiled into the unix executable file a.out and the unix command ./a.out > CPUcounts.txt was used to collect the data into a text file. Then the data was imported into Microsoft Excel. The data was plotted on both linear and logarithmic scales. Here it is:

Findings: