Motivation:

For many years it was believed that the fastest algorithm to bring comparable elements of an array into sorted order had asymptotic growth proportional to 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 faster than n\*log(n) algorithms for sufficiently small 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. For sorting algorithms, the average case is randomly generated data. In this experiment, we will guess that 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 are not the best and worst cases.

We will empirically compare the running time Timsort, Merge sort, 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. The graph will reveal how the algorithms perform compared to each other and how the algorithms react to special cases. 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.

Timsort is a hybrid of Merge sort and Insertion sort. Timsort uses an optimized combination of Merge sort and Insertion sort to take advantage of Merge sorts low asymptotic complexity and Insertion sort’s low complexity on small problem sizes. Timsort is recursive. When called on a sufficiently small problem size, Timsort uses Insertion sort. When called on a problem size above a certain threshold, Timsort begins by applying Insertion sort on sufficiently small subarrays, then merges sorted subarrays together until the entire array is sorted. In this experiment Timsort’s threshold is set at 25.

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 and Timsort, but we will see that Merge sort and Timsort are much more efficient than Insertion sort and Bubble sort for large data sizes

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. To make it more fair, the exact same data was used on all three algorithms.

To insure that the algorithms weren’t cheating, the program checked to see if the data was sorted after each execution of a sorting algorithm. The program would display an error message and terminate if an algorithm failed to sort the data. 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. This was done on a MacBook Pro with a single core processor, which may have helped the accuracy of the CPU cycle counts. Then the data was imported into Microsoft Excel. The data was plotted on both linear and logarithmic scales. Timsort is represented by the color green, Merge sort by blue, Insertion sort by orange, and Bubble sort by gray. Test cases of randomly sorted data are represented by circles, presorted (nondecreasing) by squares, and reverse sorted (nonincreasing) by triangles.

Problems encountered:

The biggest problem was noise in the data. We don’t know what is causing this noise. One possibility we considered was that the launching of the program put some tasks on the CPU’s task list which slowed down the execution of the program. If this were the case, then the tasks would go away given sufficient time. So we put some sleep statements in the code. But the sleep statements did not affect the data in the output.

Tests of the code:

Throughout the code we placed checks using the isSorted() function. We used the isSorted() function to see if the data was sorted or reversely sorted when it was supposed to be sorted or reversely sorted. If the test failed it would throw and Error object with a message, which would be caught and the message outputted to STDOUT. In order to make sure the program was actually willing to throw we put some test errors in the program. When the program threw what it was supposed to throw we remembered to remove the test errors.

The normal program output was 6 columns wide, with the CPU cycle count data from Timsort, Mergesort, Insertionsort, and Bubblesort in that order, followed by the exponent of sqrt(2) that produced the data size, and the data size itself. These columns were rows long. There were 26 rows for random data, 26 for presorted data, and 26 for reverse sorted data. The coherence of the data makes us sure that the program is correct.

Conclusion:

For data sizes larger than about 100 elements, Merge sort and Timsort are faster than both Insertion sort and Bubble sort. For data sizes smaller than about 100 elements, Insertion sort, Bubble sort, and Timsort are all faster than Merge sort. Insertion sort is faster than Bubble sort for all data sizes of random data, but not by much. Timsort is close to Insertion sort for small data sizes but Timsort is close Merge sort for large data sizes. For data that is already sorted, Insertion sort is extremely fast. On small data sizes, Timsort takes advantage of its Insertion sort parts and is also extremely fast on presorted data. But on large data sizes, Timsort does not handle presorted data like Insertion sort. Merge sort performs better on reversely sorted data than random data, unlike Insertion sort and Bubble sort. This shows that reversely sorted data is not the worst case for Merge sort. For data sizes larger than about 1,000, Timsort also performs better on reversely sorted data than random data. Finally, there is no noticable effect of powers of 2 on Merge sort or its derivate Timsort.