**Abstract**

**Introduction**

Sorts are an exceedingly important tool in a coder’s arsenal and present a challenge to find a solution that takes any set of data and puts it into a normalized form for easy use by other algorithms. A wide variety of sorts have surfaced over the years that range in implementation complexity, memory usage, start up time, and overall execution efficiency. The purpose of this study is to determine the effectiveness of various basic sorts and provide concrete evidence to support theoretical comparisons of those sorts. We used three commonly known sorts in our test: The modified Bubble sort, the Selection sort, and the Insertion sort.

**Problem Analysis and Hypothesis**

To form a proper hypothesis, we analyzed each of these algorithms using Big O notation standards. For our purposes, we used a conversion ratio of 1:3 for swaps to comparisons to ease the assessment of each sort method.

The first sort we analyzed is the modified bubble sort. It sorts by scanning through the array of values, comparing each value with the next and performing swaps where necessary. It continues scanning through the array until it passes through the entire array without performing any swaps. Below is pseudo code for a modified bubble sort.

swapped = true;  
 for (i=N-1; i>0; i--) //N-1 loops  
 if (!swapped) break  
 swapped = false  
 for (j=0; j<i; j++) // N-1 + N-2 +…+2+1  
 if (a[j] > a[j+1]) // = N(N-1)/2 comparisons  
 swap (j, j+1); // N(N-1)/4 swaps (random data)  
 swapped = true

As noted, for a random data set of size , computations will be performed. In a random data set we can assume that approximately half of the comparisons will result in a swap, however not all data sets are random. In a best case scenario, the data would already be in the desired order and no swaps would be performed, triggering a loop break. This would result in computations. In a worst case scenario, the data would be in reverse the desired order and every comparison would cause a swap. This would result in computations.

Next we analyzed the selection sort. It sorts data by locating the lowest value in the set before performing a single swap to place it in the correct position. Because of this, the selection sort performs significantly fewer swaps than the modified bubble sort. Pseudo code for the selection sort follows.

for (i=0; I < N-1; i++) // N-1 loops  
 minIndx = i  
 for (j=i+1; j <N; j++) // N-1 + N-2 +…+ 2 + 1  
 if (a[j] < a[minIndx]) // = N(N-1)/2  
 minIndx = j  
 swap (i, minIndx) // N-1

Unlike the modified bubble sort, the form of the data before the sort does not affect the execution time of the sort. For any data set of size , random or otherwise, computations will be performed.

The last sort we will analyze is the insertion sort. It sorts by maintaining a sorted subarray at the beginning of the array. Each value is taken one at a time and placed in the appropriate position inside this subarray, shifting the necessary members down to make room. Pseudo code for an insertion sort follows.

int i, j;

for (i=1; i< N; i++) // N-1 Loops

long temp = a[i]

j = i

while (j>0 && a[j-1]>=temp) // N-1 + N-2 +…+ 2 + 1

a[j] = a[j-1] // = N(N-1)/4 (random data)

j--

a[j] = temp

As noted, for a random data set of size , computations will be performed. In a random data set we can assume that the next value would cause about half of the sorted subarray to be shifted on average, however not all data sets are random. In a best case scenario, the data would already be in the desired order and no shifting would be performed, resulting in computations. In a worst case scenario, the data would be in reverse the desired order and every member of the sorted subarray would need shifted. This would result in computations.

Theoretical Runtime

|  |  |  |  |
| --- | --- | --- | --- |
| Case\Sort | Bubble Sort | Selection Sort | Insertion Sort |
| Best |  |  |  |
| Random |  |  |  |
| Worst |  |  |  |

The table above contains the theoretical runtimes of each of the sorting methods under three different conditions of the starting data. Based on this information, we hypothesized that under the best case bubble and insertion would perform equally better than selection and that with random and worst case data the insertion sort would perform far better than its counterparts. The theoretical runtimes above led us to draw these conclusions based upon their leading term degree and coefficient.

**Methods**

To test our hypothesis, we wrote a java program to compare the real operation time of each of the sorts. Data sets used to test the sorting methods were generated under rules based on the cases described in the previous section. Each sort would then be performed on a copy of the data sets in order to avoid variation in random data affecting the runtime of the sort. The clocked time was measured in nanoseconds and saved to a file in both nanoseconds and milliseconds for ease of use and understanding. Setup done before each sort, such as function calls and copying the generated array, were not included in the clocked time. We used the same computer to run all of the tests, for CPU clock speed consistency, while it was not being used for other purposes, to avoid other functions taking computational time from the sorts, and assumed that background functions of the operating system would not significantly affect the results collected.

**Results**

Overall, the insertion sort performed the best out of the three with the selection sort only just beating it out in worst case scenarios. The data collected is displayed below and depicted in the following graphs.



\*\*Each entry in the above table is the average of three trials

When the data is in near sorted, or best case, order the bubble and selection sorts perform near identically with minor advantage toward selection sorts. Insertion sorts, on the other hand, holds a clear advantage, taking less than 1/100th the time of the other two sorts with one million data points in this case.

In the case of random data, insertion and selection sort are comparable in execution time. This is due to the selection sort operating the same regardless of starting data set while the insertion sort increases to O( from O(). The bubble sort has diverged from the other two, sitting at 100 times the execution time of the insertion sort.

In the case of reverse sorted data, the three sorts perform near equally bad.

**Conclusion**