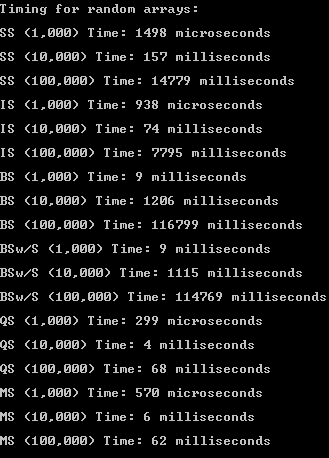
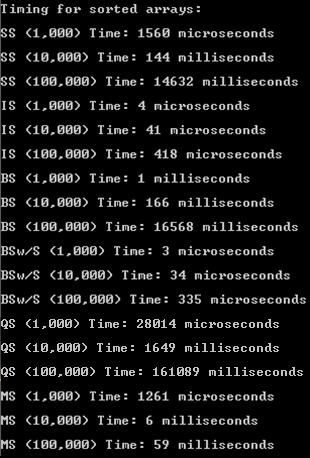
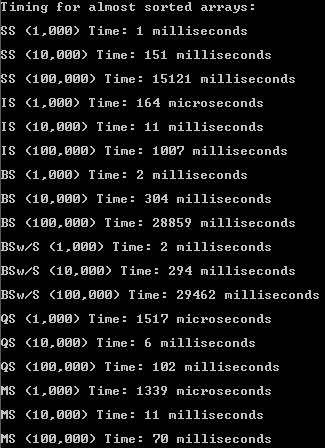
Project 2 – Analysis of Sorting Methods

By: Trevor Parsons

Experimental Results

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Random  1K | Random  10k | Random 100k | Sorted  1k | Sorted  10k | Sorted  100k | Almost Sorted  1k | Almost  Sorted  10k | Almost Sorted  100k |
| Selection | 1.5 ms | 157 ms | 14.8 sec | 1.6 ms | 144 ms | 14.6 sec | 1 ms | 151 ms | 15.1 sec |
| Insertion | 0.9 ms | 74 ms | 7.8 sec | 0.004 ms | 0.041 ms | 0.4 ms | 0.16 ms | 11 ms | 1 sec |
| Bubble | 9 ms | 1.2 sec | 116.8 sec | 1 ms | 166 ms | 16.6 ms | 2 ms | 304 ms | 28.9 sec |
| Bubble w/ Swap | 9 ms | 1.1 sec | 114.8 sec | 0.003 ms | 0.034 ms | 0.35 ms | 2 ms | 294 ms | 29.4 sec |
| Quick | 0.3 ms | 4 ms | 68 ms | 28 ms | 1.6 sec | 161 sec | 1.5 ms | 6 ms | 102 ms |
| Merge | 0.6 ms | 6 ms | 62 ms | 1.3 ms | 6 ms | 59 ms | 1.3 ms | 11 ms | 70 ms |





NOTE: The C++ code for this program did not copy correctly into Word, so it was included in a .txt file with the assignment

Selection Sort

The selection sort algorithm is bounded by two for loops that both run n times, which results in a time complexity of θ(n^2). Regardless of the order of the input before sorting, the algorithm will grow at a rate of θ(n^2), making n^2 the time complexity in best, average, and worst-case scenarios for selection sort. This means that the time required to complete the algorithm is dependent only on the size of the input.

Conclusion: This information is reflected in the experimental results. The time selection sort required to sort random 1k, sorted 1k, and almost sorted 1k arrays was roughly the same. Similarly, the sorting times for the 10k size arrays were about the same, as was the sorting times for the 100k sized arrays. This reinforces the concept that the time required to complete selection sort is dependent only on the size of the input array.

Additionally, the time used to sort rand 1k, rand 10k, and rand 100k increased by a factor of roughly 100, which is 10^2, and is consistent with the time complexity of n^2, as the arrays increased in size by a factor of 10. The same is true for the sorted and almost sorted arrays.

Insertion Sort

Like selection sort, the insertion sort algorithm is bounded by two for loops that will run n times during average and worst-case input scenarios, giving a time complexity of θ(n^2) and O(n^2), respectively. However, in the best-case scenario (when the input array is already sorted), the second for loop will never have to be run, resulting in a best-case time complexity of Ω(n), or linear growth. This is because the insertion sort speed increases when data is already sorted or somewhat sorted, since the insertion loop portion of the algorithm is required to run fewer times when the input data is sorted or somewhat sorted.

Conclusion: This information is reflected in the experimental results. The time insertion sort required to sort the rand 1k, rand 10k, and rand 100k was higher than the time required for to sort the corresponding almost sorted arrays, and much higher to sort the already sorted arrays of corresponding size. This is consistent with the theory, as the more sorted the contents of an input array was, the less time it required for insertion sort to complete.

Additionally, knowing that the size of the input arrays increased by a factor of 10, the time required to complete insertion sort for the sorted arrays increased by a factor of 10 (linear), while the almost sorted arrays sort time increased by a factor greater between 10 and 10^2, and the random arrays sort time increased by a factor of 100 (10^2). This is all consistent with the expected results of the tests.

Bubble Sort

The bubble sort algorithm is bounded by two for loops that both run n times, giving a time of complexity of worst, average, and best-case scenarios of θ(n^2). The order of the input is irrelevant for bubble sort, and the time required to complete the algorithm is only dependent on the size of the input array, as the algorithm will run the same number of comparisons regardless of input order.

Conclusion: The experimental results supported this, as the time required to sort each type of array (respectively, as input size grew) increased by a factor of 100, which is 10^2, and consistent with the expected time complexity of n^2.

Bubble Sort with Swaps

The bubble sort (with counting swaps) algorithm has the same time complexity in average and worst-case scenarios as bubble sort (for the same reasons), but when the input data is already sorted, it has a time complexity of Ω(n), or linear growth. This is because the algorithm checks if any swaps have been made during each pass of the loops. If no swaps have been made, then the array is assumed to be sorted and the process terminates. This means that an array that is already sorted will only require one pass through the array, resulting in a linear time complexity.

Conclusion: The experimental data supports the expected time complexity, as the time required to sort the random and almost sorted arrays grew by a factor of n^2, while the time to sort the already sorted array grew at a rate of n.

Quick Sort

The quick sort algorithm is bounded by the efficiency of the pivot selection in the partitioning function. Under best and average case input scenarios, a pivot is selected that allows the partition function to divide sub arrays in to equal or almost equal size, resulting in a time complexity of Ω(nlgn) for the best case and θ(nlgn) for the average case. However, when the partition function selects a pivot that divides sub arrays into size 1 and size n – 1 during each partition, the function has a time complexity of O(n^2).

Conclusion: The experimental results somewhat supported the theory of quick sort. For the random value and almost sorted arrays, the time quick sort required did not increase at exactly a rate of nlgn. In fact, it was significantly better, except for. The sorting time for the almost sorted arrays also increased at a rate slower than nlgn. However, the sorting time of the sorted arrays did grow at roughly an n^2 rate, which is consistent with the theory (the implemented quick sort algorithm used the last element of an input array as a pivot, so a sorted array input would have constituted a worst-case scenario).

Merge Sort

The merge sort algorithm is very similar to quick sort for time complexity, except in the worst-case scenario. The merge sort has a worst, average, and best-case time complexity of θ(nlgn). This is similar to the time complexity of quick sort, but merge sort always divides an input array into two roughly equal arrays, so it does not suffer from poor pivot selection in the worst case as quick sort does, allowing it to have a time complexity of O(nlgn) in the worst case.

Conclusion: The experimental results somewhat supported the theoretical expectations. The merge sort growth rates were actually slower than nlgn for all types of arrays. However, the time growth rate from smaller to larger input arrays was roughly constant among the different types of arrays, confirming that merge sort has the same rate of growth regardless of the order of the input.

General Conclusions

Overall, quick sort and merge sort preformed the best for sorting random arrays and almost sorted arrays, while bubble sort counting swaps and insertion sort preformed best for sorted arrays. Bubble sort and Bubble sort counting swaps were by far the worst algorithms for sorting random and almost sorted arrays.

Using a different implementation of quick sort that used a median-of-three method of selecting the pivot value would have drastically improved its performance on sorted arrays.

Given a general case where the contents of an array are unknown, the data and theory would suggest using merge sort or an implementation of quick sort that selected a more efficient pivot value.