Corey Lang

Project 1 – Experimental Analysis of Sorting Algorithms

table of time and #comparisons Bubble Sort

Algorithm	Timer	Num of Comparisons
Bubble Sort_Sorted_1000	1000800	999
Bubble Sort_Reverse_1000	209190400	499500
Bubble Sort_Rand_1000	153139400	497497
Bubble Sort_Sorted_2000	1000900	1999
Bubble Sort_Reverse_2000	825880600	1999000
Bubble Sort_Rand_2000	635898200	1997889
Bubble Sort_Sorted_4000	1000800	3999
Bubble Sort_Reverse_4000	3327240000	7998000
Bubble Sort_Rand_4000	2444684000	7989376
Bubble Sort_Sorted_8000	1000800	7999
Bubble Sort_Reverse_8000	13590166700	31996000
Bubble Sort_Rand_8000	10461719200	31959255

table of time and #comparisons Insertion Sort

Algorithm	Timer	Num of Comparisons
Insertion Sort_Sorted_1000	995500	0
Insertion Sort_Reverse_1000	146133500	500499
Insertion Sort_Rand_1000	77071100	248765
Insertion Sort_Sorted_2000	1000300	0
Insertion Sort_Reverse_2000	585537500	2000999
Insertion Sort_Rand_2000	299758200	1005734
Insertion Sort_Sorted_4000	2001500	0
Insertion Sort_Reverse_4000	2402528100	8001999
Insertion Sort_Rand_4000	1175890200	3970233
Insertion Sort_Sorted_8000	4003700	0
Insertion Sort_Reverse_8000	9731611500	32003999
Insertion Sort_Rand_8000	5033822000	15991004

table of time and #comparisons Quick Sort

Algorithm	Timer	Num of Comparisons
Quick Sort_Sorted_1000	4002800	10511
Quick Sort_Reverse_1000	5004600	10682
Quick Sort_Rand_1000	5004800	11825
Quick Sort_Sorted_2000	10009500	26071
Quick Sort_Reverse_2000	9007300	24946
Quick Sort_Rand_2000	10009400	23630
Quick Sort_Sorted_4000	20018300	56687
Quick Sort_Reverse_4000	20018300	55019
Quick Sort_Rand_4000	19017200	52262
Quick Sort_Sorted_8000	46041100	121814
Quick Sort_Reverse_8000	41037000	119070
Quick Sort_Rand_8000	43039900	116002

table of time and #comparisons Heap Sort

Algorithm	Timer	Num of Comparisons
Heap Sort_Sorted_1000	11010100	10208
Heap Sort_Reverse_1000	11009400	8816
Heap Sort_Rand_1000	12011600	9574
HeapSort_Sorted_2000	26023500	22300
HeapSort_Reverse_2000	25023800	19708
HeapSort_Rand_2000	27023800	21122
HeapSort_Sorted_4000	59053300	49142
HeapSort_Reverse_4000	55050500	43436
HeapSort_Rand_4000	65059300	46323
HeapSort_Sorted_8000	145131900	106638
HeapSort_Reverse_8000	112101600	94776
HeapSort_Rand_8000	145132700	100640

table of time and #comparisons Quick Sort

Algorithm	Timer	Num of Comparisons
Merge Sort_Sorted_1000	6005500	6542
Merge Sort_Reverse_1000	6005000	6542
Merge Sort_Rand_1000	6006100	6542
Merge Sort_Sorted_2000	15013700	14000
Merge Sort_Reverse_2000	12011100	14000
Merge Sort_Rand_2000	14012800	14000
Merge Sort_Sorted_4000	28025600	31626
Merge Sort_Reverse_4000	24021200	31626
Merge Sort_Rand_4000	30027400	31626
Merge Sort_Sorted_8000	47042300	66878
Merge Sort_Reverse_8000	51046900	66878
Merge Sort_Rand_8000	59053400	66878

The timer column is in units of nanoseconds.

Each graph represents a different sorting algorithms with the various arrays of sizes and order.

Corey Lang
Project 1 – Experimental Analysis of Sorting Algorithms

The theoretical performance of each of the sorting algorithms can be shown as an average in terms of their theta. The theta unit is described to be a some value that will appear above and below the calculated computation time when multiplied by some constant. The bubble sort algorithm has an average computation time of $\Theta(n^2)$. This exponential computation time means that the time taken to compute the sort will grow exponentially when the number of elements in the array increases. This is not very optimal when dealing with large datasets. The sorting algorithm that shares this average computation time of $\Theta(n^2)$, in this project, is the insertion sort algorithms. The best case for these two sorting algorithms is $\Theta(n)$ meaning the best case is linear. The worst case for these sorting algorithms is $\Theta(n^2)$ the same as the average case. The heap sort algorithm has an average computation time of $\Theta(n \log n)$ meaning that compared to the insertion sort and bubble sort this sorting algorithm will be more optimal for larger datasets. The worst case for this sorting algorithm is the same as the average case. The sorting algorithm that shares this computation run time is the merge sorting algorithm. The last sorting algorithm mentioned in this project is the quick sorting algorithm. This algorithm has an average computational time of $\Theta(n \log n)$ similar to the other recursive sorting algorithms, heap and merge, but the worse case run time is $\Theta(n^2)$ much like the worst case found in bubble sort and insertion sort.

The practical performance results seem to align with our theoretical understanding of the algorithms, with more efficient algorithms like merge sort and heap sort outperforming less efficient algorithms like bubble sort and insertion sort, and the input order affecting the performance of different algorithms differently. This could be found in the results as bubble sort and insertion sort have poor performance on average, taking a long time to sort even small input sizes. Quick sort, heap sort, and merge sort have much better performance on average, with heap sort and merge sort showing consistent performance across input sizes and orders. However, I was surprised initially by my data when the insertion sort algorithm on a sorted array made zero comparisons but since all the elements were already in the correct key, my code never made an increment to the comparison variable.

Upon examining the data, the time taken by each algorithm differs for arrays of varying sizes and orders. Generally, as the size of the array increases, so does the time taken by each algorithm, which is expected due to the increased number of comparisons and swaps required to sort a larger array. The time taken by each algorithm depends on the order of the array, with a sorted array taking the least time to sort and a reverse sorted array taking the most time. Bubble sort is the slowest algorithm, only suitable for small arrays or mostly sorted lists, while insertion sort is faster than bubble sort but still slower than other algorithms, performing well on small and mostly sorted arrays. Quick sort is one of the fastest algorithms, performing well on arrays of all sizes and orders, and its performance is not affected by the order of the array. Heap sort is faster than bubble and insertion sorts but slower than quick and merge sorts, performing well on arrays of all sizes and orders, but its performance is affected by the order of the array. Merge sort is the second fastest algorithm after quick sort, making the fewest comparisons when sorting an array, and its performance is not affected by the order of the array.