Wren Paris-Moe

9/19/2019

CS 317

Professor Jeffery Miller

Lab Project 1 – Sorting Algorithm Analysis

Possibly the most important procedure executed by computers is that of sorting. Hundreds of algorithms dedicated to sorting a list of items have been developed by computer scientists and mathematicians. A sorting algorithm is a process of commands that turns an unorganized array into an explicitly arranged list. This can mean alphabetical, increasing, or decreasing order etc. Analysis of sorting algorithms includes investigating the best case, average case, and worst-case scenarios of run time for the process to complete. These cases vary immensely depending on the arrangement or size of the data and efficiency of the algorithm. Some sorts are optimal for certain data sets, but ineffective for others. For the purposes of this laboratory, the average case run time complexities (Big-Theta) of Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort will be examined.

Below is pseudocode used in the main method used to measure the runtime of each algorithm. The sorting algorithms were run on a range of array lengths increasing by a scale of base ten to test the time complexities as *n*, the array size, reaches infinity. However, depending on the efficiency, different ranges of *n* were used to measure algorithms of distinct time complexities. For example, Merge Sort and Quick Sort could sort an array size of up to 10,000,000 elements while other sorting methods became wildly inefficient after a length of 100,000 elements. Bubble sort on an array of length *n* = 1,000,000 took three hours without completion. Each *n* value was timed over 10 trials to record an accurate average run time, T(*n*). The number of nanoseconds since the beginning of the day was recorded directly before and after the sorting algorithm was run. The difference between the two values was calculated to get the runtime of the algorithm.

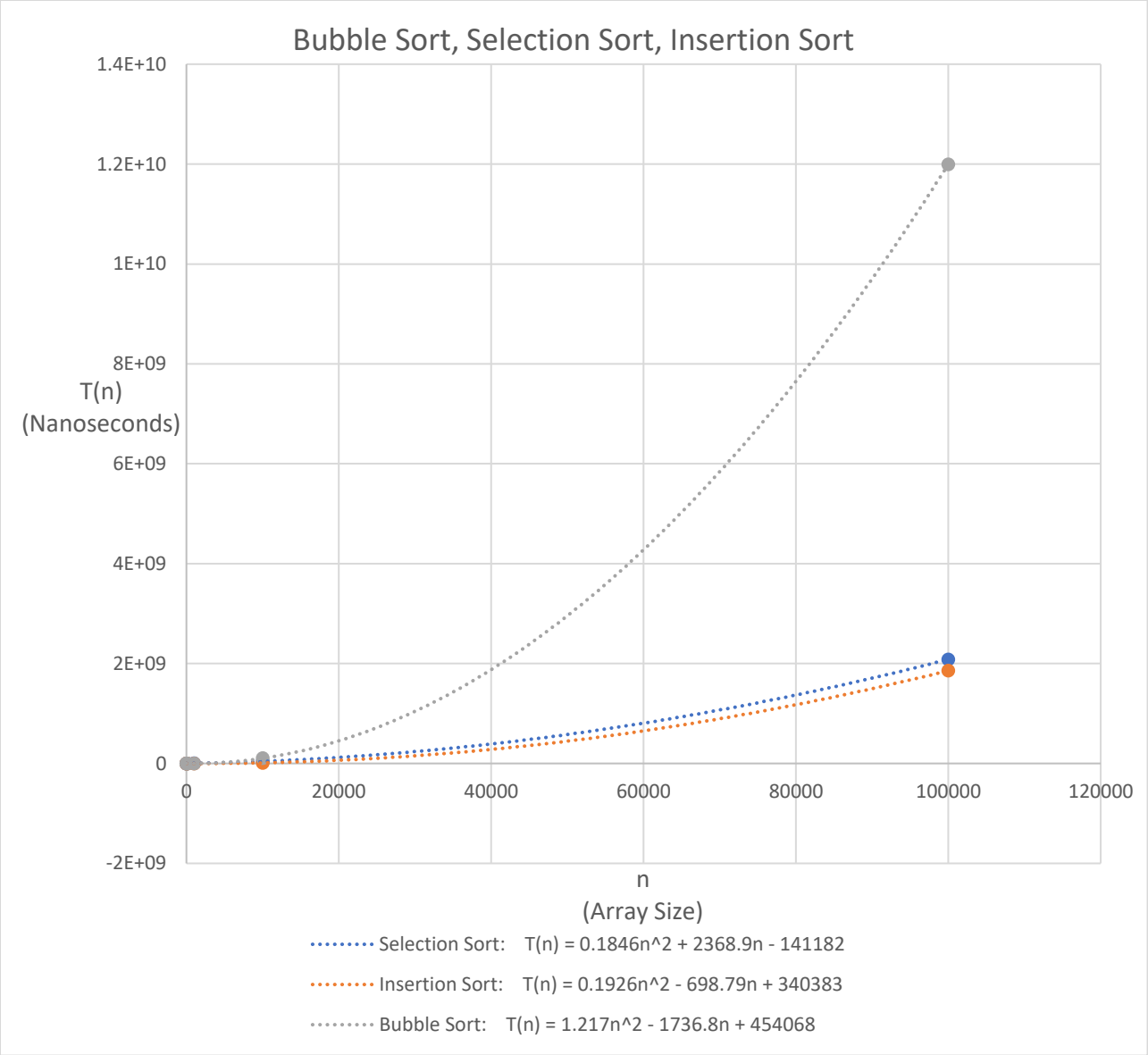
```
main(){
  for(n = 1; n < 10000000; n*10){
    long timeSum = 0;
    for(int i = 0; i < 10; i++) {
      int[] arr = randomArray(n);
      long start = System.nanoTime();
      sort(arr);
      long end = System.nanoTime();
      long elapsed = end – start;
      timeSum += elapsed;
    }
    long timeAvg = timeSum / 10;
    System.out.println("n=" + n + ": " + timeAvg);
  }
 }
```

Bubble Sort, Insertion Sort, & Selection Sort

Out of the five algorithms, Bubble Sort, Insertion Sort, and Selection Sort were more closely related than the others. As *n* got larger, the sorting runtime was much longer for these algorithms and especially for Bubble Sort. Bubble Sort was the simplest sorting algorithm examined. It works by repeatedly switching nearby elements when they are not in sorted order. Insertion sort was another simple algorithm that works similarly to sorting a deck of cards. Essentially, the array gets traversed until a value smaller than the previously navigated elements is found. This is sent to its appropriate place in the list. This process is repeated until the list is fully organized. Selection sort is similar to insertion sort except the minimum value is repeatedly sent to the beginning of the array. This creates two subarrays while the algorithm is running, one sorted and one unsorted. The algorithm repeats until the unsorted array no longer exists.

	Elapsed Time - T(n) (nanoseconds)		
n (Array Size)	Selection Sort	Insertion Sort	Bubble Sort
1	140	110	110
10	1430	1190	1440
100	98970	68160	180260
1000	2121560	814470	1054480
10000	42044190	12508090	104664730
100000	2083196090	1856541340	11996817280

Above is the data for the first three algorithms with *n*, the array size, in the leftmost column and T(*n*), the runtimes of each sort, in the right three columns. The runtime was measured in nanoseconds. By examining the data for these algorithms, it is easy to notice that the data increases faster than a linear trend since each succeeding T(*n*) value grows by more than a magnitude of ten. Bubble Sort was clearly the least efficient of the three algorithms. With an array length of *n* = 100000, the final runtime was over 12,000,000,000 ns while the runtime for Selection Sort and Insertion Sort was near 2,000,000,000 ns. This means that Bubble Sort can be up to six times as inefficient as other sorting algorithms when used on longer lists.



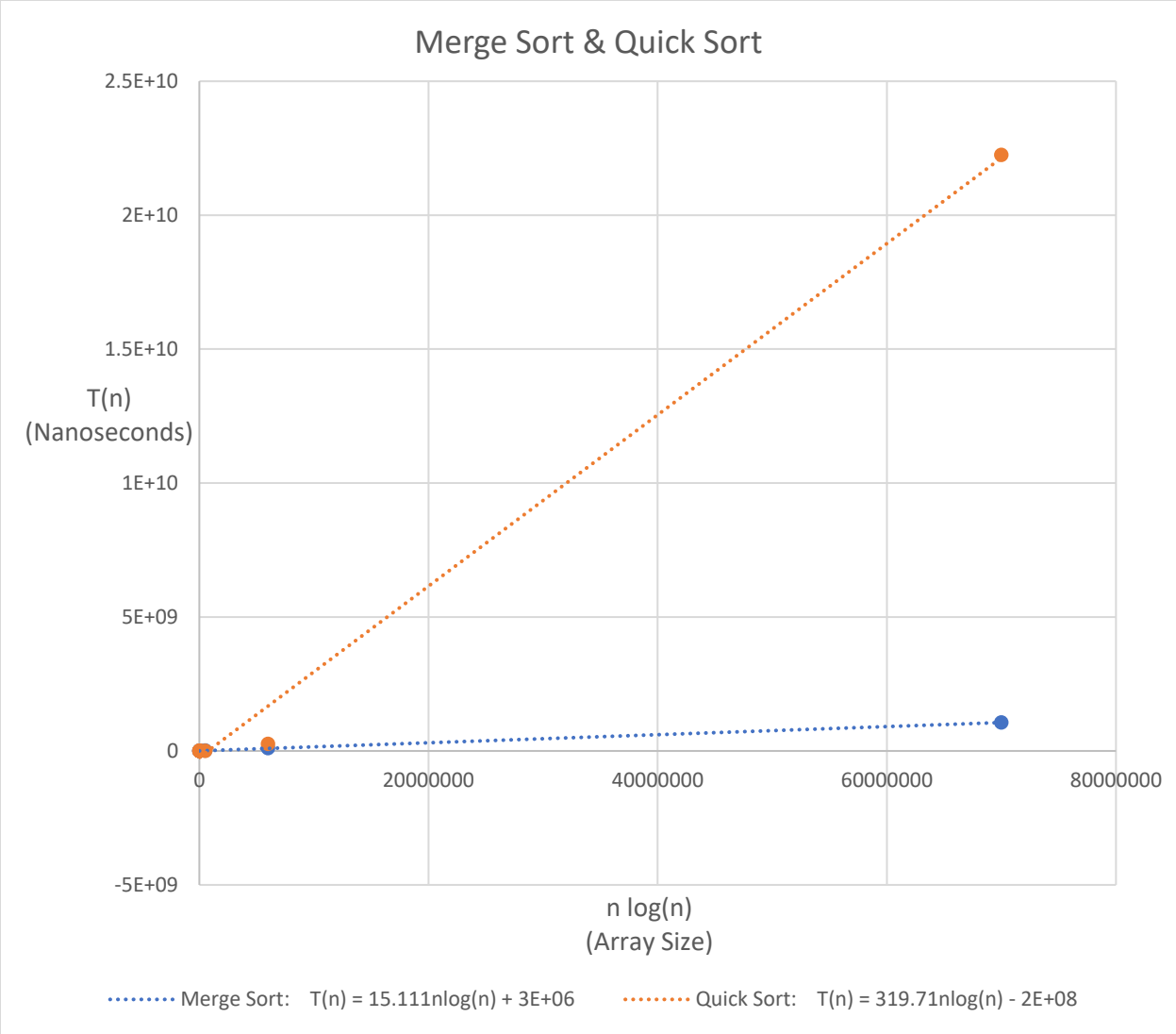
Using excel, the runtime data was recorded and graphed on a scatter plot with best fit trendlines to represent the tendencies of the data. The trendline for each sorting algorithm was labeled in the legend at the bottom of the graph. Selection Sort was colored blue, Insertion Sort was orange, and Bubble Sort was gray. The array size, n, was plotted on the x-axis while the runtime, $T(n)$, was plotted on the y-axis. A logarithmic scale was not used for the x-axis, so four out of the six data points are difficult to distinguish. However, without having the values evenly spaced, the trendline is easier to decipher and understand. Excel was used to create best fit trendlines for each of the sets of data. It was found that a quadratic line best fits the runtime data for every sorting algorithm on the graph, though each with differing coefficients. The quadratic functions for Selection Sort, Insertion Sort, and Bubble Sort all have an R squared value of $R^2 = 1$ meaning the trendlines were extremely accurate. By observing the graph, it becomes apparent how much worse of an algorithm Bubble Sort was than the rest. Bubble Sort dwarfed the other functions on the scatter plot. Additionally, the coefficients of the quadratic equation for Bubble Sort were much larger than the coefficients for Selection Sort and Insertion Sort.

Merge Sort & Quick Sort

Merge Sort and Quick Sort were more closely associated than the other three algorithms studied in this laboratory. As *n* got larger, T(*n*) increased at a much slower rate than the other algorithms. This required a separate chart and graph for the data of Merge Sort and Quick Sort. Both Merge Sort and Quick Sort involve a more complex sorting procedure than the three previously examined algorithms. Merge Sort and Quick Sort are examples of a "divide and conquer" algorithm. This means they are recursive algorithms. Merge Sort works by dividing an input array in half, calling itself on the two halves, and doing this repeatedly until there are *n* arrays of length one. Then the arrays are merged back together while simultaneously being sorted until we have the full original array back together in sorted fashion. Quick Sort works similarly to Merge Sort. Quick Sort uses a pivot element to partition the input array around the chosen element. This is done recursively until all elements have been placed correctly in the sorted array. Quick Sort can be implemented in many ways depending on how the pivot element is chosen. For this laboratory, the last element was always chosen as the pivot. Other indices can be chosen for the pivot like the first element, a random element, or the median element. This can lead to ambiguity in the efficiency of Quick Sort depending on how the data is arranged.

			Merge Sort	Quick Sort
n (Array Size – number of element)	log(*n*)	*n*log(*n*)	T(*n*) (Run Time – nanoseconds)	
1	0	0	100	80
10	1	10	4510	1610
100	2	200	25490	20260
1000	3	3000	162010	66520
10000	4	40000	1458510	551120
100000	5	500000	13211910	7334970
1000000	6	6000000	103924500	264457040
10000000	7	70000000	1059567590	22257089490

Above is the data for Merge Sort and Quick Sort with *n*, the array size, in the leftmost column and T(*n*), the runtimes of each sort, in the two rightmost columns. *n*log*(n)* was calculated for the purposes of plotting the data with an accurate best fit line. Again, the runtime for each algorithm was measured in nanoseconds. In being recursive algorithms, Merge Sort and Quick Sort were able to use a much higher range of *n* since they are more efficient. The computer maxed out at a length of *n* = 100,000 for the other three algorithms while Merge Sort and Quick Sort could sort up to 10,000,000 elements. The rate of increasing runtime for these two sorts was closer to linear growth than quadratic. The data also illustrated that Merge Sort was the most efficient algorithm out of the five sorts studied in this laboratory. The final runtime for Quick Sort took a little over 22,000,000,000 ns while Merge Sort ran for around 1,000,000,000 ns. That was almost 22 times as large of a value. This further demonstrates how Merge Sort was the best sorting algorithm for arrays of large sizes.



Excel was used to record and graph the data for Merge Sort and Quick Sort on a Scatter Plot with best fit lines to represent the trends of the algorithms. Each trendline was labeled at the bottom of the graph in the legend. Merge sort was colored blue while Quick Sort was orange. The runtime, T(n), in nanoseconds was plotted on the y-axis while the $n$log(n) version of the array size is on the x-axis. The x-axis values increased by base ten causing four out of the six points to be difficult to distinguish. However, this allows the trendlines on the graph to be legible. Excel was used to find the best fit lines for the data of Merge Sort and Quick Sort. Excel is only able to form an exponential, linear, logarithmic, polynomial, power, or moving average trendline. However, an $n$log(n) equation is the best fit function for each data set. Therefore, $n$log(n) was used on the x-axis, instead of n, with a linear trendline to result in a final $n$log(n) equation. Merge Sort had an R^2 value of 0.9998 and Quick Sort had an R^2 value of 0.9947 meaning that the approximations were extremely accurate. By examining the scatter plot, it was easy to decipher how much more efficient Merge Sort is than Quick Sort. The last value of Quick Sort dwarfed that of Merge Sort making it seem close to the x-axis when really it had a value of over 1,000,000,000 ns. The coefficient of Quick Sort's approximation was more than 21 times that of Merge Sort's further illustrating the disparity in efficiency between sorting algorithms.

Discussion

The main take away from this lab were the trends that the approximations made on the runtime data of each sorting algorithm. By using the average runtime values as n moves towards infinity, a trendline is able to explain the tendencies of any algorithm. These lines represent the Big-Theta notation of the algorithm. An important detail that confirms the validity of the data is that the approximations found in this lab match the already known average cases of each algorithm. Selection Sort, Insertion Sort, and Bubble Sort are all quadratic while Merge Sort and Quick Sort are $n\log(n)$. Additionally, there was a correlation between the simplicity of an algorithm and its average case runtime. Selection Sort, Insertion Sort, and Bubble Sort were the simplest of the five algorithms and all have slower runtimes. Bubble Sort was the simplest sorting algorithm having by far the worst runtimes. Merge Sort and Quick Sort are more complex recursive algorithms making them much more efficient. Clearly, simplicity comes at a cost when relating to sorting algorithms.

The intersections between lines show when one algorithm becomes faster than another. The function for Insertion Sort intersected with Selection Sort at $n = 203$ meaning that was the number of elements where Insertion Sort became faster than selection sort. Insertion Sort intersected with Bubble Sort at $n = 67$ meaning that it was almost immediately more efficient. Merge Sort intersected with Quick Sort at $n = 130295$. This demonstrated from that point on, Merge Sort was more efficient. It also suggested that before that point, Quick Sort was more efficient with that range of array sizes. Merge Sort never intersected with Insertion Sort, the fastest of the quadratic sorting algorithms, meaning it was more efficient than Selection Sort and Bubble Sort for all ranges of n. However, this is known to not always be true. This is an example of how a trendline does not always exactly match the data at small ranges and is meant for looking at n as it moves towards infinity.

Merge Sort and Quick Sort were much better for large data sets due to their “divide and conquer” procedure. They were able to partition the data into smaller more convenient sections, sort those, and then merge them back into the original array. Merge Sort can work well on any data set. However, due to Quick Sort’s pivot element, the algorithm can be unstable. Since the pivot element is not always chosen correctly, the worst-case scenario (Big-O) is quadratic instead of $n\log(n)$. Insertion Sort was the best algorithm for small data sets. This is because a simpler sorting algorithm is wanted for smaller arrays. It was the fastest of the quadratic algorithms. A recursive algorithm is a poor choice for a small data set because each partition requires adding two extra entries to the stack frame. This was shown by the data where Insertion sort has the fastest runtime for an array of length $n = 10$.