Michael Thoreson

Adam Neely

Richard Bennett

Week 5 Lab Report

After reviewing the animations for each sorting algorithm, we found that they vary significantly in performance across use cases. For the following table, each row represents the use case, and each table represents the sorting algorithm used. Each cell will represent what order the sort completed compared to the other two algorithms for that use case. Parens around each value will indicate a significant lead time between the algorithms completing, and brackets around sets of values will indicate an insignificant difference in completion time. These results were found with a problem size of 20.

	Insertion	Selection	Bubble
Random	1st	3rd	2nd
Nearly Sorted	1st	(3rd)	2nd
Reversed	[1st]	3rd	[2nd]
Few Unique	1st	3rd	2nd

From this table, it can be demonstrated that selection sort was the most time-intensive algorithm out of the three. After running a test with n=50, we found similar results.

We then compared tested each algorithm against itself on different use cases, where n=50:

	Insertion	Selection	Bubble
Random	3rd	1(tie)	2nd(tie)
Nearly Sorted	1st	1(tie)	1st
Reversed	4th	1(tie)	2nd(tie)
Few Unique	2nd	1(tie)	3rd

While selection and bubble sort both excel at nearly sorted lists, selection sort has much more consistent results. However, at n=20 and n=50, Selection sort underperforms in every case.

After running the algorithms in slowSortCompare.cpp, where n = 10,000, we found the following result. The number in each column represents how many moves each algorithm made in the course of sorting the array:

	Insertion	Selection	Bubble
Random	76006248	29961	76006248
Already Sorted	0	0	0
Nearly Sorted	12942	2706	12942
Reversed	149985000	1500	149985000
Few Unique	75330000	29100	75330000

And this table represents the number of comparisons:

	Insertion	Selection	Bubble
Random	50690822	50093272	75335366
Already Sorted	19998	50005000	10000
Nearly Sorted	28626	50007505	114259
Reversed	99999999	75010000	100000000
Few Unique	5023992	50048800	73990750

We decided that a good general metric of performance would be to combine moves (swaps x 3) and comparisons:

	Insertion	Selection	Bubble
Random	126,697,070	50,123,233	151,341,614
Already Sorted	19998	50005000	10000
Nearly Sorted	41,568	50,010,211	127,201
Reversed	249,984,999	75,011,500	249,985,000
Few Unique	80,353,992	50,077,900	73,990,750

For n=10,000, and using the algorithms contained in slowSortCompare.cpp, we found that Random sets were done the fastest by Selection, with Bubble and Insertion significantly running behind that. While they each did a similar number of comparisons (especially selection and insertion), Selection required significantly fewer moves.

Conversely, already sorted arrays were processed the fastest by Bubble sort, and the slowest by far by selection. This is likely due to the fact that selection sort requires the same number of comparisons on each pass of the array, and bubble sort only compares adjacent values.

Insertion sort outperformed the other two methods for Nearly sorted arrays, because it requires less comparisons, and nearly sorted arrays require less moves. Selection sort did worst here, but only marginally worse than a fully sorted array.

Reversed arrays are most effectively sorted using Selection sort, as it requires more moves than any other array structure, and Bubble and Insertion sort are more move-intensive. Selection sort does noticeably more comparisons on a reversed array than a sorted or nearly-sorted array, but far fewer moves than the other algorithms.

Finally, selection sort performed the best out of the three for arrays containing many duplicates. This is likely because Insertion sort has to do many comparison operations between identical values, while selection sort does not have to compare these identical values once they've been sorted. Bubble sort also underperforms because it has to make many more comparisons and moves than the others.

These results differ from what is demonstrated by the animations on toptal.com. One possible reason for this is a difference in implementation – some implementations will handle sorting differently from others, with varying performance results. Another likely reason for this discrepancy is the difference in size of n. In their examples, n <=50, whereas in our implementation the arrays are 10,000 indices long. Selection sort had the most consistent and scalable performance out of the three algorithms, likely because it makes a consistent number of comparisons and only moves after comparing the entire unsorted segment of the array.