CPSC 319 – Assignment #1

Ethan Reed, Section T07

February 1st, 2022

Method:

For this experiment, a java class was created with methods required for quick, merge, selection, and insertion sorts. Each sorting method was provided in the lecture notes, apart from the merge() method, which was obtained from geeksforgeeks.org[1]. Each method was then called on an array and timed. Times were collected with every combination of array length, initial array sorting, and sorting algorithm shown in Table 1 below. The integers in the array were randomly generated with values between 0 and the length of the array, then the array was either left random, sorted into ascending order, or sorted into descending order. The sorted arrays were outputted into a text file to verify the final sorting, and the sort times were printed to the terminal in nanoseconds.

Data:

Pre-sort	Algorithm	10	100	1 000	10 000	100 000	1 000 000
Ascending	quick	7900	23700	293100	1166700	22395800	54192300
	insertion	2700	4800	26900	273300	2071000	5226500
	merge	9300	52200	596700	1864900	13738600	84531800
	selection	5100	88100	2639900	16654400	827140400	88557743200
Descending	quick	6400	24500	372200	1788300	24956400	57089500
	insertion	3200	100800	2900600	20600700	1280682900	132991698000
	merge	7500	49600	715600	1694000	13989000	69238800
	selection	4500	94400	2883500	100664200	9914440900	976594133700
Random	quick	7200	30000	481300	1811600	37279000	127050100
	insertion	3300	52500	2244500	16780800	646581700	64915774700
	merge	9500	57700	714200	2532100	18531400	155073500
	selection	6000	92200	2570100	59161700	4969366100	495580328800

Table 1, sort time for every combination of pre-sort, array size, and sorting algorithm. times measured in ns.

Analysis:

quick: for the most part, quick sort had the fasted times when the length of the array was 100 - 1000, and when the length of the array was 1000000 or greater. Its best times were when it was given an array in ascending order, but it performed about as well when it was given an array in descending order.

insertion: insertion sort was best with small arrays, always being the fastest when the length of the array was 10, but it quickly lost its lead. This algorithm performed quite poorly when the length of the array was large, with one exception: when given an array already sorted into ascending order, it was the fastest for every array length.

merge: merge sort performed about as well as quick sort, performing even better when the length of the array was 10 000 or 100 000. merge sort typically did its best when it was given an array in descending order.

selection: Although selection sort fared well when the arrays were of length 10, it was consistently the worst sorting method at all larger lengths. It performed its best when given an

array in ascending order. Above the 10 000 mark for length, the method performed around five times worse when given a randomly sorted array, and about ten times worse when given an array pre-sorted in descending order.

Complexity Analysis:

quick:

Worst case: $O(n^2)$, this case is when the pivot is always chosen so that one subarray has 0 elements, and the other has all remaining elements. This maximizes the number of levels of recursion. This case is easily avoidable, as the pivot can be chosen at random. With a pivot chosen at random, the odds of the pivot always being the min value or max value are minimal.

Average case: O(n lg n), this case occurs when the split of the proportions is a normal distribution between the best and worst cases. The running time will be the best-case running time multiplied by some constant, because constants are absorbed by big O notation, the classification is the same as for the best case.

Best case: O(n lg n), this case occurs when the array is always partitioned down the middle. Each level of the recursive sort has twice the partitions of the level before it, so if the number of levels is x, $2^x = n$ and $x = \lg n$. multiplying the number of levels (lg n) by the time complexity of each level (O(n)) gives an overall time complexity of O(n lg n). If the pivot selection is randomized to avoid the worst case, the best case will also be rare.

insertion:

Worst case: $O(n^2)$, the outer loop of an insertion sort algorithm always must loop through n - 1 times to sort every value in the array (except for the first, which will eventually bubble to the correct position). In the worst case, where the algorithm is given an array in reverse order, the inner loop will also have to go through n - x iterations (x depending on the values original position in the array), as it must shift almost every element of the array over one at a time to insert the value currently being sorted. Both the inner and outer loops have an O(n) classification, so according to the product rule, the overall classification is $O(n^2)$.

Average case: $O(n^2)$, in the average case, where the array starts in random order, the outer loop will still go through n - 1 iterations. The inner loop will have to shift half as many values as the worst case on average, but it will still be some value n - x. Although the average case should likely be faster than the worst case, both loops still have the same O(n) classification, and the overall algorithm will still have an $O(n^2)$ classification.

Best case: O(n), when given an array already sorted into ascending order, the outer loop of the algorithm will still go through n - 1 iterations. The inner loop on the other hand, will not have to shift any values, and will take some constant time c. The algorithm will in the best case have an O(n) classification, as the constant time spent in the inner loop is absorbed by the big O notation.

merge:

> **Worst case: O(n lg n)**, all cases of merge sort are conveniently nearly the same. All cases behave like the best case of quick sort, in the sense that the array is repetitively split down the middle into partitions. The partitions are then merged in a separate function with an O(n) classification. Because the partitioning has an O(lg n) classification just like in the best case of quick sort, and the merging has an O(n) classification, the overall classification for all cases is O(n lg n).
>
> **Average case: O(n lg n)**, see above.
>
> **Best case: O(n lg n)**, see above.

selection:

> **Worst case: O(n^2)**, selection sort works by taking each element, up to and including the second last one, and then swapping it with the least element above it in the array. Because the procedure does not change depending on the original sorting of the array, this method has an O(n^2) classification in all cases. The O(n^2) classification is because the algorithm runs n - x (x depending on the current position in the array) comparisons and a swap, for n - 1 elements in the array. Multiplying our two expressions leaves us with an O(n^2) expression.
>
> **Average case: O(n^2)**, see above.
>
> **Best case: O(n^2)**, see above.

Interpretation:

Selection sort and insertion sort's long times when given large amounts of data makes sense given the O(n^2) classification for most cases. Selection sort managed to perform about ten times worse with these larger arrays. Selection sort also performed better with pre-sorted arrays; this is likely because no swaps were occurring. Insertion sort had a O(n) classification in its best case, which explains why it performed so well when it was given an array already sorted into ascending order.

The behavior for quicksort and merge sort makes sense given the O(n log n) classifications for all cases. Quicksort also has a O(n^2) classification for its worst case, but this case did not obviously occur during testing thanks to the pivot being chosen at random.

Conclusion:

for large arrays, merge sort and quicksort are the best options. Be aware that quicksort can perform very badly if given an array sorted in just the wrong order, but this is unlikely. Insertion sort and selection sort are both simple and viable when sorting very small arrays, but selection sort is almost never the right pick, due to their being better in-place sorting options with both large and small arrays. Because merge sort relies on the creation of many temporary arrays, it should be avoided if storage is a concern.

Reference(s)

[1] GeeksforGeeks 2022, accessed 1st February 2022, <http://www.geeksforgeeks.org>