Tommy Hollenberg
06/12/2017
CS162 S17
Final Project

The function headers have been filled out and the questions in the comments have been answered in detail in the sorting.c file. Please look there for the "Function Header" and "Very Thorough Description" sections of the report. Although I did not include pictures, I did a very thorough job explaining each sort, including loop invariants and detailed comments for every line of code.

For each sorting algorithm, I tested four different pre-sorted orders (random, descending, ascending, and uniform), and four different array sizes (500 elements, 5,000 elements, 50,000 elements, and 500,000 elements). A table summarizing the testing scheme and the results is included at the bottom of this report.

Bubble Sort

Bubble sort contains two nested for-loops. The outer for loop executes n times. The inner for-loop executes n–i times in the i-th iteration of the outer loop. Therefore, the time complexity is:

$$\sum_{i=0}^{n-1}(n-i) = \sum_{i=1}^{n} i = \frac{n*(n+1)}{2} = O(n^2)$$

This is both the best-case and the worst-case time complexity in this implementation. Bubble sort can be implemented to quit if no swaps are performed in an entire execution of the inner loop, indicating that the array is already sorted. This improvement allows a best-case time complexity of $O(n)$ for a pre-sorted array.

However, this time complexity is based on the number of comparisons, not the number of swaps. In practice, even this implementation ran between 2-4x faster on pre-sorted arrays than on reverse-sorted arrays, which require the least and most number of swaps, respectively. The run time for all four pre-sorted orders increased roughly proportionally with the square of the array size.

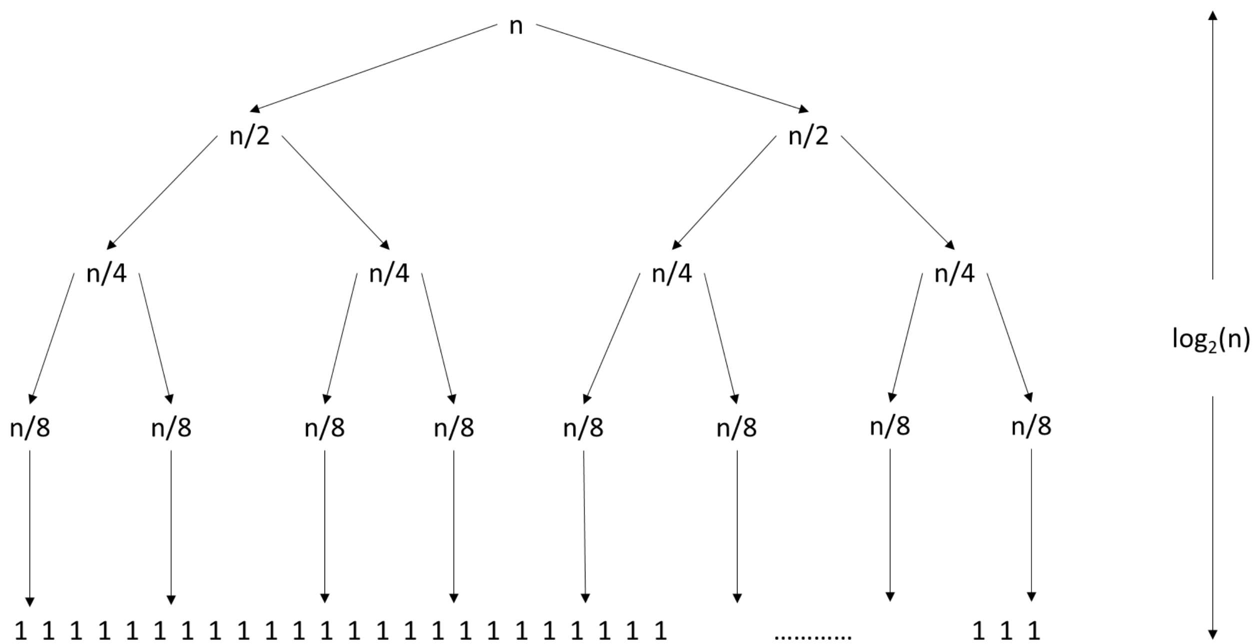
Insertion Sort

Insertion sort also contains two nested for-loops. As in Bubble sort, the outer for-loop executes n times. The inner for-loop executes at most i times in the i-th iteration of the outer loop. Therefore, the worst-case time complexity is also $O(n^2)$. However, each inner-loop is cut short when the insertion location of the element is found. This results in faster running times when the array is closer to being sorted already, with a best case running time of $O(n)$ for a pre-sorted array. This is perfectly represented in the test results, where the

running times of the random and descending test cases increase quadratically with the number of elements, while those of the ascending and uniform (already sorted) test cases increase linearly.

Merge Sort

Splitting the array in half takes constant time. However, the merge function is linear in the size of the merged array because each value must be written twice, once into the temp array and once back into the nums array (the fact that it is twice and not once is insignificant).



At each stage of recursion, the total number of elements across all merge calls is n, and there are $\log_2 n + 1$ stages. However, merge is not called in the final stage, when the subarray size is 1 and the base case is reached. It is called as a tail call in all other stages, and overall there are n elements being merged in each stage, so the time complexity of the Merge sort is $O(n \log n)$. The run time data follows this trend, but performs slightly better than expected at each interval.

In the merge function, if one of the two sorted subarrays being merged is exhausted, the remaining elements of the second subarray can be copied into the temp array without performing any more comparisons. However, the same number of writes and recursive calls are made regardless of the pre-sorted state. For this reason, the algorithm performs equally well on the pre-sorted (ascending, uniform) and reverse-sorted (descending) test cases. In the former, the entire lower_half subarray comes before any elements of the upper_half subarray, and the opposite is true in the latter. The randomized array is equally likely to take elements from each subarray, so up to twice

as many comparisons may be required for each merge function call. This is reflected in the run time data, where the random test cases take around 50% longer than the other three.

Selection Sort

Selection sort, like Bubble sort and Insertion sort, contains two nest for-loops. Exactly like Bubble sort, the outer loop executes n times, and the inner loop executes n−i times in the i-th iteration of the outer loop. The same number of comparisons are made regardless of the pre-sorted order of the elements, so both the best-case and worst-case time complexities are $O(n^2)$. However, the number of swaps required in selection sort is $O(n)$. Since this is insignificant compared to the number of comparisons, the run times were consistent across all four pre-sorted order categories of test cases, which all increased quadratically with the number of elements. Selection sort was about twice as fast as Bubble sort for arrays with a lot of inversions (requiring considerably more swaps for Bubble sort), but roughly the same for pre-sorted arrays.

Quicksort

Quicksort is a recursive algorithm. Each recursive call consists of a linear time partition function call that organizes the elements into two sections of values less than or equal to the pivot value (which is always chosen as the value of the highest index) and values greater than the pivot value, respectively. Then, quicksort is called recursively for both sections. If the array is divided roughly evenly each time, this is very similar to Merge sort (with two recursive calls each of size n/2 and a linear time merge function). Thus, the best-case (and average-case) time complexity is $O(n \log n)$. However, unlike Merge sort, the partitioning is highly depending on the pre-sorted array state, so even partitioning cannot be guaranteed. Frustratingly, the worst-case time complexity of $O(n^2)$ is attained when the array is already sorted (or reverse-sorted). In this case, since the rightmost value is the highest (or lowest), all n−1 elements will be in the less than or equal to (or greater than) category, and the recursive calls will be of size n−1 and 0. This leads to a recursive depth of n, with n−i elements in the i-th stage of recursion, which is equivalent to the equation shown above for the Bubble sort algorithm. Not only is does this result in much slower run times, but at large n, the recursive depth causes the stack to overflow, resulting in a segmentation fault.

Median-of-3 pivot selection:

Some of the worst-case behavior can be avoided by implementing a median-of-3 pivot selection. Rather than simply choosing the rightmost element as the pivot, the median value of the left, right, and middle ((left + right)/2) elements is chosen. This results in much more evenly divided subarrays, particularly for pre-sorted and reverse-sorted arrays. This effect is evident in the test results. Whereas the

ascending and descending testcases were considerably slower than the random testcases in the rightmost element pivot selection method, this discrepancy is eliminated with the median-of-3 pivot selection method, and even the random testcases even performed better with this method. However, this method had no effect on the uniform test cases, which were just as slow as in the rightmost element pivot selection, and still caused a segmentation fault at large n. This is because all elements are the same, so it does not matter which element is chosen as the pivot. All n-1 other elements will be partitioned into the less than or equal to category, with none in the greater than category, resulting in quadratic time complexity.

Tripartition:

This problem can be fixed by changing the partition function, so that the elements are instead grouped into one of three groups: less than the pivot value, greater than the pivot value, or equal to the pivot value. This technique is sometimes referred to as a "fat pivot," because rather than simply being one element, all occurrences of the pivot value are grouped around the pivot value and removed from subsequent recursive calls. This transforms the worst-case uniform array into the best-case. All elements are partitioned into the "equal to" category in linear time, leaving no elements for further recursive call, and resulting in a time complexity of $O(n)$. The success of this strategy is immediately apparent from the test results, where rather than causing a segmentation fault, the largest uniform test case sorted in just 2.2 ms.

Both:

Combining these two optimizations eliminates all segmentation faults and worst-case behavior. However, the tripartition function is slower than the regular partition function, despite having the same time complexity, because there is the possibility of two comparisons per loop iteration versus just one. This is reflected in the test results, where the "both" running times are slower than the "median-of-3" running times for all pre-sorted orders aside from uniform, where the tripartition algorithm shines.

Filename	N	Pre-sorted Order	Sorting Algorithm	Time Complexity	Run Time
500r	500	random	Bubble Sort	$O(n^2)$	1.5 ms
500d		descending			1.9 ms
500a		ascending			700 µs
500u		uniform			433 µs
5000r	5000	random	Bubble Sort	$O(n^2)$	110 ms
5000d		descending			190 ms
5000a		ascending			90 ms
5000u		uniform			43 ms
50000r	50000	random	Bubble Sort	$O(n^2)$	12.4 s
50000d		descending			10.8 s
50000a		ascending			6.8 s
50000u		uniform			4.4 s
500000r	500000	random	Bubble Sort	$O(n^2)$	1100 s
500000d		descending			1040 s
500000a		ascending			486 s
500000u		uniform			452 s
500r	500	random	Insertion Sort	$O(n^2)$	290 µs
500d		descending			570 µs
500a		ascending			4 µs
500u		uniform			4 µs
5000r	5000	random	Insertion Sort	$O(n^2)$	28 ms
5000d		descending			57 ms
5000a		ascending			35 µs
5000u		uniform			35 µs
50000r	50000	random	Insertion Sort	$O(n^2)$	2.85 s
50000d		descending			5.69 s
50000a		ascending			354 µs
50000u		uniform			354 µs
500000r	500000	random	Insertion Sort	$O(n^2)$	285 s
500000d		descending			568 s
500000a		ascending			3.5 ms
500000u		uniform			3.5 ms
500r	500	random	Merge Sort	$O(n \log n)$	80 µs
500d		descending			54 µs
500a		ascending			54 µs
500u		uniform			54 µs
5000r	5000	random	Merge Sort	$O(n \log n)$	1.0 ms
5000d		descending			670 µs
5000a		ascending			680 µs
5000u		uniform			724 µs
50000r	50000	random	Merge Sort	$O(n \log n)$	12.4 ms
50000d		descending			8.1 ms
50000a		ascending			8.2 ms
50000u		uniform			8.4 ms

500000r	500000	random	Merge Sort	$O(n \log n)$	145 ms
500000d		descending			95 ms
500000a		ascending			95 ms
500000u		uniform			96 ms
500r	500	random	Selection Sort	$O(n \log n)$	510 µs
500d		descending			510 µs
500a		ascending			480 µs
500u		uniform			492 µs
5000r	5000	random	Selection Sort	$O(n \log n)$	48 ms
5000d		descending			49 ms
5000a		ascending			47 ms
5000u		uniform			49 ms
50000r	50000	random	Selection Sort	$O(n \log n)$	4.7 s
50000d		descending			4.9 s
50000a		ascending			4.7 s
50000u		uniform			4.7 s
500000r	500000	random	Selection Sort	$O(n \log n)$	503 s
500000d		descending			510 s
500000a		ascending			522 s
500000u		uniform			472 s
500r	500	random	Quicksort (naïve)	$O(n \log n)$	83 µs
500d		descending			471 µs
500a		ascending			549 µs
500u		uniform			551 µs
5000r	5000	random	Quicksort (naïve)	$O(n \log n)$	959 µs
5000d		descending			44 ms
5000a		ascending			52 ms
5000u		uniform			52 ms
50000r	50000	random	Quicksort (naïve)	$O(n \log n)$	12.4 ms
50000d		descending			4.4 s
50000a		ascending			5.2 s
50000u		uniform			5.15 s
500000r	500000	random	Quicksort (naïve)	$O(n \log n)$	147 ms
500000d		descending			Segfault*
500000a		ascending			Segfault*
500000u		uniform			Segfault*
500r	500	random	Quicksort (median-of-3 pivot selection)	$O(n \log n)$	74 µs
500d		descending			76 µs
500a		ascending			32 µs
500u		uniform			580 µs
5000r	5000	random	Quicksort (median-of-3 pivot selection)	$O(n \log n)$	980 µs
5000d		descending			1.2 ms
5000a		ascending			390 µs
5000u		uniform			52 ms
50000r	50000	random	Quicksort	$O(n \log n)$	12.5 ms

50000d		descending	(median-of-3 pivot selection)		15.1 ms
50000a		ascending			4.6 ms
50000u		uniform			5.2 s
500000r	500000	random	Quicksort (median-of-3 pivot selection)	$O(n \log n)$	147 ms
500000d		descending			189 ms
500000a		ascending			48.6 ms
500000u		uniform			Segfault*
500r	500	random	Quicksort (tripartition)	$O(n \log n)$	94 µs
500d		descending			1.1 ms
500a		ascending			750 µs
500u		uniform			3 µs
5000r	5000	random	Quicksort (tripartition)	$O(n \log n)$	1.3 ms
5000d		descending			104 ms
5000a		ascending			73 ms
5000u		uniform			23 µs
50000r	50000	random	Quicksort (tripartition)	$O(n \log n)$	15.3 ms
50000d		descending			10.5 s
50000a		ascending			7.3 s
50000u		uniform			226 µs
500000r	500000	random	Quicksort (tripartition)	$O(n \log n)$	199 ms
500000d		descending			Segfault*
500000a		ascending			Segfault*
500000u		uniform			2.2 ms
500r	500	random	Quicksort (both)	$O(n \log n)$	82 µs
500d		descending			83 µs
500a		ascending			79 µs
500u		uniform			3 µs
5000r	5000	random	Quicksort (both)	$O(n \log n)$	1.3 ms
5000d		descending			1.8 ms
5000a		ascending			1.7 ms
5000u		uniform			23 µs
50000r	50000	random	Quicksort (both)	$O(n \log n)$	14.7 ms
50000d		descending			47.0 ms
50000a		ascending			46.7 ms
50000u		uniform			227 µs
500000r	500000	random	Quicksort (both)	$O(n \log n)$	185 ms
500000d		descending			1.40 s
500000a		ascending			1.39 s
500000u		uniform			2.25 ms

*Too many recursive calls cause stack overflow.