4190.407 Algorithms HW 2-2

Yeonjun Kim

2024-13755

Dept. of Computer Science and Engineering

2025-10-05

In this assignment, I implemented 4 different sorting algorithms in Python: Insertion Sort, Merge Sort, Quick Sort, and Dual-Pivot Quick Sort.

You can write the integer array in **`input.txt`** and run the program by **`$ python3 202413755_Task2.py`**. The output will be written in **`output.txt`**.

Theoretical Comparison

The four sorting algorithms have different theoretical time complexities, shown as below.

Table 1 time complexity comparison in big-O notation.

	Insertion Sort	**MergeSort**	**QuickSort**	**DualPivot QuickSort**
Best Case	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Average Case	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Worst Case	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n^2)$

While Insertion Sort has the fastest(linear) time complexity in best case, it becomes slower(quadratic) than other algorithms in average and worst cases. Merge Sort has a consistent time complexity of $O(n \log n)$ in every case. Quick Sort and DualPivotQuick Sort have time complexity of $O(n \log n)$, but it becomes quadratic in the worst case.

Experimental Setup

In my experiments, I generated testcases with different N ($\in \{10^3, 10^4, 10^5\}$). For each N, I generated four testcases: random, sorted, reversed, and nearly sorted.

- To generate 'random' cases, I sampled N integers in `range(1, 10000001)`.
- To generate 'sorted' cases, I used `sort()` in Python.
- To generate 'reversed' cases, I used `sort()` then `reverse()` in Python.
- To generate 'nearly sorted' cases, I used `sort()` then randomly swapped two elements N/20 times, to make the array 90% sorted.

For runtime measurements, I used `time.process_time()` in Python.

(The test experiment code was not included in the submission files. Please refer to my repo for implementation details.)

https://github.com/Superfish83/2025_2_Algorithms/tree/main/hw2/time_test

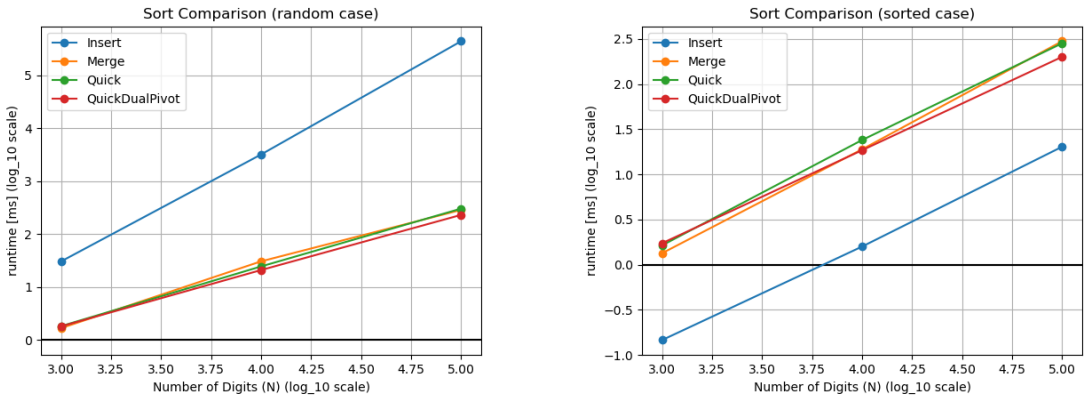
Performance Comparison

The table and figure below show the test results.

Table 2 runtime comparison results (in milliseconds).

		Insertion Sort	MergeSort	QuickSort	DualPivot QuickSort
$N = 10^3$	Random	30.55	1.68	1.81	1.81
	Sorted	0.15	1.34	1.63	1.73
	Reversed	57.32	1.35	1.76	1.69
	NearlySorted	3.68	1.47	1.57	1.66
$N = 10^4$	Random	3194.97	30.68	24.68	21.01
	Sorted	1.59	18.94	24.00	18.50
	Reversed	6697.08	19.03	24.24	19.72
	NearlySorted	678.84	25.88	22.72	21.67
$N = 10^5$	Random	440040.71	287.17	303.90	231.36
	Sorted	20.24	298.12	282.88	200.19
	Reversed	804355.77	321.39	324.33	187.09
	NearlySorted	49218.31	344.59	320.29	181.77

(a) **(b)**



(c) **(d)**

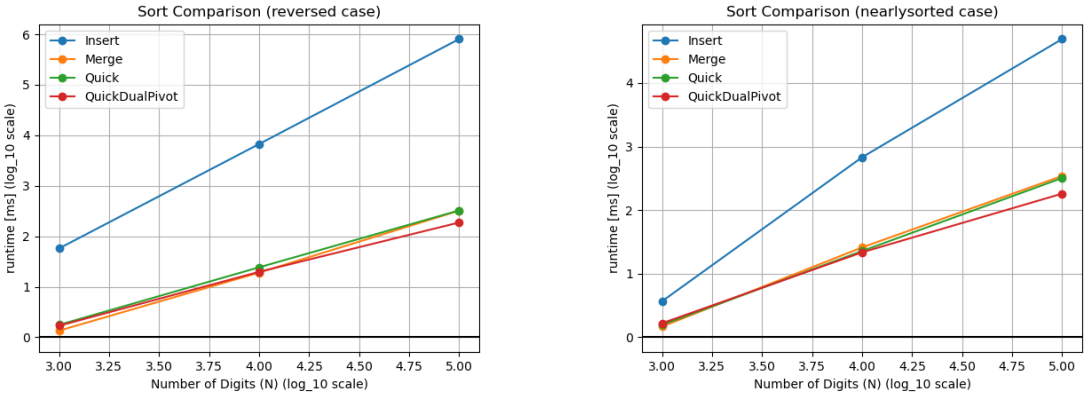


Figure 1 Runtime comparison results. Runtime versus N, the number of elements, is plotted (in log scale). (a) random case, (b) sorted case, (c) reversed case, (d) nearly sorted case.

Analysis & Conclusion

In most cases – in (a), (c), and (d), Insertion Sort showed the largest runtime. The time consumed grew significantly large as N grew. Meanwhile, it took the shortest time in (b), the sorted case.

Merge, Quick, and DualPivotQuick Sort showed a more gradual runtime growth in all cases. Among them, DualPivotQuick Sort was the fastest. Also, Quick Sort was slightly faster than Merge Sort in general.

Quick and DualPivotQuick Sort has the worst-case time complexity of $O(n^2)$. However, they outperformed the Merge Sort which has worst-case time complexity of $O(n \log n)$. It is because the randomization in Quick Sort avoids the worst-case scenario at most times. Also, Quick Sort and DualPivotQuick Sort have a smaller number of comparisons in average, as well as better memory locality.