Sorting Algorithms Performance Report

by Abusaid Aruzhan

Introduction

This report provides a clear and concise overview of the solutions to Problem Set #2,#3, focusing on sorting algorithms implemented in Python for use in Google Colab.

Below, I present the implementations, test results, complexity analysis, and a performance comparison report.

Step1: Implementing Sorting Algorithms

I developed the following sorting algorithms as per the requirements:

1. My own, the most basic O(n^2) sorting: a simple selection-like sort(takes 2 adjacent elements. If the left one is bigger than the right one, it swaps them. Does this many times until everything is in order.)

   The complexity is O(n^2) (the more elements, the longer it takes).

2. Quick sort: implemented with both random and median of three pivots to improve average case performance.
3. Merge sort: standard divide and conquer approach.
4. Heap sort: built by using a max heap structure.

Step2:Testing the Algorithms

To verify correctness, I tested the algorithms on various cases, comparing results with python's sorted() function. Test cases included random, empty, single element, sorted and reverse order arrays.

Output:

- My bad sort on [3, 1, 4, 1, 5] → [1, 1, 3, 4, 5] (matches sorted())
- Merge Sort on [] → [] (matches)
- All tests passed, confirming functionality across cases.

For larger sizes (like 1000), generate arr = [random.randint(0, 1000) for _ in range(1000)] and check.

Step3: Analysis of Sorting Algorithms

I analyzed the time and space complexities applying the Master Theorem where applicable.

1. Bad sorting: Time: O(n^2) in all cases due to nested loops. Space: O(1) in-place. No Master Theorem applies (iterative).
2. Quick Sort (Random/Median): Time: Average O(n log n), Worst O(n²) (rare with median pivot). Space: O(log n) recursion average, O(n)is worst. Master Theorem: T(n) = 2T(n/2) + O(n), a=2, b=2, log_b(a)=1, f(n)=O(n) ~ n^1, Case 2: O(n log n).
3. Merge Sort: Time: O(n log n) always. Space: O(n) for temporary arrays. Master Theorem: T(n) = 2T(n/2) + O(n), same as above, O(n log n).
4. Heap Sort: Time: O(n log n) (build heap O(n), n extracts O(log n) each). Space: O(1) in-place. Master Theorem not directly applicable (heapify recurrence T(n) = T(2n/3) + O(1) leads to O(log n) per call).

Step4: Comparing Sorting Algorithms

I compared the practical performance of My Bad Sort, Quick Sort (median pivot), Merge Sort, and Heap Sort on random datasets of sizes 100, 1000, 5000, and 10,000. Timings were measured using timeit.

Size	Bad Sort	Quick Median	Merge Sort	Heap Sort
100	0.000298	0.000112	0.000134	0.000108
1000	0.031922	0.001214	0.001467	0.001732
5000	0.905671	0.006231	0.008192	0.010584

10000	3.67234 5	0.013456	0.017823	0.023119

Analysis

- Bad Sort scales poorly ($O(n^2)$), taking ~3.67s for 10,000 elements.
- Quick Sort (Median) is the fastest in practice (~0.013s for 10k) due to efficient partitioning.
- Merge Sort and Heap Sort are consistent (~0.017s and ~0.023s for 10k), with Merge using extra space and Heap being in-place.
- For small sizes (100), all are fast (<0.001s), but the difference grows with size.

Plot Description

A plot of time vs. size would show Bad Sort rising steeply (quadratic), while Quick, Merge, and Heap grow slowly (logarithmic). This confirms theoretical expectations.