MSCS-532: Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization

Oishani Ganguly

GitHub Repository:
https://github.com/Oishani/MSCS532_Assignment5

1. Implementation of Deterministic Quicksort

`deterministic_quicksort.py` uses a deterministic algorithm for implementing Quicksort, selecting the first element of each subarray as the pivot during partitioning. The algorithm recursively sorts elements on either side of the pivot, ensuring an in-place sort. To validate correctness and robustness, a suite of test cases was developed in `test_deterministic_quicksort.py`, including edge cases (empty and single-element lists), sorted and reverse-sorted arrays, randomly generated datasets, and inputs with duplicate values. Each test confirms that the output matches Python's built-in `sorted()` result, providing confidence in the algorithm's correctness across diverse scenarios.

2. Performance Analysis of Deterministic Quicksort

Quicksort is a divide-and-conquer sorting algorithm that recursively partitions an array around a pivot element. Its time complexity depends heavily on how balanced the partitions are at each recursive step.

Best and Average Case

In the best-case scenario, the pivot always divides the array into two equal halves. This yields a recurrence of:

$$T(n) = 2T(n/2) + O(n)$$

where O(n) accounts for the partitioning step. Solving this recurrence using the Master Theorem gives a time complexity of O(n log n). This is optimal for comparison-based sorting algorithms and matches the performance of merge sort and heap sort in their best cases.

The average-case complexity is also O(n log n), but for different reasons. When pivots are chosen deterministically (e.g., always the first element), the partitioning is not guaranteed to be balanced. However, over all possible input permutations, the expected number of comparisons made during partitioning still leads to a logarithmic depth of recursive calls. According to Cormen et al. (2009), the average number of comparisons converges to 1.39(n log n), making the algorithm efficient in practice despite not guaranteeing perfectly balanced partitions.

Worst Case

In the worst-case scenario, the pivot selection is extremely poor, such as when the pivot is always the smallest or largest element. This happens with deterministic schemes like first-element pivot when the input is already sorted or reverse sorted. The resulting partitions are highly unbalanced: one subarray has n - 1 elements and the other has 0. This leads to a recurrence of:

$$T(n) = T(n-1) + O(n)$$

which solves to O(n^2). Each recursive call processes almost the entire array again, leading to quadratic time complexity. This behavior is a major limitation of deterministic pivot strategies and can be mitigated by using randomized or median-of-three pivoting.

Space Complexity

The space complexity of Quicksort is O(log n) in the best and average cases, stemming from the recursion stack depth in balanced partitions. However, in the worst case, the depth can reach O(n) due to skewed partitions, significantly increasing memory usage. Unlike merge sort, which requires O(n) additional space for merging, Quicksort operates in-place, using constant extra space for swapping and a small number of variables. This makes Quicksort generally more space-efficient than its counterparts.

Additional Overheads

Despite its elegant recursion, Quicksort's performance can be influenced by overheads such as stack memory for recursive calls and inefficient partitioning in unbalanced scenarios. Tail call optimization and hybrid approaches (e.g., switching to insertion sort for small subarrays) are commonly used in practice to improve efficiency.

3. Implementation and Analysis of Randomized Quicksort

Implementation

The implementation of randomized Quicksort in `randomized_quicksort.py` modifies the standard Quicksort algorithm by selecting the pivot element randomly from the current subarray, rather than using a fixed position like the first or middle element. This pivot is swapped to the beginning of the subarray before partitioning proceeds. The recursive sorting logic remains the same, but the random pivot selection reduces the likelihood of encountering unbalanced partitions, which can lead to worst-case performance. The implementation maintains in-place sorting and low space overhead while providing more robust performance across different input distributions. This makes it well-suited for practical applications with unpredictable input.

Analysis

Randomization significantly improves the practical performance of Quicksort by reducing the likelihood of consistently encountering its worst-case behavior. In a deterministic Quicksort, where the pivot is chosen in a fixed manner - such as always selecting the first or last element - adversarial input (like an already sorted or reverse-sorted array) can result in highly unbalanced partitions. This leads to a time complexity of O(n^2), as each recursive call processes nearly the entire subarray without meaningful size reduction (Cormen et al., 2009).

By introducing randomization into the pivot selection process, Quicksort breaks any correlation between the input order and pivot choice. Each time the algorithm runs, it picks a pivot uniformly at random from the current subarray, making it statistically unlikely that it will consistently select the worst possible pivot across recursive levels. This randomization leads to a more balanced partitioning on average, which restores the expected time complexity of O(n log n), even for inputs that would otherwise degrade performance in deterministic versions (Sedgewick & Wayne, 2011).

Furthermore, randomized Quicksort does not require any additional memory beyond what deterministic Quicksort uses, maintaining the same in-place behavior and O(log n) average-case space complexity for the recursion stack. In practice, randomized pivoting adds negligible overhead but offers a powerful safeguard against worst-case performance. It also makes the algorithm robust and suitable for use in standard libraries and production environments, where input distribution cannot be controlled.

Overall, randomization serves as a probabilistic defense mechanism that protects Quicksort's efficiency across all input cases without sacrificing its core advantages in space and average runtime performance.

4. Empirical Analysis

The three benchmark plots (`quicksort_random_benchmark.png`, `quicksort_reverse_benchmark.png`, `quicksort_sorted_benchmark.png`) vividly illustrate how pivot-selection strategy impacts Quicksort's performance in practice, and they align closely with our theoretical complexity analysis.

Random Input

On uniformly random arrays, both deterministic (first-element pivot) and randomized Quicksort exhibit near-linear scaling in wall-clock time across input sizes from 100 to 10,000. The curves are almost straight lines, reflecting the O(n log n) average-case behavior. Deterministic

Quicksort is marginally faster here because it avoids the overhead of sampling a random pivot, but the difference remains constant as n grows. This small constant gap is the cost of calling `random.randint` and swapping. Asymptotically it does not change the n log n curve.

Sorted and Reverse‑Sorted Input

For both pre-sorted and reverse-sorted data, deterministic Quicksort's runtime skyrockets, curving sharply upward. By n=5000, it already takes half a second and continues to grow superlinearly. This is the hallmark of its worst-case O(n^2) recurrence, where every partition splits into sizes (1, n - 1). Each level of recursion does Θ(n) work but only reduces the problem size by one, so the total cost sums to:

$$\sum_{k=1}^{n} k = \Theta(n^2).$$

In stark contrast, randomized Quicksort remains flat and grows only modestly: even at n=10,000, times are on the order of 0.01 s. By choosing a pivot uniformly at random, the algorithm probabilistically avoids consistently poor splits. The expected subarray sizes stay balanced, preserving O(n log n) performance for every distribution we tested.

Conclusion

- **Average‑Case Efficiency***:* On random inputs, both versions behave as expected in theory - O(n log n). The slight constant‑factor penalty for randomization is visible but negligible for large n.
- **Worst‑Case Vulnerability***:* Deterministic pivot selection catastrophically degrades on already sorted sequences, exactly matching the theoretical O(n^2) worst-case. Randomization immunizes the algorithm against any fixed adversarial ordering.
- **Practical Trade-offs***:* If data is genuinely random, a simple deterministic pivot may be fastest; but in library or production settings where input characteristics are unknown (or potentially adversarial), randomized Quicksort delivers robust performance with only minimal extra cost.

These empirical results reinforce the theoretical lesson: while pivot strategy does not change the asymptotic average-case complexity, it makes all the difference between catastrophic worst-case blow-up and consistently reliable n log n behavior.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). *MIT Press*.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.