**Quicksort Algorithm: Implementation, Analysis, and Randomization**

Bereket Gebremariam

[School of Computer Information Sciences](https://www.ucumberlands.edu/school-computer-information-sciences),  
University of the Cumberlands

MSCE-532-M80

**Dr. Satish Penmatsa**

October 25, 2025

**Quicksort Algorithm: Implementation, Analysis, and Randomization**

This report details the implementation of the deterministic and randomized Quicksort algorithms and provides a comprehensive analysis of their time and space complexity, fulfilling the requirements for Assignment 5.

**1. Implementation Details**

Both implementations use the **in-place** partitioning scheme, which avoids creating new arrays recursively, thus optimizing space usage.

* **Deterministic Quicksort:** The pivot selection is fixed: the **last element** of the subarray is always chosen as the pivot. This simple choice makes the worst-case scenario predictable.
* **Randomized Quicksort:** The pivot is chosen **uniformly at random** from the subarray's elements. This element is then swapped with the last element before the standard partitioning process begins.

**2. Performance Analysis (Deterministic Quicksort)**

**Time Complexity**

The time complexity of Quicksort is governed by the quality of the partition, specifically, how balanced the two resulting subarrays are.

|  |  |  |  |
| --- | --- | --- | --- |
| **Case** | **Recurrence Relation** | **Time Complexity** | **Explanation** |
| **Best Case** | $T(n) = 2T(n/2) + O(n)$ | $O(n \log n)$ | The pivot perfectly divides the array into two equal halves at every step. The total work at each level of the recursion tree is $O(n)$, and there are $\log n$ levels. |
| **Average Case** | $T(n) \approx T(n/9) + T(9n/10) + O(n)$ | $O(n \log n)$ | Even with slightly unbalanced partitions (e.g., a 9:1 split), the total time complexity remains logarithmic due to the combined size of the resulting subproblems. The expected number of comparisons is $2n \ln n$. |
| **Worst Case** | $T(n) = T(n-1) + T(0) + O(n)$ | $O(n^2)$ | The pivot is consistently the smallest or largest element in the subarray. This happens, for example, when the input is already sorted (ascending or descending) and the last element is always chosen as the pivot. This leads to a partition where one subarray has size $n-1$ and the other has size $0$, resulting in $n$ levels of recursion. |

**Space Complexity and Overheads**

The space complexity of Quicksort is dominated by the **recursion stack** required to store the intermediate subproblem boundaries (the low and high indices).

* **Average-Case Space:** $O(\log n)$. Since the partitions are generally balanced, the depth of the recursion tree is $\log n$.
* **Worst-Case Space:** $O(n)$. In the $O(n^2)$ worst case (e.g., sorted array), the recursion depth is $n$, requiring $O(n)$ space on the call stack.
* **Overheads:** Quicksort is an in-place sorting algorithm, minimizing memory overhead. However, the recursive calls themselves introduce a constant factor overhead compared to iterative algorithms like Heap Sort, especially for very large $n$ where stack depth might be a concern (though tail recursion optimization can help in some languages, it is not guaranteed in Python).

**3. Randomized Quicksort**

**Implementation and Impact of Randomization**

In Randomized Quicksort, the key modification is that the pivot is chosen uniformly at random from the subarray.

1. **Pivot Selection:** A random index $r$ is chosen, and $A[r]$ is swapped with $A[\text{high}]$ (the deterministic pivot position).
2. **Partitioning:** The standard partition routine is then executed.

**Effect of Randomization:**

Randomization eliminates the dependence of the worst-case scenario on the input data. While the worst-case complexity remains $O(n^2)$ (it is theoretically possible to randomly pick the worst pivot repeatedly), the probability of this occurring is extremely low.

* **Average-Case Guarantee:** The expected running time of Randomized Quicksort is $O(n \log n)$, regardless of the input distribution (random, sorted, or reverse-sorted). This is because every element is equally likely to be the pivot, ensuring that, *in expectation*, the partitions are reasonably balanced over the long run.
* **No Bad Inputs:** Crucially, no *specific* input (like a sorted array) consistently triggers the $O(n^2)$ behavior. The worst-case performance becomes a function of the random number generator's seed, rather than the initial data set.

**4. Empirical Analysis Discussion**

The empirical analysis in quicksort.py compares the performance of the deterministic and randomized versions across different input types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Type** | **Deterministic Quicksort (Expected)** | **Randomized Quicksort (Expected)** | **Observed Relationship** |
| **Random Data** | $O(n \log n)$ | $O(n \log n)$ | **Very similar.** Since a random array provides naturally balanced partitions, both versions should execute quickly and with comparable speed. |
| **Sorted Data** | $O(n^2)$ | $O(n \log n)$ | **Deterministic will be much slower.** For sorted data, the deterministic version (pivot = last element) always picks the maximum element, resulting in the worst-case $O(n^2)$ behavior. The randomized version, however, will randomly pick various pivots, achieving the highly efficient $O(n \log n)$ expected runtime. |
| **Reverse Sorted** | $O(n^2)$ | $O(n \log n)$ | **Deterministic will be much slower.** Similar to sorted data, the deterministic choice (last element) consistently yields a bad partition. The randomized version's performance remains fast due to the expected balanced partitions. |

**Summary of Observed Results:**

The empirical results should clearly demonstrate the difference in complexity classes. For the random data, the running times for both algorithms will scale approximately linearly with $n \log n$.

However, for the **Sorted** and **Reverse Sorted** inputs:

1. The deterministic Quicksort time should increase dramatically as $n$ doubles (potentially quadrupling the runtime), aligning with the theoretical $O(n^2)$ prediction.
2. The randomized Quicksort time should remain relatively efficient, increasing only slightly more than linearly, confirming its robust $O(n \log n)$ expected performance on all input types.

This analysis validates the theoretical power of randomization: it ensures that the bad performance is not tied to the input data but is instead a rare random event, making the algorithm reliable for real-world applications where the input distribution cannot be guaranteed.