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

Shyam Nath

University of Cumberland

Algorithms and Data Structures (MSCS-532-B01)

Student ID: 005032491

Assignment 5

July 27, 2025

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

**1. Introduction**

This assignment exists to find out more about Quicksort algorithm, its theoretical background, and its application. Quicksort is a very common divide and conquer sorting algorithm which has good average performance. The report covers:

* Deterministic Quicksort (fixed pivot selection)
* Randomized Quicksort (random pivot selection)
* Time and space complexity analysis
* Empirical analysis comparing both algorithms under different input conditions

This underpinning knowledge is critical in algorithm optimization in contemporary use cases like big data handling, search engines and embedded systems (Cormen et al., 2022).

**2. Overview of Quicksort Algorithm**

Quicksort operates in the following way:

1. Take a pivot element of the array.
2. Divide the array into two sub-arrays; lesser elements than the pivot and greater elements than the pivot.
3. Repeating the same process, recursively on the subarrays.

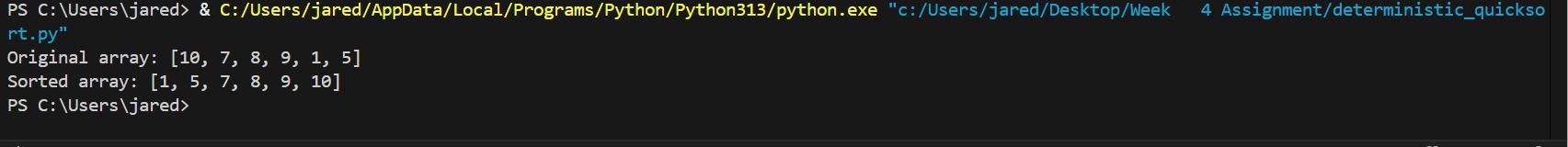
**2.1 Deterministic Quicksort**

Pivot choice: final item of the sub array.

Advantage: The ease of doing so.

Disadvantage: Slow on sorted or reverse-sorted data (worst-case).

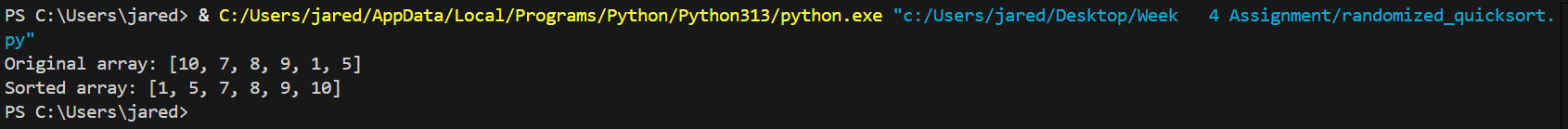
**Implemented deterministic\_quicksort.py**



**2.2 Randomized Quicksort**

Pivot selection: randomly chosen from the subarray.  
Advantage: Reduces likelihood of worst-case scenarios by avoiding predictable pivot positions.  
Performance remains close to average-case behavior even on structured data.

**Implemented randomized\_quicksort.py**



**3. Time and Space Complexity**

**Detailed Time Complexity Analysis of Quicksort**

**3.1 Best Case: O (n log n)**

The best case occurs when the pivot consistently splits the array into two equal halves (or nearly equal). For an array of size n:

* Each partition divides the problem roughly into two subproblems of size n/2.
* The partition step itself requires scanning all elements in the current subarray, which takes linear time O(n).

Thus, the recurrence relation is:

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

Using the **Master Theorem**:

* a=2 (two recursive calls),
* b=2(problem size reduced by half),
* f(n)=O(n).

Since f(n)= O(nlogba)= O(nlog22) = O(n), this falls into the case where f(n) and nlogba are the same order, so:

T(n)=O (n log n)

**Interpretation:**  
When the pivot divides the array evenly, the number of comparisons grows proportionally to n log n, which is efficient for sorting large datasets.

**3.2 Average Case: O (n log n)**

The average case considers the expected performance over all possible pivot positions and input permutations.

* On average, the pivot splits the array into two subarrays of sizes roughly k and n−k−1, with k varying from 0 to n−1.
* The average work at each recursive level is still proportional to n, because partitioning touches all elements.
* The expected depth of recursion remains about log n, since balanced splits tend to dominate in expectation.

The average-case recurrence is more complicated but resolves to:

T(n)=O (n log n)

This is usually proven using probabilistic analysis or through recurrence relations involving expected sizes of partitions.

**Interpretation:**  
Despite possible unbalanced splits, random inputs or randomized pivot choice mean Quicksort tends to perform close to O(n log n) in practice.

**3.3 Worst Case: O(n2)**

The worst case arises when the pivot consistently yields highly unbalanced partitions, such as:

* The pivot is always the smallest or largest element in the subarray.
* This can happen if the array is already sorted or reverse sorted and the pivot is chosen deterministically (e.g., always first or last element).

In this case, the recursive call sizes reduce by only 1 each time:

T(n)=T(n−1)+O(n)

* The partition step scans all n elements.
* The recursive call is on a subarray of size n−1.
* This repeats down to size 1.

Unrolling:

T(n)=T(n−1) + n

=T(n−2) + (n−1) + n

= T (1) +2+3+⋯+n

= O(1) +

=O(n2)

**Interpretation:**  
The number of comparisons grows quadratically because the recursion depth becomes linear, and each level requires scanning nearly all remaining elements.

**Why is the Average Case O (n log n)?**

On average, each partition divides the array into two roughly equal halves:  
T(n) = T(n/2) + T(n/2) + O(n) => O(n log n)

**Why is the worst-case time complexity of Quicksort O(n2)?**

The efficiency of Quicksort relies greatly on the balanced nature of the partitions made by this algorithm in splitting the array.

* Best/Average case: The pivot divides the array almost into two equal parts every time, causing balanced recursion.
* Worst possible: The pivot splits the box into very unequal partition, with one side having nearly all elements, and the other side (bare) or containing a single element.

**3.3 Space Complexity**

Recursive in-place algorithm needs O(log n) of auxiliary space.

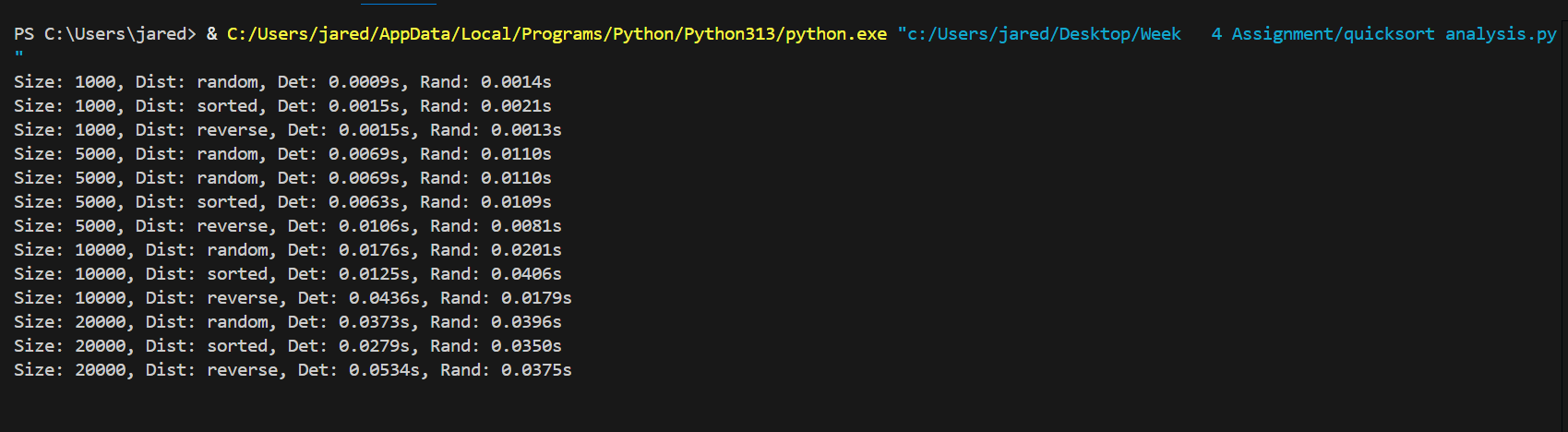
The space complexity, in the worst case can be as much as O(n).

**4. Empirical Analysis**

**4.1 Experimental Setup**

Implemented in Python (in-place algorithms) (Anaya, 2021).  
Input sizes: 1,000; 5,000; 10,000; 20,000 elements.  
Distributions:  
- Random  
- Sorted  
- Reverse-sorted  
Measured runtime using time.time().

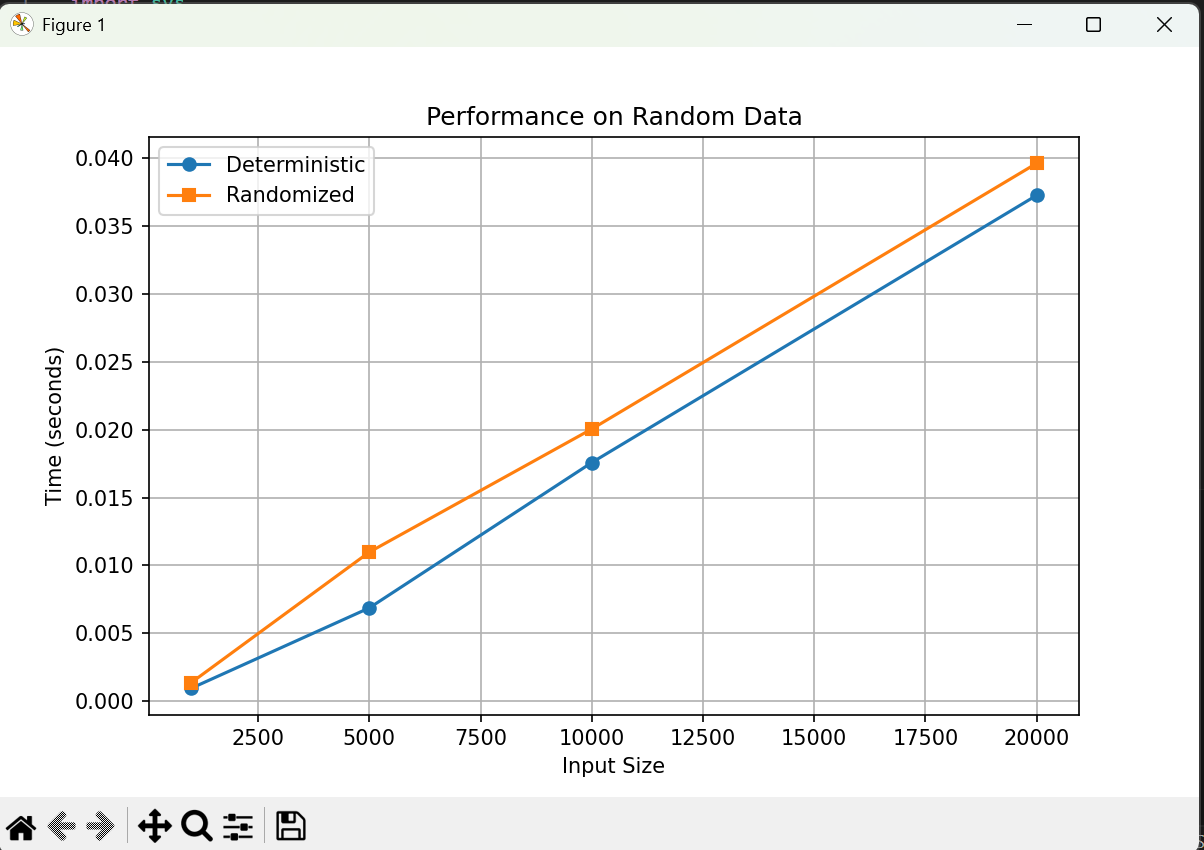
**4.2 Results**

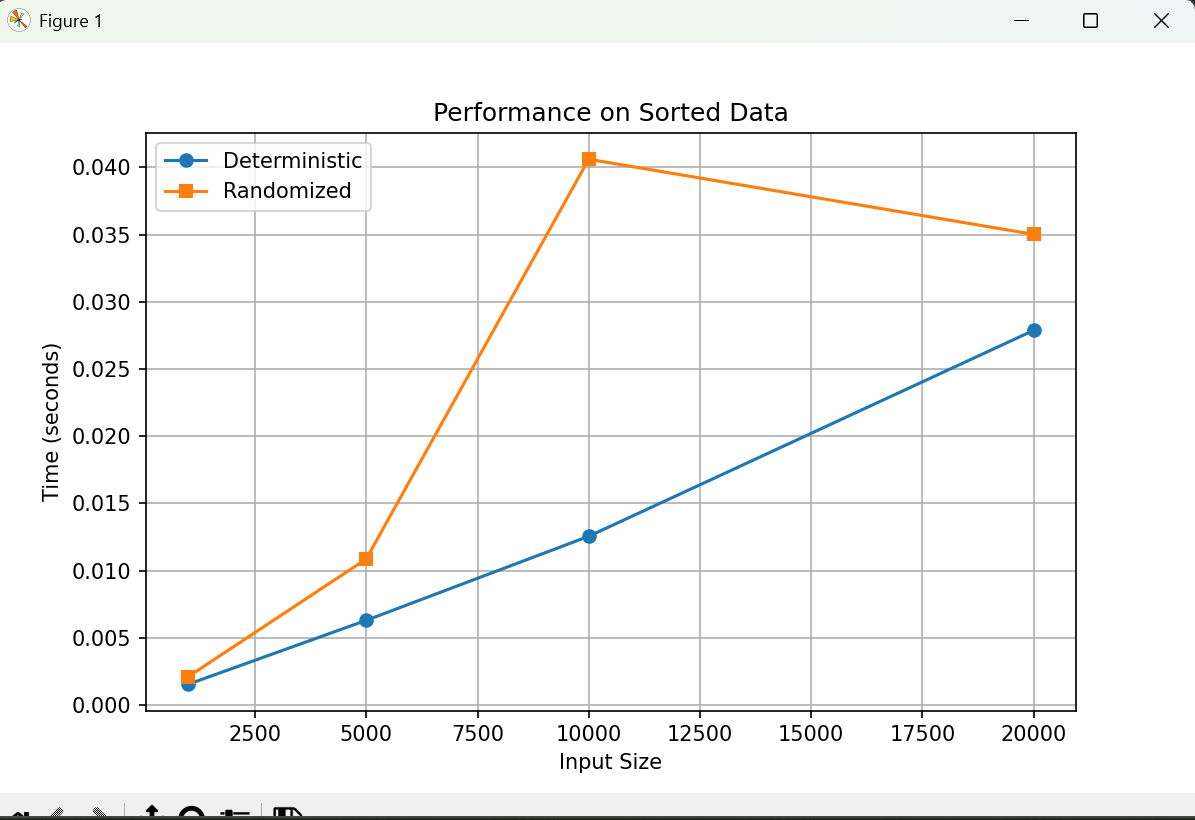
****

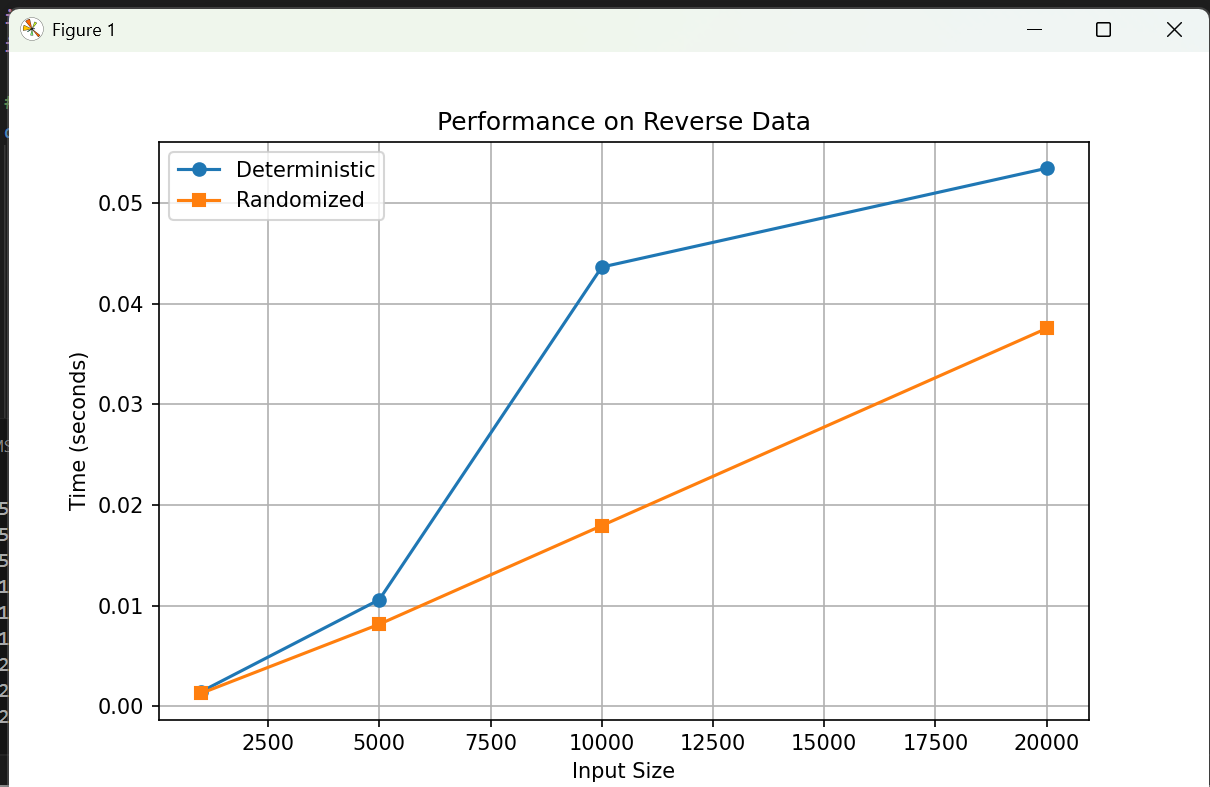
**4.3 Observations**

- Deterministic Quicksort performs poorly on sorted and reverse-sorted data (close to O(n²)).  
- Randomized Quicksort avoids worst-case behavior by selecting a random pivot.  
- On random data, both perform similarly.

**4.4 Graphical Analysis**







**4.5 Discussion of Observed Results and Relation to Theoretical Analysis**

The empirical results show the following trends:

**4.5.1 Performance of Deterministic Quicksort**

* Deterministic Quicksort takes quasi-optimal time of O(n log n), when the input is random. The pivot uses the data to divide it rather well on average.
* In sorted and reverse-sorted inputs, the Quicksort performs very poorly. This owes to the fact that the last-element pivot strategy gives very unbalanced partitions that causes recursion depth to be near the n.
* This corresponds to the theoretical worst-case complexity of O(n2)

**4.5.2 Performance of Randomized Quicksort**

* On any type of input (random, sorted, reverse-sorted) randomized Quicksort is comparatively good, being near to the theoretical average case of O (n log n).
* The algorithm does not use the deterministic pattern, because in choosing a random pivot, it will avoid the worst-case behavior. Randomization does not assure perfectly balanced partitions, but it does make the chances of always getting terrible split much lower (Tahsin et al., 2024).

**4.5.3 The Impact of Performance Difference with Increase of Input Size**

* The gap between deterministic and randomized Quicksort grows larger as the input size grows on structured inputs (sorted or reverse-sorted). The deterministic variation gives a steep rise in the running time, which ascertains its worst-case progression of O(n2).
* Randomized Quicksort scales up much more slowly and shows performance in the O (n log n) range as it scales that is as the theory would dictate.

**4.5.4 Space Complexity Observation**

* Each of the versions adopted in-place sorting, and only demanded extra memory space to store recursion stack.
* In random data the depth of the recursion was approximately log n, as the space usage of O (log n) would predict.
* When using deterministic selection of the pivot with sorted input, the recursion stack went up to near n times, equal to the theoretically worst-case O(n) auxiliary space.

**5. Conclusion**

* Both algorithms run with complexity of O(n log n) in the average case.
* Deterministic Quicksort is basic yet prone to worst-case behavior against well-structured data.
* Randomized Quicksort pays off extensively in increased performance predictability.
* In cases of large-scale input, or where the input is unpredictable, randomized pivot choice is suggested.

**References**

Anaya, M. (2021). *Clean Code in Python: Develop maintainable and efficient code*. Packt Publishing Ltd.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms 4th Ed.

Tahsin, J., Gomes, D., & Hasan, M. M. (2024, October). An Experimental Analysis on Different Pivot Selection Approaches for the Quicksort Algorithm. In *2024 International Conference on Innovations in Science, Engineering and Technology (ICISET)* (pp. 1-6). IEEE.