**Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomisation**

Luqman Ali Syed

University

Algorithms and Data Structures - Bi-term2Week 4  
Professor Name

July 23, 2025

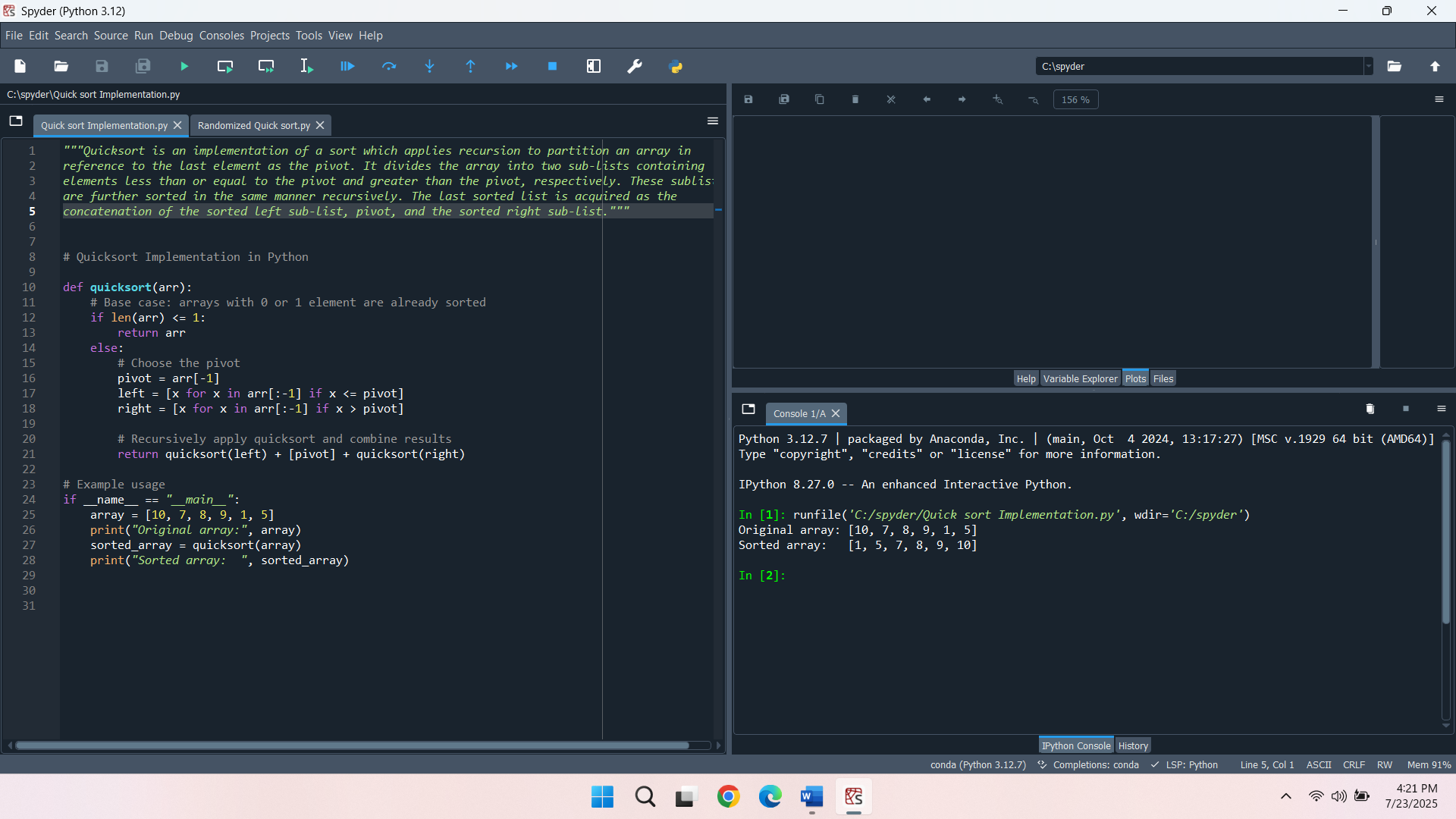
**Introduction**

Quicksort is an effective sorting algorithm with broad application that implements the divide-and-conquer approach. Famous in terms of its O(nlogn) average-case time complexity. Quicksort is a desirable algorithm in numerous real-world settings, such as system-level libraries to the processing of large amounts of data. This piece of work will examine both the deterministic and the randomised versions of Quicksort, both in terms of implementation, performance analysis, and their comparison under different test scenarios. Focusing on theoretical research and practical experiments, the assignment identifies the advantages and drawbacks of both directions and underlines the necessity to focus on the selection of pivots that can guarantee the best performance.

**Quicksort Implementation and Analysis**

**1. Implementation**

Quicksort is an implementation of a sort which applies recursion to partition an array in reference to the last element as the pivot. It categorizes the array into two sub-lists with elements that are less than or equal to the pivot and those that are bigger than the pivot, respectively. These sub-lists are further sorted in the same manner recursively (Aftab, 2021). The last sorted list is acquired as the concatenation of the sorted left sub-list, pivot, and the sorted right sub-list.



**2. Performance Analysis**

**Time Complexity of Quicksort**

* **Best Case O(nlogn):** An ideal situation occurs when at each level of call, the pivot partitions the array into two halves. This results in a balanced recursion tree with logn depth, and each level processes all n elements during partitioning, leading to O(nlogn) overall.
* **Average Case - O(nlogn):** The pivot separates the array into reasonably balanced sections on average. The splits are not always best, but good enough to maintain the depth around logn. Since each level still scans the full array for partitioning, the total time complexity remains O(nlogn).
* **Worst Case - O(n2):** The worst time complexity is when the pivot is the minimum or maximum in every iteration, leading one subarray to have n-1 elements and the other having 0 elements. This leads to a recursion depth of n and a total of n+(n−1)+(n−2)+...+1=O(n2) comparisons.

**The Average-Case Time Complexity is O(nlogn) and the Worst Case is O(n2):**

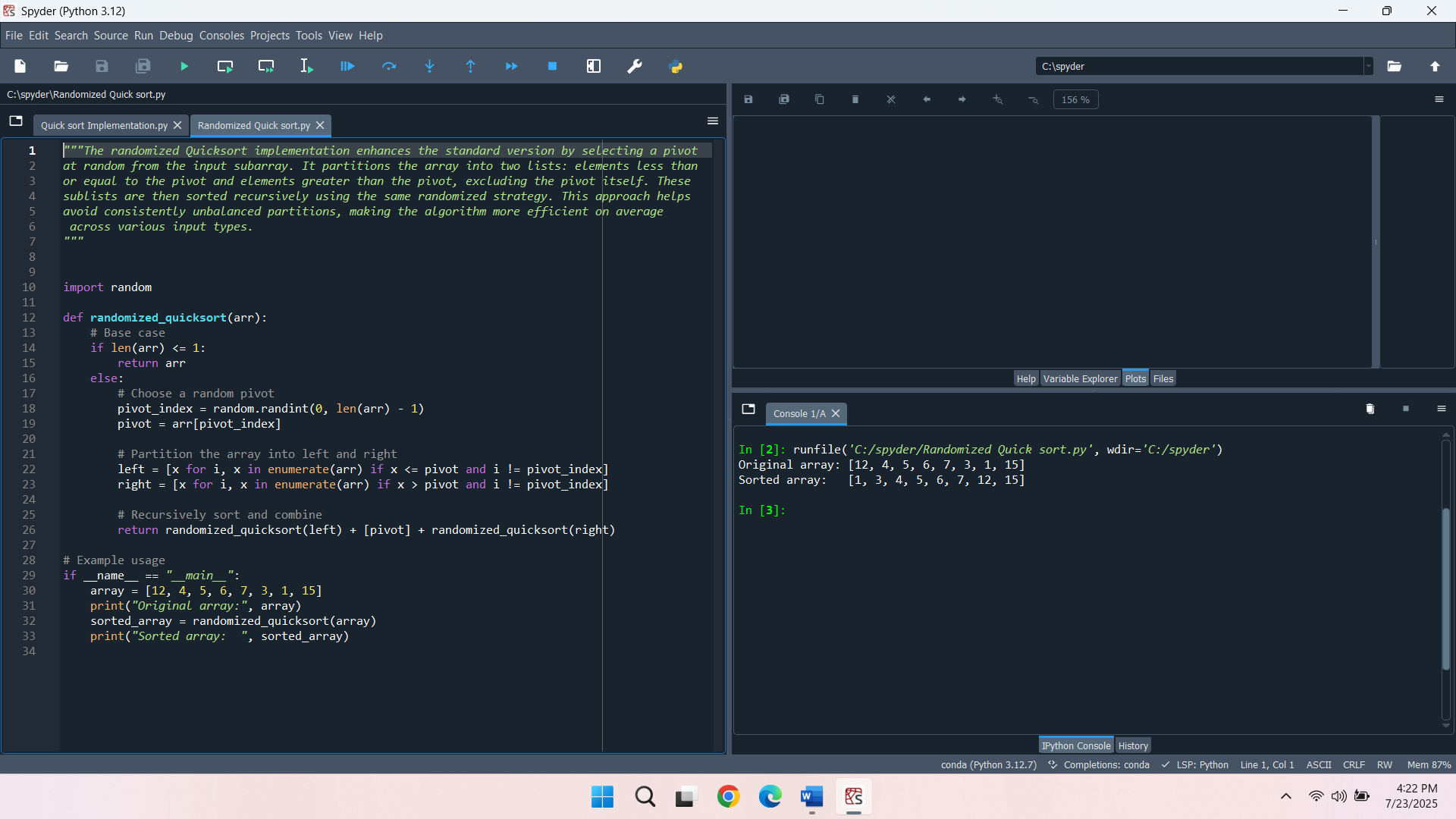
The average-case complexity is O(nlogn) because, with random or unsorted data, the pivot tends to split the array into reasonably balanced subarrays over time. This balanced recursion tree is logn deep, and n partitions take place in each step (Kalmodiya & Dixit, 2024). In contrast, the worst-case O(n2) arises when the pivot causes highly unbalanced splits, leading to a linear-depth tree with n levels, where each level processes nearly all remaining elements.

**Space Complexity and Additional Overheads:**

* **Space Complexity:** In the best and average cases, the recursive depth is logn, so the space complexity is O(logn) due to the function call stack. In the worst case, the recursive depth becomes n, leading to a space complexity of O(n).
* **Additional Overheads:** Minimal overhead because Quicksort is an in-place sorting algorithm, it does not have to use additional arrays such as Merge Sort. Nevertheless, it suffers overhead due to recursion, and it may suffer stack overflows and grow slowly when using poor pivot choices. This risk can be mitigated by selecting pivots at random or using a median of three strategies.

**3. Randomised Quicksort**

The randomised Quicksort version improves the simple one since it randomly chooses a pivot within the input subarray. It divides the array into two lists: elements less than or equal to the pivot and elements greater than the pivot, not the pivot itself. These sub-lists are optimised again with the same randomised technique used in sorting recursively. The method allows for avoiding repeatedly lopsided partitions, so the algorithm is on average more efficient over a wide range of inputs (Wassenberg, 2024).



**Analysis**

Randomisation significantly reduces the chance of Quicksort hitting its worst-case time complexity of O(n2). In the deterministic model, poor pivot selection, such as smallest or largest element in sorted or almost sorted arrays, however, always causes unbalanced partitions. Nonetheless, when a pivot is chosen randomly, the chances of disproportionately unbalanced splits between recursive calls are negligibly small.

Consequently, randomised Quicksort has a time complexity of O(nlogn) even in an adversarial setting on average. It is strong and effective to use in practice, particularly where the distribution of inputs is unseen or perhaps designed in such a manner as is otherwise likely to make the performance worse. The randomness averages out average-case performance in more consistent data types.

**4. Empirical Analysis**

**Empirical Comparison of Running Time**

In comparing deterministic and randomised versions of the Quicksort algorithm, they were implemented on different-sized arrays with different distributions. In this analysis, the input types were:

* Random arrays
* Sorted arrays
* Reverse-sorted arrays

In order to extend the performance, arrays with varying sizes were used in each of the inputs. To find the consistency each of the algorithms was timed several times using standard timing functions.

The deterministic Quicksort that employs a fixed pivot worked well on random inputs and it generally will exhibit a near O(nlogn) behaviour. However, the algorithm showed extreme performance degradation in the case that the input was sorted or reverse-sorted, in that case, the partitions were never balanced. Such scenarios contributed to almost O(n2) behaviour, particularly, with increasing input.

The randomised Quicksort in its turn was stable with any input data. This was done in the algorithm by making a random choice of pivot to avoid redundant repetitive poor partition. Randomised Quicksort also had an average time complexity of near O(n log n) when run on sorted or reverse-sorted arrays, but times were significantly lower on average than they were on deterministic Quicksort on the same arrangement.

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

The introduced results can serve to strengthen the theoretical analysis of the Quicksort performance with little effort. As expected, deterministic Quicksort is also subject to an input pattern that causes unbalanced splits with the fixed pivot, causing a worst-case time complexity of O(n2). This may be witnessed in the test placed on the sorted as well as reverse-sorted array where the input size would scale by leaps and bound of the input size.

Randomised Quicksort, however, was illustrated to possess efficient and solid behaviour against all input types. The randomness over learning prevents the probability of always performing bad splits and therefore prevents the probability of worst-case behaviour. This conforms with the theoretical intuition that the expected time complexity of randomised Quicksort is O(n logn) no matter whether the input is considered to be first ordered or not.

In conclusion, it can be said that deterministic Quicksort is likely to be efficient only with random data, whereas the randomised version is more reliable and can scale with size, which is more valid in situations when it is impossible to guarantee the input properties.

**Conclusion**

Quicksort, both in deterministic and randomised forms, shows how extraordinarily enormous the role of pivot selection can be to determine the efficiency of an algorithm. The deterministic version is very good on randomly ordered inputs, but very bad on sorted and reverse-sorted on unbalanced partitioning. Randomised Quicksort avoids this disadvantage by choosing pivots randomly and, as a result, demonstrates near-optimal performance on all kinds of input. The theoretical results are justified by empirical analysis, and randomisation indeed makes the algorithm more robust and safer against the worst-case behaviour. All in all, there should be an analysis of these varieties of Quicksort to understand and apply them all to give the developers and data scientists the equipment to construct faster, more scalable systems.

**References**

Aftab, A., Ali, M. A., Ghaffar, A., Shah, A. U. R., Ishfaq, H. M., & Shujaat, M. (2021). Review on performance of quick sort algorithm. *Int. J. Comput. Sci. Inf. Secur.(IJCSIS)*, *19*(10.5281).

Kalmodiya, T., & Dixit, M. (2024, February). Exhaustive Analysis and Time Complexity Evaluation of Sorting Algorithms. In *2024 2nd Intesrnational Conference on Computer, Communication and Control (IC4)* (pp. 1-6). IEEE.

Wassenberg, J., Blacher, M., Giesen, J., & Sanders, P. (2022). Vectorized and performance‐portable quicksort. *Software: Practice and Experience*, *52*(12), 2684-2699.