**Assignment 3: Understanding Algorithm Efficiency and Scalability**

Luqman Ali Syed

University

Algorithms and Data Structures - Bi-term2Week 3

Professor Name

July 16, 2025

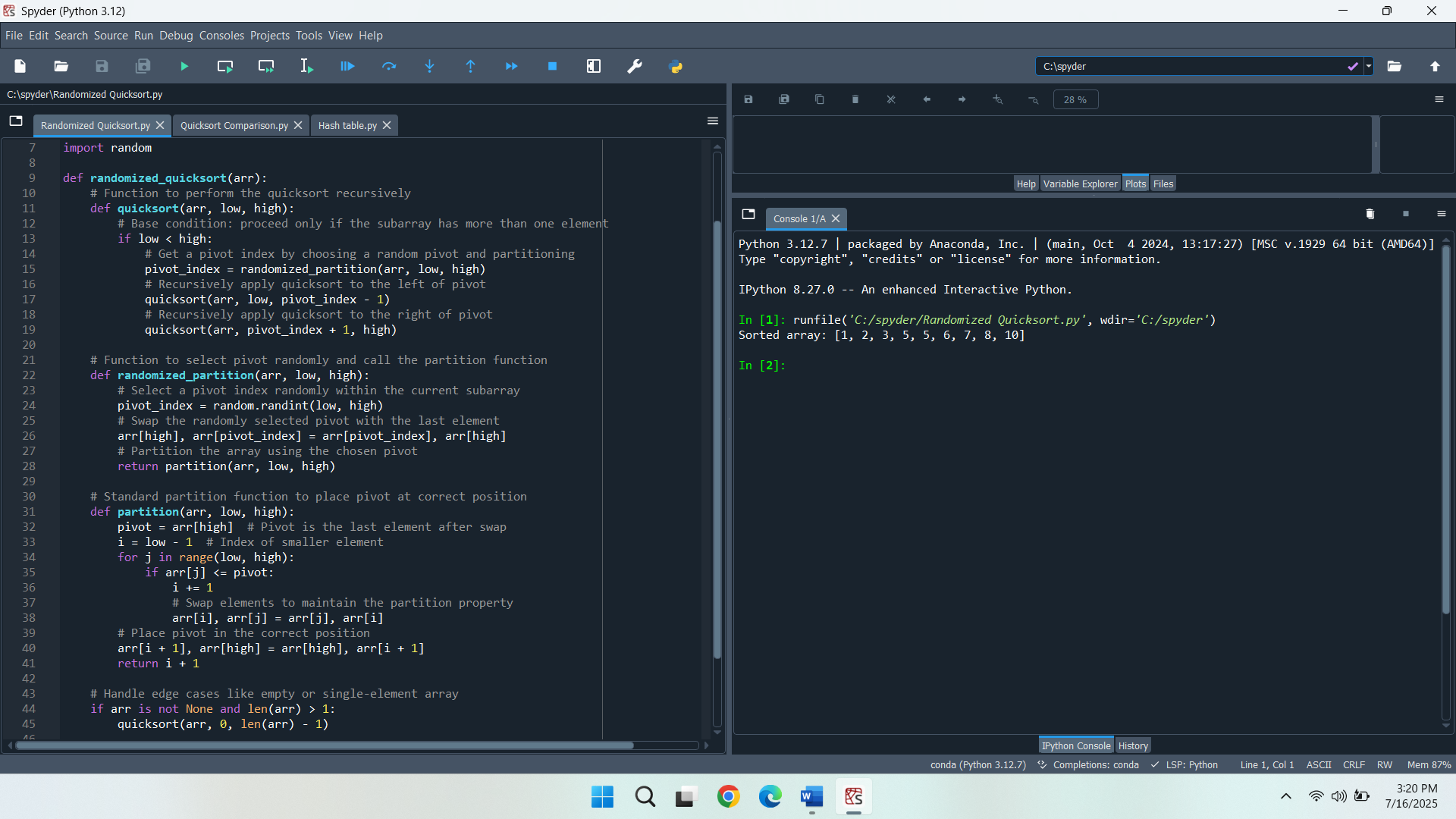
**Understanding Algorithm Efficiency and Scalability**

**Introduction**

Investigates the complexity and efficiency of most common algorithms, Randomized Quicksort and Hashing with Chaining. Randomized Quicksort is an improvement of the standard Quicksort, each pivot is chosen randomly to decrease the probability of worst-case performance. With Hashing with Chaining, the problem of collision in hash table is resolved by keeping linked lists at each index, collision is therefore sparse making hash table searches simple and fast. The implementation and test of the two algorithms are done with different input conditions so as to compare their time complexity and resource consumption. The benchmarking is aiming at deciphering how these algorithms run in terms of size and distribution. Comparison results give information on selecting the right algorithm depending on requirements. The research backing facilitates the direction of effective and scalable designs of software.

**Part 1: Randomized Quicksort Analysis  
1. Implementation**

Randomized Quicksort is a sorting algorithm which improves the standard Quicksort by choosing a pivot uniformly at random among the elements of the current subarray. That random choice reduces the likelihood of worst-case outputs with sorted or virtually sorted inputs. The algorithm divides the array in a way that the elements, which are lower than the pivot, are shifted to its left, and the ones which are higher shifted to its right. This is again done on the resulting subarrays (Mohammadagha, 2025). The implementation is robust by properly dealing with edge cases such as empty arrays, arrays of repeated values and already sorted inputs.



**2. Analysis**

**Average-Case Time Complexity Analysis of Randomized Quicksort**

The time complexity of randomized quicksort, in average case, is calculated as the time complexity of a set of steps taken during sorting based on the expected number of comparisons. Each time the algorithm recurs, a random pivot is chosen in the subarray, and using that pivot the array is partitioned. The randomly selected pivots do guarantee that the array is divided into more or less balanced partitions on average.

**Recurrence Relation Technique**

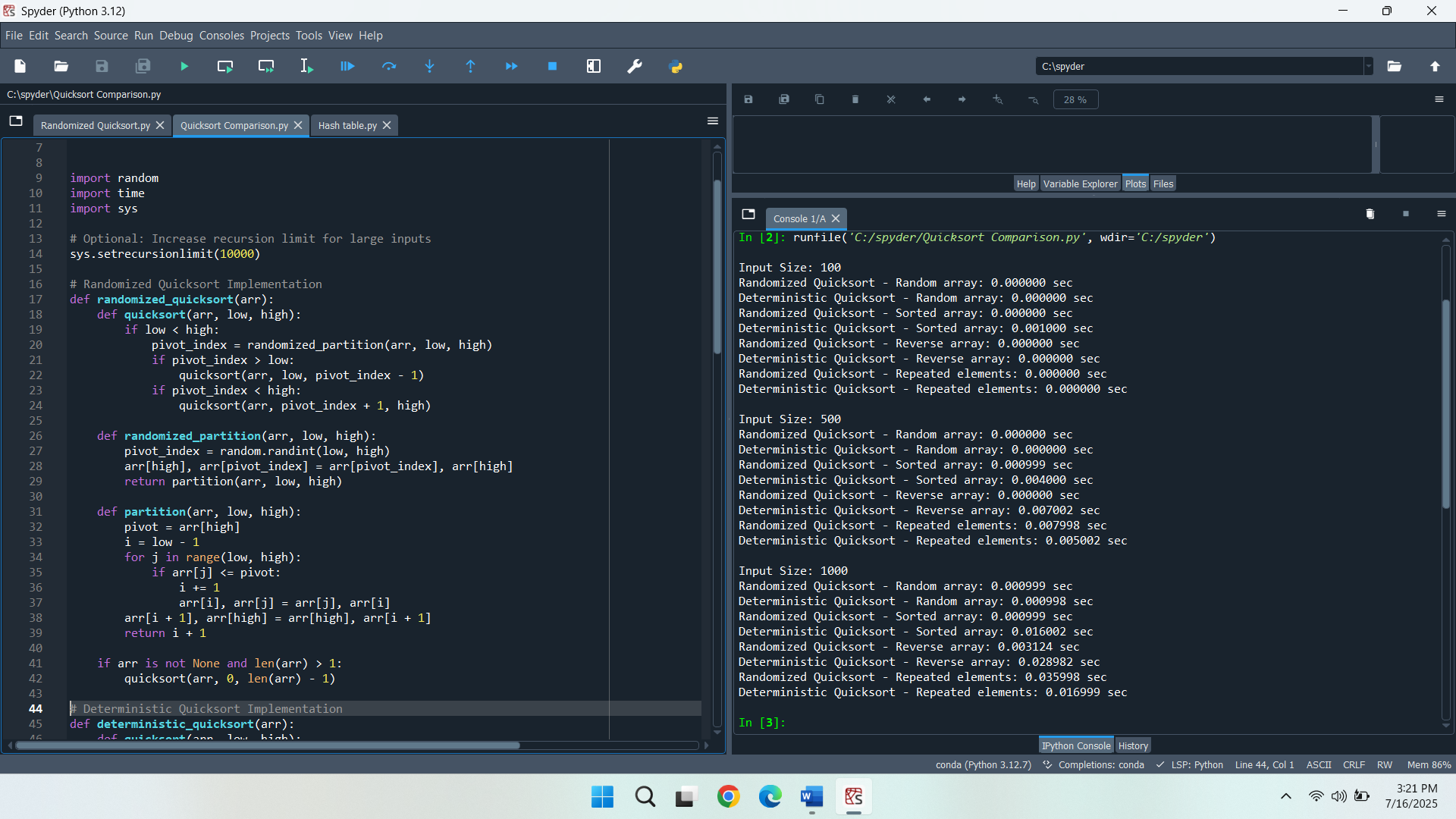
The recurrence relation representing the expected number of operations for an input of size n can be expressed as T(n) = T(k) + T(n − k − 1) + O(n), where k is the number of elements less than the pivot. Because the pivot is selected randomly, the value of k also varies uniformly across the range of values 0 to n 1. Taking the average over all possible k leads to the recurrence T(n) = (1/n) \* ∑(from k=0 to n−1) [T(k) + T(n−k−1)] + O(n). This results in T(n) = O(n log n) when solved using the substitution method or recursion tree analysis.

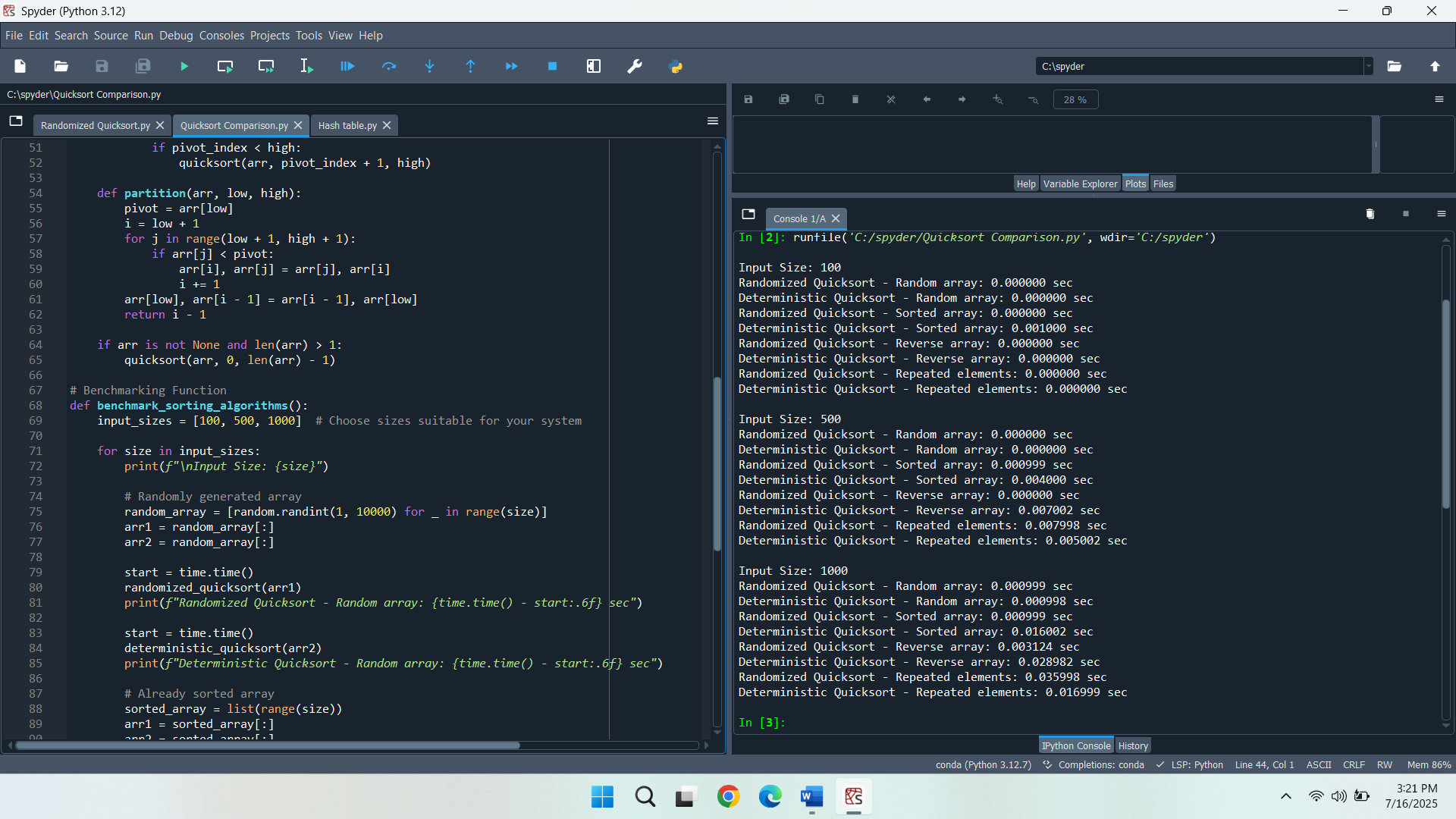
**Indicator Random Variable Approach**

X is the total number of comparisons done by the algorithm. For each pair of distinct elements i and j, define an indicator random variable X\_ij, which is 1 if elements i and j are compared and 0 otherwise. The expectation that i and j would be compared is two divided by the number of elements between i and j plus one because the index of the pivot is chosen randomly. Summing over all pairs gives the expected number of comparisons as E[X] = ∑ E[X\_ij] = O(n log n), confirming the average-case complexity.

**3. Comparison**

Applies two sorting algorithms, Randomized Quicksort and Deterministic Quicksort based on the first element and sets the first as the pivot. Each algorithm is applied to arrays of different nature such as random, sorted, reverse-sorted, and repeated elements. Each code also performs time testing on every case and compares the performance using time module on three different sizes of input (Goodrich & Kitagawa, 2024). Randomized Quicksort chooses a random pivot to balance the partition, whereas Deterministic Quicksort always uses the first element, and this is dangerous to balanced partitions. This benchmarking capability creates and stresses every kind of array, and prints the running times to assess performance.





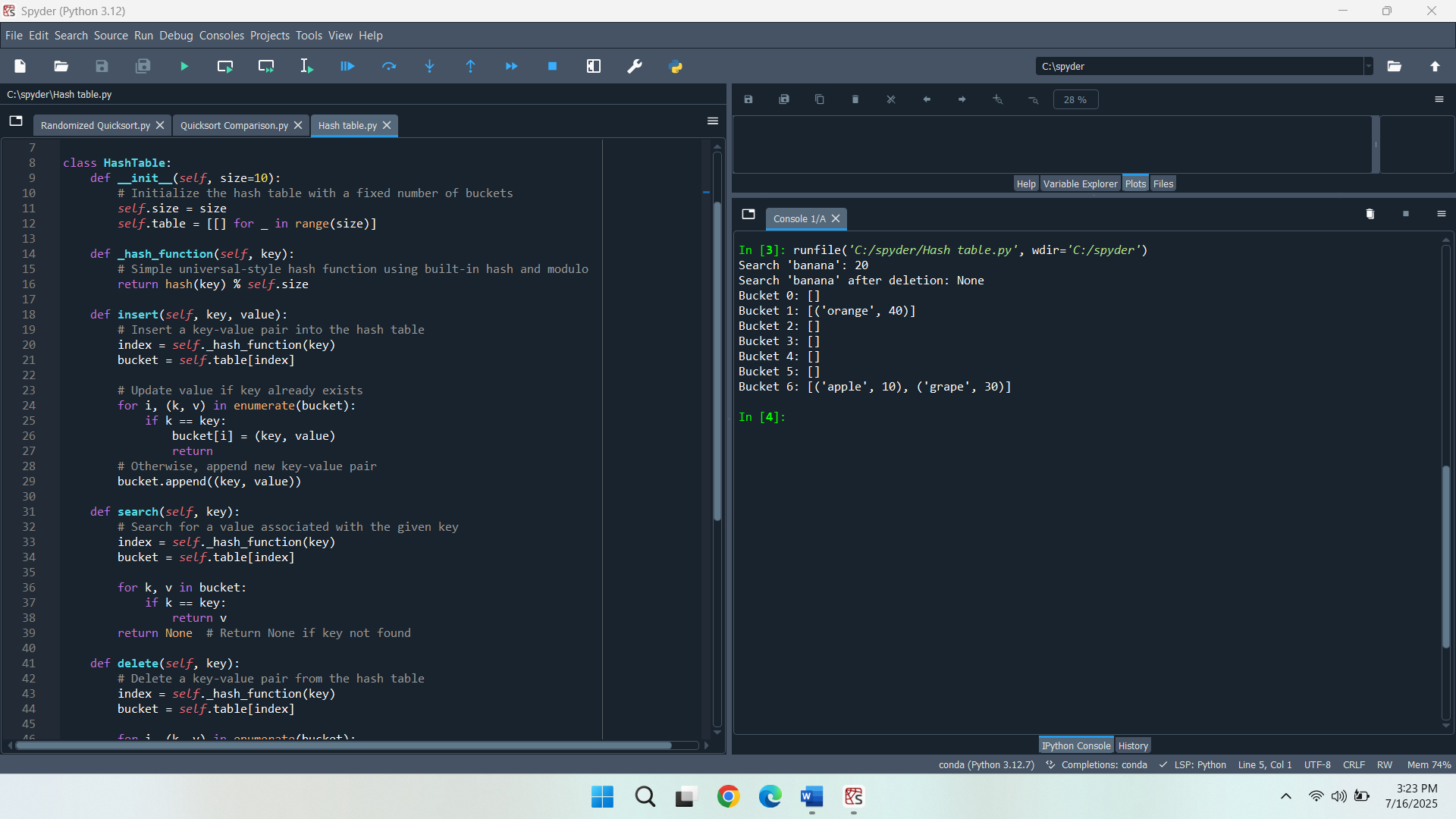
**Discussion of Observed Differences and Theoretical Analysis**

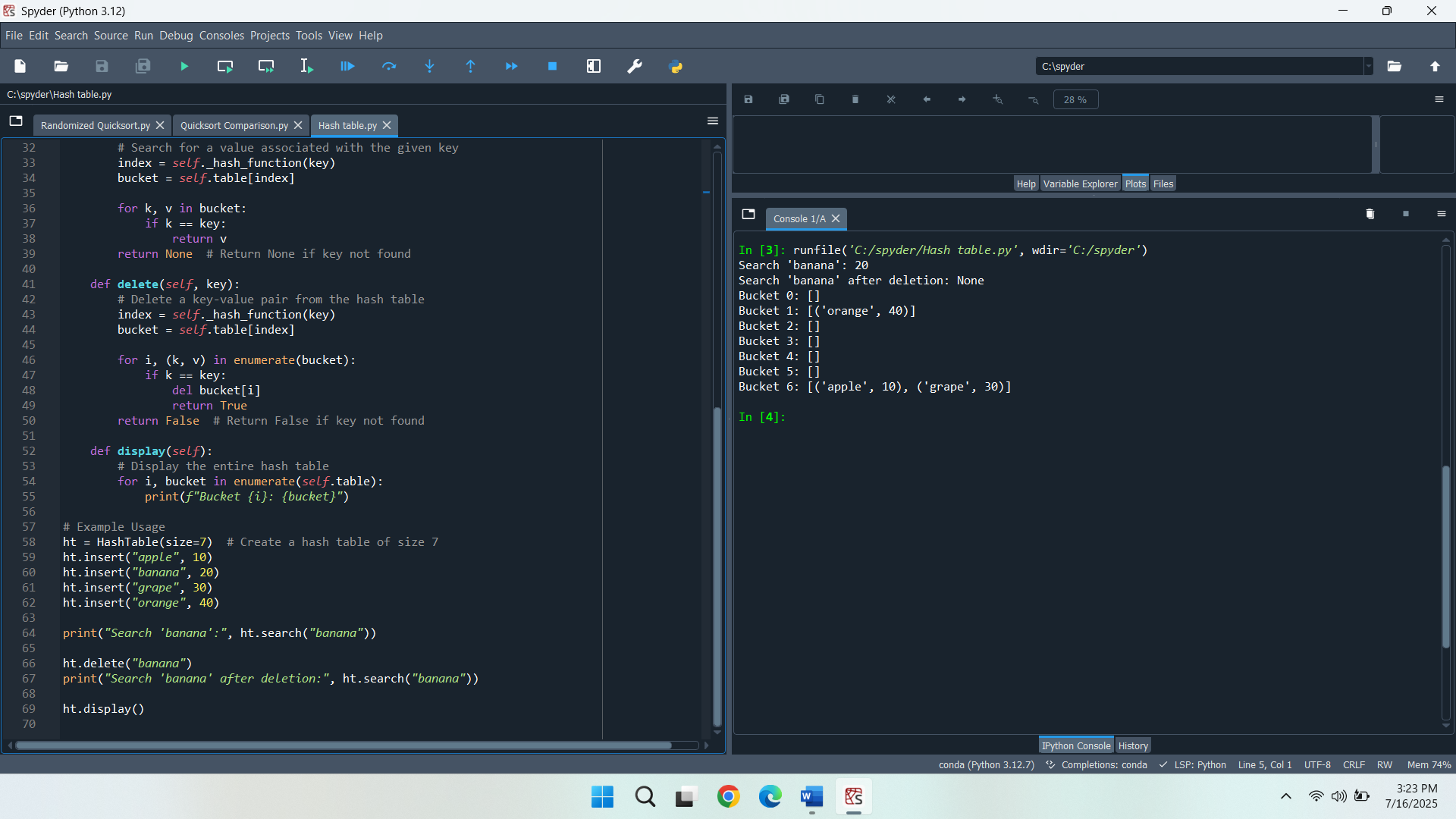
The Randomized Quicksort is efficient over all input kinds and sizes because of its pivot selection strategy. For random arrays, both sorting methods demonstrate comparable and fast runtimes as expected under average-case conditions of O(n log n). However, for already sorted and reverse-sorted inputs, Deterministic Quicksort performs significantly worse as input size increases, which confirms its worst-case time complexity of O(n²) when poor pivot choices lead to unbalanced partitions. Randomized Quicksort avoids this degradation through its probabilistic pivoting approach, maintaining balanced partitions and stable O(n log n) performance.

When repeated elements are involved, both algorithms suffer a slight slowdown, whereby Randomized Quicksort occasionally has a slightly higher time. Such difference in theoretical expectation can be caused because in such memory-limited or stack-limited systems the overhead of repetitive calls and pivot randomization may be used. Also, both implementations do not employ optimized methods such as the three-way partitioning, more appropriately dealing with duplicates. All in all, the findings confirm the expectations of the theory on the one hand, and tout the strength of Randomized Quicksort and frailty of Deterministic Quicksort given a set input. There are some minor differences which can be explained by practical constraints like interpreter overheads or at system level.

**Part 2: Hashing with Chaining  
1. Implementation**

Hashing with chaining is an algorithm in use to address the collision arising in a hash table by storing more than one key-value pair at each index using a linked list or dynamic array. A hash function designed to be a part of a universal family assists in distributing keys throughout the table relatively evenly and reducing the collisions to a minimum ensuring constant-time practices on the average, whereas deletions, insertions, and searches are maintained (Sakan et al., 2022). With this implementation, there are fundamental operations, insert to add a key-value pair, search to find a value associated with a key, and delete to delete a key in the table. Chaining makes it so that when there are more keys with the same index they are still processed but there is no overwrite.





**2. Analysis**

**Time Complexity Analysis under Simple Uniform Hashing**

Under the assumption of simple uniform hashing, where each key is equally likely to hash into any of the available slots, the expected time complexity for search, insert, and delete operations in a hash table with chaining is O(1 + α), where α is the load factor. As the hash function allocates keys uniformly, the size of each element in hash buckets is roughly similar and hence each operation is time efficient on average. Insertion entails hashing the key and adding it to the correct bucket and is constant time on average. Deletion and searching need to scan the bucket, but the bucket will average only a small number of elements, so both these operations will also be close to constant time.

**Impact of Load Factor on Performance**

The load factor α is defined as the ratio of the number of stored elements n to the number of available slots m in the table (α = n/m). Greater load factor results in more significant possible collisions, which result in longer chains within each bucket. This will corner the performance directly by spending more time on search and delete operations, as they can be compelled to go through a greater number of elements in a bucket. A low load factor provides shorter chain usage and faster access times, where as a high load factor can cause performance to go to linear time in the worst case.

**Strategies for Maintaining Low Load Factor and Minimizing Collisions**

Low load factor is necessary for the constant time performance. This can be done using dynamic resizing so that when a specific load is hit, the table would be set to twice its size. Under resizing, all existing elements rehashed and placed in a new table with added slots to join more keys in it. Selecting a good hash function according to a universal family reduces clustering to a minimum, and balances out the distribution of keys. Further performance techniques include employing prime numbers as table size together with the delinking of concerns of key types in hashing in order to decrees collision rates and facilitate even hashing behaviour.

**Conclusion**

Randomized Quicksort performs consistently and efficiently against the inputs of differing type because of its balanced partitioning strategy, facilitated by random pivot selection. Compared to Deterministic Quicksort, it does not have this performance degradation on sorted or reverse-sorted data, where fixed pivot selection causes recursion to be heavily unbalanced and results in higher time complexity; that is, it does not use a fixed pivot like Deterministic Quicksort. Hashing by chaining offers a workable means of managing hash clashes in hash tables in the assumption of even hashing distributions, and stably runs insert, search, and delete keys. The operations are carried out on the basis of the load factor, which determines the mean chain length within the table. High quality hash functions and dynamic resizing of load factor guarantee a good distribution of keys and reduced performance bottlenecks. Collectively, the implementations and analyses show the direct effect of algorithmic design and efficiency tuning of computer performance on efficiency in data manipulation and operations.

**References**

Goodrich, M. T., & Kitagawa, R. (2024). Making Quickhull More Like Quicksort: A Simple Randomized Output-Sensitive Convex Hull Algorithm. *arXiv preprint arXiv:2409.19784*.

Mohammadagha, M. (2025). Hybridization and Optimization Modeling, Analysis, and Comparative Study of Sorting Algorithms: Adaptive Techniques, Parallelization, for Mergesort, Heapsort, Quicksort, Insertion Sort, Selection Sort, and Bubble Sort.

Sakan, K., Nyssanbayeva, S., Kapalova, N., Algazy, K., Khompysh, A., & Dyusenbayev, D. (2022). DEVELOPMENT AND ANALYSIS OF THE NEW HASHING ALGORITHM BASED ON BLOCK CIPHER. *Eastern-European Journal of Enterprise Technologies*, *116*(9).