Understanding Algorithm Efficiency and Scalability

Student Name: Sindhuja Martha

Algorithms and Data Structures (MSCS-532-M80): **Assignment 3**

MS in Computer Science, University of the Cumberlands, KY

Professor: **Dr. Solomon**

February 8, 2026

Understanding Algorithm Efficiency and Scalability

Randomized Quicksort Analysis

Randomized Quicksort is a comparison-based sorting algorithm that selects the pivot uniformly at random from the subarray being partitioned. This approach reduces the likelihood that adversarial input patterns will consistently produce unbalanced partitions, which is a primary cause of worst-case performance in deterministic versions.

Average-Case Time Complexity Analysis

Let T(n) represent the expected running time to sort an array of size n. Because the pivot is selected uniformly at random, each element has an equal probability of selection. After partitioning, the array is divided into two subarrays whose sizes depend on the rank of the pivot. The expected running time satisfies the following recurrence:

E[T(n)]=1n∑k=0n−1(E[T(k)]+E[T(n−k−1)])+Θ(n)E[T(n)] = \frac{1}{n} \sum_{k=0}^{n-1} \left(E[T(k)] + E[T(n-k-1)] \right) + \Theta(n)E[T(n)]=n1k=0∑n−1(E[T(k)]+E[T(n−k−1)])+Θ(n)

The Θ(n) term represents the cost of partitioning the array around the pivot. Solving this recurrence produces an expected running time of O(n log n), a result that is well established in the algorithm analysis literature (Cormen et al., 2022).

This result can also be derived using indicator random variables. For any pair of elements (i, j), these elements are compared only if one is selected as the first pivot among all elements between them. The probability of this occurrence is 2/(|j - i| + 1). Summing the expected number of comparisons over all pairs yields an expected O(n log n) total number of comparisons, thereby confirming the average-case complexity.

Empirical Comparison with Deterministic Quicksort

Experimental Setup

An empirical comparison was conducted between Randomized Quicksort and Deterministic Quicksort, with the latter utilizing the first element as the pivot. Median runtimes, measured in milliseconds, were collected over multiple runs for input sizes from 1,000 to 20,000 elements. Four distinct input distributions were evaluated in these experiments:

- Randomly generated arrays
- Already sorted arrays
- Reverse-sorted arrays
- Arrays containing repeated elements

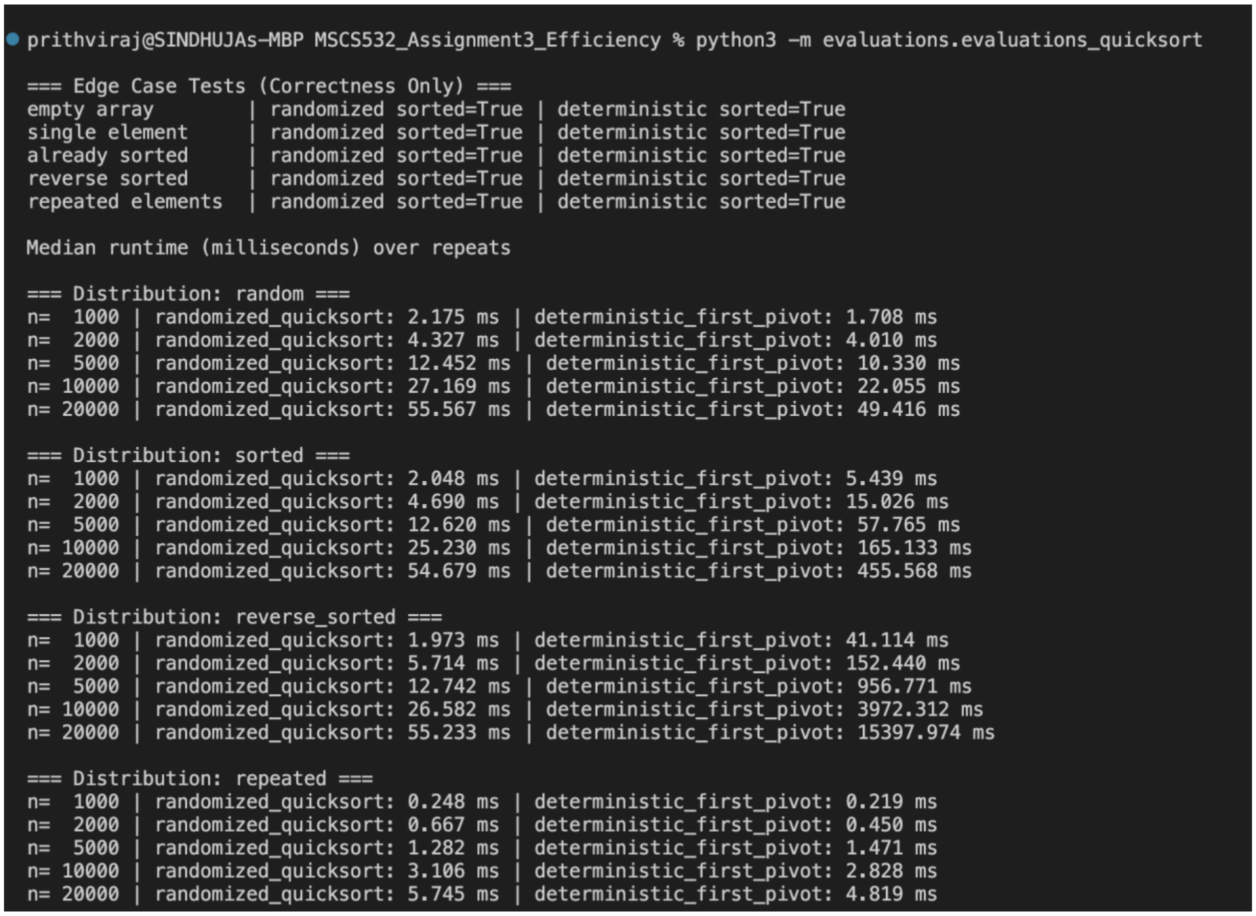
Correctness tests were additionally performed on edge cases, including empty arrays and single-element arrays, to ensure algorithmic validity. These cases were excluded from timing measurements because their execution cost was negligible.

Results and Discussion

For **randomly generated arrays**, both Randomized Quicksort and Deterministic Quicksort demonstrate similar growth trends consistent with O(n log n) complexity. Deterministic Quicksort is occasionally marginally faster for smaller input sizes, likely due to reduced constant overhead because it does not require random pivot selection.

For **already sorted arrays**, Deterministic Quicksort exhibits a substantial increase in runtime as input size increases. For instance, at n = 20,000, the deterministic execution time exceeded 450 milliseconds, whereas Randomized Quicksort remained close to 55 milliseconds. This observation is consistent with the theoretical worst-case time complexity of O(n^2) for deterministic pivot selection.

For **reverse-sorted arrays**, the performance disparity is even more pronounced. Deterministic Quicksort demonstrates severe degradation, with runtimes exceeding 15 seconds at n = 20,000, while Randomized Quicksort continues to scale efficiently. These results confirm that reverse-sorted input represents a classical adversarial case for first-pivot Quicksort.


```
prithviraj@SINDHUJAs-MBP MSCS532_Assignment3_Efficiency % python3 -m evaluations.evaluations_quicksort

=== Edge Case Tests (Correctness Only) ===
empty array        | randomized sorted=True | deterministic sorted=True
single element     | randomized sorted=True | deterministic sorted=True
already sorted     | randomized sorted=True | deterministic sorted=True
reverse sorted     | randomized sorted=True | deterministic sorted=True
repeated elements  | randomized sorted=True | deterministic sorted=True

Median runtime (milliseconds) over repeats

=== Distribution: random ===
n=  1000 | randomized_quicksort: 2.175 ms | deterministic_first_pivot: 1.708 ms
n=  2000 | randomized_quicksort: 4.327 ms | deterministic_first_pivot: 4.010 ms
n=  5000 | randomized_quicksort: 12.452 ms | deterministic_first_pivot: 10.330 ms
n= 10000 | randomized_quicksort: 27.169 ms | deterministic_first_pivot: 22.055 ms
n= 20000 | randomized_quicksort: 55.567 ms | deterministic_first_pivot: 49.416 ms

=== Distribution: sorted ===
n=  1000 | randomized_quicksort: 2.048 ms | deterministic_first_pivot: 5.439 ms
n=  2000 | randomized_quicksort: 4.690 ms | deterministic_first_pivot: 15.026 ms
n=  5000 | randomized_quicksort: 12.620 ms | deterministic_first_pivot: 57.765 ms
n= 10000 | randomized_quicksort: 25.230 ms | deterministic_first_pivot: 165.133 ms
n= 20000 | randomized_quicksort: 54.679 ms | deterministic_first_pivot: 455.568 ms

=== Distribution: reverse_sorted ===
n=  1000 | randomized_quicksort: 1.973 ms | deterministic_first_pivot: 41.114 ms
n=  2000 | randomized_quicksort: 5.714 ms | deterministic_first_pivot: 152.440 ms
n=  5000 | randomized_quicksort: 12.742 ms | deterministic_first_pivot: 956.771 ms
n= 10000 | randomized_quicksort: 26.582 ms | deterministic_first_pivot: 3972.312 ms
n= 20000 | randomized_quicksort: 55.233 ms | deterministic_first_pivot: 15397.974 ms

=== Distribution: repeated ===
n=  1000 | randomized_quicksort: 0.248 ms | deterministic_first_pivot: 0.219 ms
n=  2000 | randomized_quicksort: 0.667 ms | deterministic_first_pivot: 0.450 ms
n=  5000 | randomized_quicksort: 1.282 ms | deterministic_first_pivot: 1.471 ms
n= 10000 | randomized_quicksort: 3.106 ms | deterministic_first_pivot: 2.828 ms
n= 20000 | randomized_quicksort: 5.745 ms | deterministic_first_pivot: 4.819 ms
```

Figure 2 shows Output for Randomized Quicksort Algorithm in VSCode for dataset sizes 1000, 2000, 5000, 10,000, 20,000

Dataset Sizes = 1,000 and 2,000

Dataset Type	**1,000 (RQ)**	**1,000 (DQ)**	**2,000 (RQ)**	**2,000 (DQ)**
Random	2.132	2.656	4.810	3.946
Sorted	2.274	5.683	4.364	15.257
Reverse Sorted	1.963	38.471	4.570	159.137
Repeated Elements	0.276	0.243	0.602	0.462

Table 1: Shows median execution time (milliseconds) for Randomized Quicksort (RQ) and Deterministic Quicksort (DQ) on small input sizes.

Dataset Sizes = 5,000 and 10,000

Dataset Type	**5,000 (RQ)**	**5,000 (DQ)**	**10,000 (RQ)**	**10,000 (DQ)**
Random	12.423	10.330	27.584	24.009
Sorted	12.956	61.483	28.171	190.475
Reverse Sorted	12.494	967.903	28.400	4,028.410
Repeated Elements	1.503	2.003	3.752	3.007

Table 2: Highlights scalability differences between Randomized and Deterministic Quicksort as input size increases.

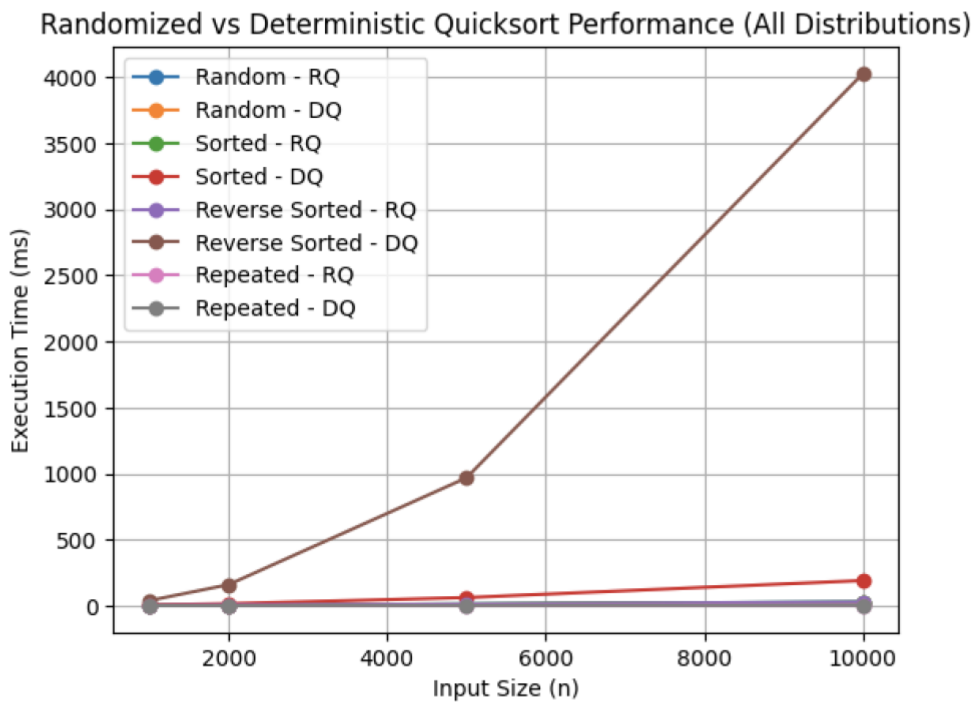


Figure 3 shows Graph for the values for Randomized and Deterministic Quicksort implementation

For **arrays containing repeated elements**, both algorithms perform efficiently, with runtimes substantially lower than those observed for other input distributions. The implementation of three-way partitioning groups equal elements together, thereby reducing unnecessary comparisons and producing near-linear performance.

In summary, the empirical results provide strong support for the theoretical analysis. Randomized Quicksort consistently avoids worst-case scenarios, whereas Deterministic Quicksort remains highly sensitive to input order. Minor deviations from theoretical predictions are attributable to implementation overhead, Python interpreter costs, and system-level timing noise, rather than algorithmic inefficiencies.

Hashing with Chaining

Expected Operation Costs under Simple Uniform Hashing

A hash table utilizing chaining resolves collisions by storing multiple key–value pairs in a single bucket through a list-based structure. Under the assumption of **simple uniform hashing**, each key is equally likely to be assigned to any of the m buckets.

Let n denote the number of stored elements, and let $\alpha = n/m$ represent the load factor. Given these assumptions, the expected time complexities are as follows:

Insert: O(1+α)O(1 + \alpha)O(1+α)

Search: O(1+α)O(1 + \alpha)O(1+α)

Delete: O(1+α)O(1 + \alpha)O(1+α)

These bounds arise because each operation requires scanning only the expected length of a single chain, which is proportional to the load factor (Cormen et al., 2022).

Impact of Load Factor on Performance

The load factor directly determines the average chain length. As α increases, chains become longer and operations require more time to complete. In the observed implementation, the hash table stored two elements in four buckets, resulting in a load factor of 0.5. At this load factor, insert, search, and delete operations were performed efficiently and produced correct results.

Maintaining a low load factor ensures that the expected cost of each operation remains approximately constant. If the load factor increases without control, the hash table's performance degrades toward linear time complexity.

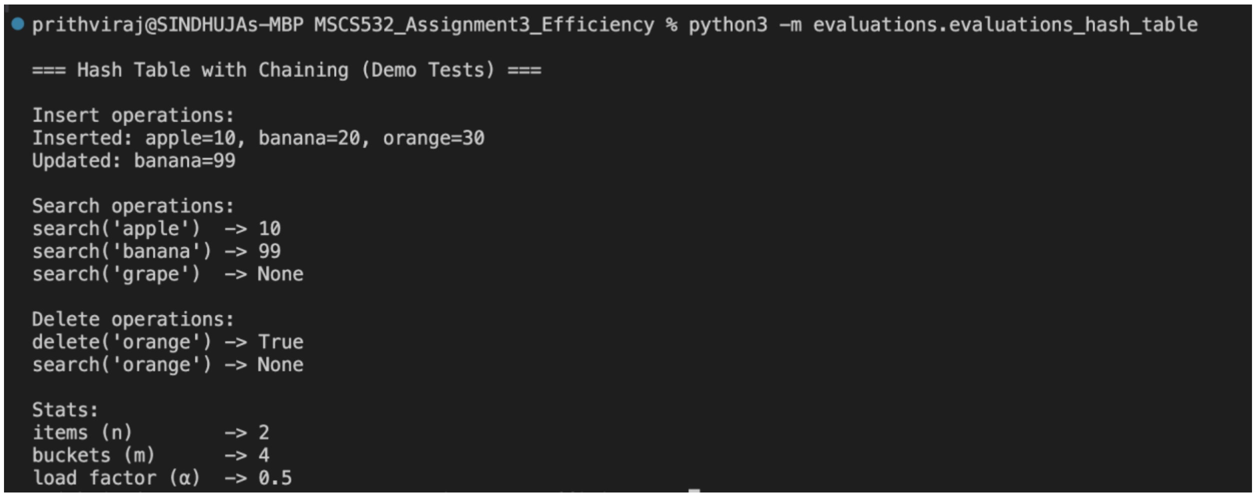
Collision Minimization and Dynamic Resizing

Hash tables maintain efficient performance by employing several key strategies:

- **Dynamically resizing the table** when the load factor surpasses a predefined threshold.
- **Rehashing** elements into a larger table to decrease chain lengths
- **Utilizing well-distributed hash functions** to minimize clustering

These strategies ensure that collisions remain manageable and that theoretical performance guarantees are maintained as the number of stored elements increases.

Results and Discussion: Hashing with Chaining



```
prithviraj@SINDHUJAs-MBP MSCS532_Assignment3_Efficiency % python3 -m evaluations.evaluations_hash_table

=== Hash Table with Chaining (Demo Tests) ===

Insert operations:
Inserted: apple=10, banana=20, orange=30
Updated: banana=99

Search operations:
search('apple')  -> 10
search('banana') -> 99
search('grape')  -> None

Delete operations:
delete('orange') -> True
search('orange') -> None

Stats:
items (n)       -> 2
buckets (m)     -> 4
load factor (α)  -> 0.5
```

Figure 2 shows Output for Hashing with Chaining Algorithm implementation in VSCode Terminal

The hash table employing chaining was empirically evaluated using a sequence of insert, search, and delete operations. The results indicate correct functionality for all supported operations. Insert operations added new key–value pairs and updated existing keys, as demonstrated by the change in the value associated with the key *banana* from 20 to 99. These results confirm that the implementation handles duplicate keys by overwriting values instead of creating redundant entries.

Search operations returned the correct values for existing keys (*apple* and *banana*) and produced None for a missing key (*grape*), demonstrating proper handling of unsuccessful searches. Delete operations also functioned as expected: the key *orange* was removed successfully, and subsequent search attempts confirmed its absence from the table.

The reported statistics indicate that the hash table stored two elements across four buckets, resulting in a load factor of 0.5. At this load factor, the expected chain length remains small, which accounts for the efficient completion of all operations. This empirical behavior is consistent with the theoretical expectation that insert, search, and delete operations execute in $O(1 + \alpha)$ expected time under simple uniform hashing.

The low load factor observed in the experiment plays a critical role in minimizing collisions and maintaining constant-time performance. Because the table size exceeds the number of stored elements, collisions are infrequent and chains remain short. In practical implementations, dynamic resizing is commonly used to preserve this property as the number of elements grows. By increasing the number of buckets when the load factor exceeds a threshold, the hash table avoids performance degradation and maintains its expected time bounds.

Overall, the empirical results validate the theoretical analysis of hashing with chaining. The implementation demonstrates correct functionality and confirms that maintaining a low load factor is essential for achieving efficient average-case performance.

Conclusion

Theoretical analysis and empirical evidence demonstrate that Randomized Quicksort achieves an average-case time complexity of O(n log n) and significantly outperforms deterministic pivot selection on adversarial inputs. Similarly, hashing with chaining yields efficient expected performance for insertion, search, and deletion operations when the load factor is appropriately managed. These findings underscore the critical role of randomness and structural design in achieving scalable and robust algorithmic performance.

References

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