MSCS-532: Assignment 3: Understanding Algorithm Efficiency and Scalability

Oishani Ganguly

GitHub Repository:
https://github.com/Oishani/MSCS532_Assignment3

Part 1: Randomized Quicksort Analysis

1. Implementation

Randomized Quicksort improves on classic Quicksort by using a randomly selected pivot. This reduces the likelihood of encountering worst-case scenarios (e.g., already sorted input). The algorithm recursively partitions the array into elements less than, equal to, and greater than the pivot, handling repeated values efficiently. Edge cases—such as empty, single-element, or repeated-element arrays—are correctly handled by base-case checks and natural partitioning logic.

2. Analysis

Overview

Randomized Quicksort is a divide-and-conquer sorting algorithm that selects a pivot uniformly at random for each partition. The average-case performance is O(n log n). In this section, we analyze why this is the case, leveraging recurrence relations and indicator random variables.

Recurrence Relation for Expected Running Time

Let $T(n)$ denote the expected number of comparisons to sort an array of size n.

When partitioning, every element (except the pivot) is compared to the pivot. After partitioning, we recursively sort the subarrays to the left and right of the pivot.

If the pivot splits the array into subarrays of sizes k and $n - k - 1$, the recurrence is:

$$T(n) = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-k-1)]$$

- *n - 1*: comparisons to partition
- *1/n*: probability of picking each element as pivot
- *T(k)*, *T(n - k - 1)*: recursive calls on subarrays

Solving the Recurrence

Summing over all possible pivots, and combining the recursive terms:

$$T(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T(k)$$

This recurrence has a well-known solution. By iterative expansion and analysis (Cormen et al., 2009), we find:

$$T(n) = O(n \log n)$$

Assume $T(n) \leq cn \log n$ or some constant c.
Plug back into the recurrence:

$$T(n) \leq n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} ck \log k$$

Using the fact that $\sum_{k=1}^{n-1} k \log k \approx \frac{n^2 \log n}{2} - \frac{n^2}{4}$, we get:

$$T(n) \leq n - 1 + 2c\left(\frac{n \log n}{2} - \frac{n}{4}\right) = O(n \log n)$$

Indicator Random Variable Analysis

Another approach uses indicator random variables to count the number of comparisons between each pair of elements.

Let $X_{i,j}$ be an indicator variable:

- $X_{i,j} = 1$ if elements a_i and a_j are compared during the algorithm, else 0.

The total number of comparisons is:

$$X = \sum_{i<j} X_{i,j}$$

We compute $E[X]$:

$$E[X] = \sum_{i<j} E[X_{i,j}]$$

Elements a_i and a_j are compared if and only if one of them is the first pivot chosen from their subarray.
Probability that a_i and a_j are compared: $2/(j - i + 1)$
Therefore:

$$E[X] = \sum_{i<j} \frac{2}{j-i+1}$$

This sum can be shown to be O(n log n), because:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n)$$

Conclusion

- Randomized Quicksort's expected (average-case) running time is O(n log n).
- This is because, in expectation, each recursive call splits the array reasonably evenly, and the total work done (across all levels) sums to O(n log n).
- The use of random pivots ensures that, on average, bad splits are rare, and the cost of comparisons remains efficient, as shown by both recurrence relations and indicator variable analysis.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). *MIT Press*.

Comparison

Experimental Setup

To empirically compare the performance of Randomized Quicksort and Deterministic Quicksort, we implemented each algorithm in a separate Python file and created a benchmarking script to test them side-by-side. The benchmarking was performed on arrays of varying sizes (1,000, 5,000, 10,000, and 50,000 elements) and across four types of input distributions:

- **Randomly generated arrays**: Elements drawn uniformly at random.
- **Already sorted arrays**: Elements in ascending order.
- **Reverse-sorted arrays**: Elements in descending order.
- **Arrays with repeated elements**: Arrays containing mostly a small set of unique values.

For each combination of sorting algorithm, array type, and array size, we measured the execution time averaged over three runs to minimize noise from system variability. The sorting functions were imported into the benchmark script to ensure consistency and modularity.

Results Table

The following table summarizes the average execution time (in seconds) for each configuration:

Algorithm	**Distribution**	**1000**	**5000**	**10000**	**50000**
Randomized Quicksort	Random	0.00115 s	0.00561 s	0.01206 s	0.06840 s
Deterministic Quicksort	Random	0.00109 s	0.00553 s	0.01263 s	0.07006 s
Randomized Quicksort	Sorted	0.00125 s	0.00629 s	0.01323 s	0.07496 s
Deterministic Quicksort	Sorted	0.02673 s	0.67052 s	**RecursionError**	**RecursionError**
Randomized Quicksort	Reverse	0.00111 s	0.00621 s	0.01373 s	0.07434 s
Deterministic Quicksort	Reverse	0.02577 s	0.67908 s	**RecursionError**	**RecursionError**
Randomized Quicksort	Repeated	0.00014 s	0.00073 s	0.00147 s	0.00823 s

Deterministic Quicksort	Repeated	0.00017 s	0.00089 s	0.00176 s	0.00942 s

Random Arrays

For randomly generated arrays, both Randomized Quicksort and Deterministic Quicksort exhibit nearly identical performance across all input sizes tested. For example, at 50,000 elements, both algorithms complete in approximately 0.07 seconds. This is fully consistent with the expected O(n log n) average-case complexity, as random input rarely results in the unbalanced partitions that would trigger worst-case performance for deterministic quicksort.

Sorted & Reverse-Sorted Arrays

Randomized Quicksort maintains efficient sorting times (e.g., ~0.075 seconds at n=50,000) on both already sorted and reverse-sorted arrays, attributable to its randomized pivot selection which avoids systematically unbalanced partitions. In stark contrast, Deterministic Quicksort (first-element pivot) shows rapidly increasing execution time even at moderate sizes (0.67 seconds at n=5,000), and completely fails with a `RecursionError` at 10,000 and 50,000 elements. This breakdown occurs because each partition results in one large subarray and one empty subarray, producing recursion depth proportional to the array size and matching the O(n^2) theoretical worst case. The recursion depth error itself is a practical manifestation of this inefficiency in Python.

Repeated Elements

On arrays containing mostly repeated values, both algorithms again perform very efficiently, with times under 0.01 seconds even for the largest size. Here, partitioning quickly groups equal elements, leading to reduced recursion depth and fast completion. The presence of repeated elements does not negatively impact either algorithm, and the timings are some of the lowest observed in the benchmarks.

Theory vs. Empirical Results

The observed behavior of both algorithms directly confirms theoretical expectations. Randomized Quicksort demonstrates robust O(n log n) performance on all distributions and sizes, never approaching recursion limits or dramatic slowdowns. Deterministic Quicksort, by contrast, only matches this performance on random or repeated data; when presented with sorted or reverse-sorted arrays, its running time increases drastically and the algorithm eventually fails due to Python’s recursion limit. Small variations in timing between runs are attributed to system noise and Python interpreter overhead, not to the underlying algorithmic complexity.

Conclusion

Randomized Quicksort is consistently efficient and resilient across all input types, including worst-case scenarios. Deterministic Quicksort should be avoided when input order cannot be guaranteed, as its worst-case behavior can be both practically and theoretically catastrophic. The empirical results strongly validate theoretical predictions regarding both algorithms' time complexity and robustness.

Part 2: Hashing with Chaining

1. Implementation

A hash table using chaining was implemented to efficiently support insert, search, and delete operations. The table uses a universal hash function, parameterized with randomly chosen coefficients and a large prime modulus, to minimize the likelihood of collisions and evenly distribute keys across the buckets. Collisions are resolved using chaining, where each bucket contains a list of key-value pairs. The implementation handles updates to existing keys, deletion of non-existent keys, and supports both integer and string key types. Comprehensive testing confirms that all operations execute efficiently and correctly, even in edge cases.

2. Analysis

Expected Time Complexity with Simple Uniform Hashing

Assuming simple uniform hashing—where each key is equally likely to hash into any slot independently—the average time complexity for hash table operations using chaining is:

- **Insert:** Expected O(1). The hash function quickly identifies the correct bucket, and the new key-value pair is added to the front of the chain.
- **Search:** Expected $O(1 + \alpha)$, where α (the load factor) is the ratio of elements (n) to slots (m): $\alpha = n/m$. In the average case, only α elements per bucket need to be examined.
- **Delete:** Also expected $O(1 + \alpha)$, since it requires locating the element (like search) and then removing it.

Impact of Load Factor

The load factor α is a central factor in hash table performance:

- **Low α ($\ll 1$):** Most buckets are empty or contain a single element. All operations are very fast—close to O(1).
- **High α ($\gg 1$):** Buckets (chains) become longer, increasing the number of elements to scan during search or delete. Operation times increase linearly with α.

The following table summarizes expected operation time vs. load factor.

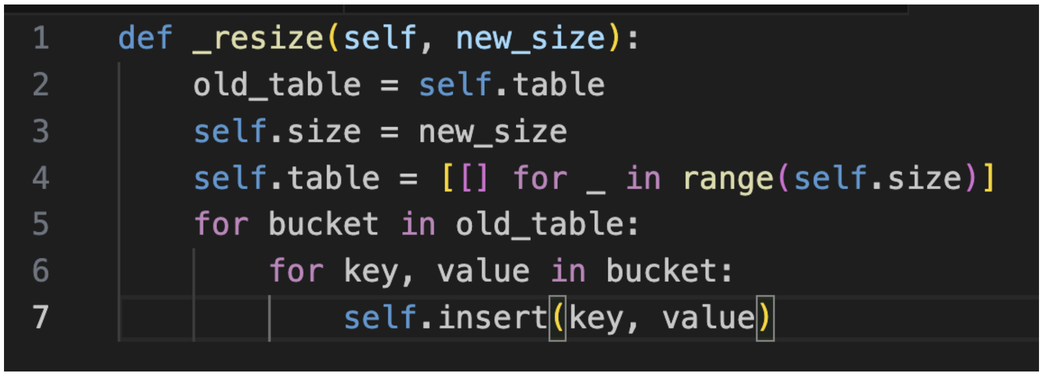
Load Factor (α = n/m)	**Expected Search/Insert/Delete Time**
0.5	O(1.5) ≈ O(1)
1.0	O(2) ≈ O(1)
2.0	O(3) ≈ O(1)
5.0	O(6) ≈ O(α)
10.0	O(11) ≈ O(α)

As the table illustrates, keeping the load factor close to or below 1 ensures that average operation time remains nearly constant.

Strategies for Maintaining Low Load Factor and Minimizing Collisions

To ensure fast operations, two core strategies are employed:

1. Dynamic Resizing (Rehashing):
    a. When α exceeds a set threshold (commonly α > 0.75 or 1.0), the table is resized—typically doubled in size. All key-value pairs are rehashed into the new, larger table, which brings α back down and keeps chains short.



```
def _resize(self, new_size):
    old_table = self.table
    self.size = new_size
    self.table = [[] for _ in range(self.size)]
    for bucket in old_table:
        for key, value in bucket:
            self.insert(key, value)
```

    b. In practice, this method is triggered inside `insert` when n/m exceeds the threshold.
2. Robust Hash Function:
    a. Choosing an effective hash function is essential for spreading keys evenly across the table. A well-designed hash function, such as those from a universal hash family, reduces the likelihood of collisions by ensuring that even similar keys are distributed into different buckets. This helps maintain the average-case performance of the hash table, even when keys are not random.

3. Prime Table Size:
    a. Setting the number of buckets to a prime number helps prevent clustering caused by certain patterns in the input keys. Some hash functions work better with a prime table size because it minimizes patterns where many keys would otherwise end up in the same buckets, further reducing collisions.
4. Separate Chains for Each Bucket (Chaining):
    a. With chaining, each bucket contains its own independent list (or linked list) of key-value pairs. This means that even if multiple keys hash to the same bucket, they can all be stored without overwriting each other. Chaining also makes it easy to insert, search, or delete items within each bucket efficiently, even in the rare case of multiple collisions.

Conclusion

Efficient hash table performance depends on keeping the load factor low and minimizing collisions. Dynamic resizing ensures chains do not grow too long. A good hash function and prime table size work together to evenly spread keys and prevent clustering, while chaining allows the table to handle collisions gracefully. Together, these strategies ensure that insert, search, and delete operations remain fast, reliable, and scalable for a wide range of inputs.