Assignment 3

MSCS-532-M80

Bibek Itani

Oct 4, 2025

Dr. Satish Penmatsa

**Part 1**

**Implement the Randomized Quicksort algorithm where the pivot element is chosen uniformly at random from the subarray being partitioned.**

In this assignment, I implemented both Randomized Quicksort and Deterministic Quicksort (using the first element as the pivot). To handle recursion limitations in Python, I used iterative versions of both algorithms. The randomized pivot in Quicksort is selected uniformly from the subarray being partitioned, ensuring a fair distribution of partitions across different inputs.

**Time Complexity Analysis**  
  
The average case time complexity of Randomized Quicksort is:  
  
O(n log n )

This is achieved due to the high probability that the pivot divides the array relatively evenly over recursive steps. Using indicator random variables and expected value analysis, we can show that the number of comparisons follows a logarithmic growth pattern.

Let T(n) be the expected time to sort n elements:

T(n) = T(k) + T(n-k-1) + O(n), where k is the number of elements less that the pivot. By averaging over all k, the recurrence resolves to O(n log n ) in expectation.

The worst-case complexity remains O( n^2 ), but the use of a random pivot makes the outcome highly unlikely.

Comparison with Deterministic Quicksort

I ran both algorithms on four types of arrays:

* Randomly generated
* Already sorted
* Reverse sorted
* Repeated elements

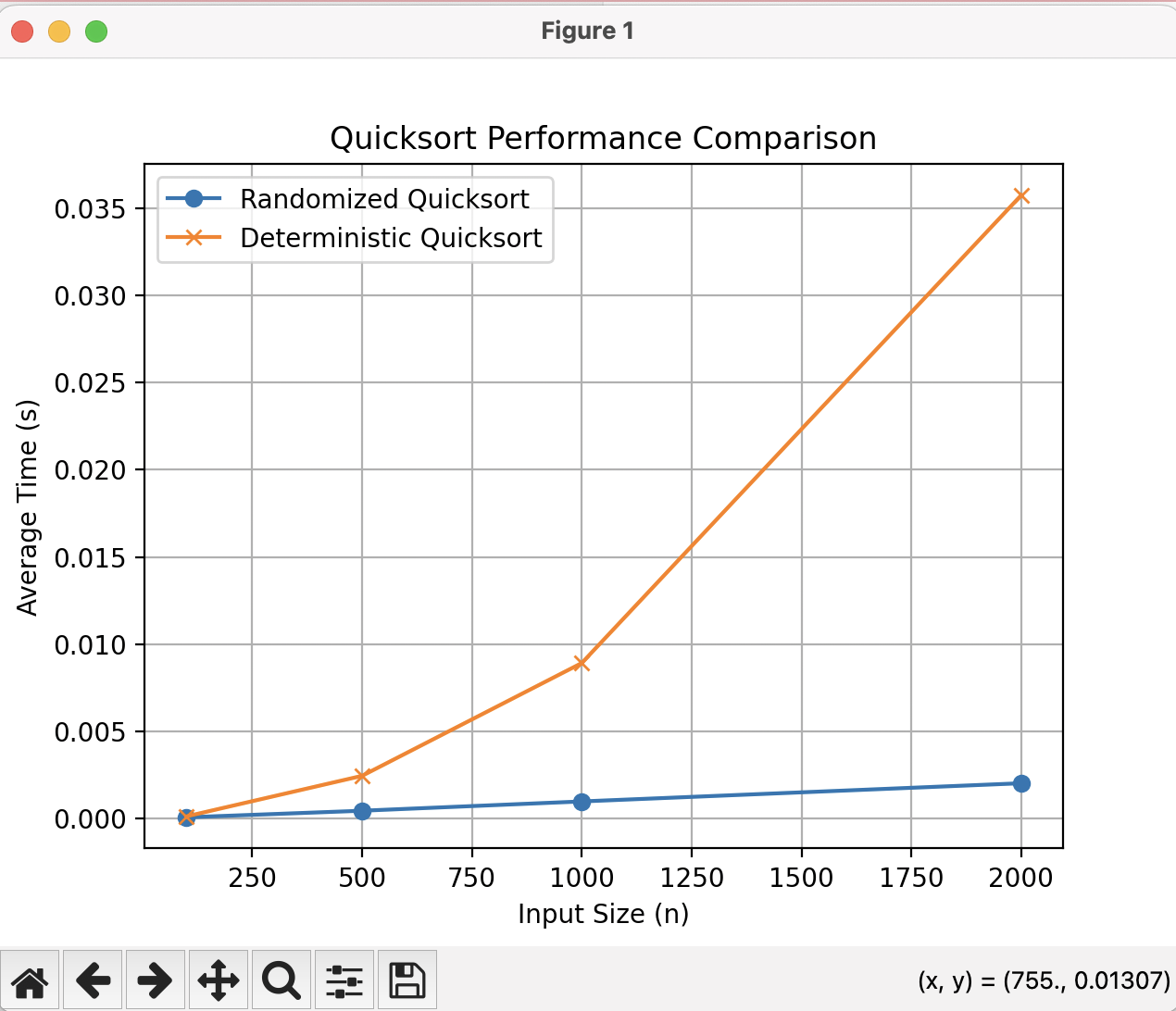
Input Sizes Tested:

* 100
* 500
* 1000
* 2000 elements

Results Summary:

On random arrays, both algorithms performed similarly, though Randomized Quicksort had slightly better average times. On sorted and reverse-sorted arrays, Deterministic Quicksort showed significantly worse performance due to poor pivot choice. On arrays with repeated elements, performance was nearly the same, though both algorithms experienced slight slowdowns due to reduced partitioning efficiency.

Graph:



The experimental data mostly matched theoretical expectations. Randomized quicksort consistently performed more stably across all input types. The deterministic version showed inefficiency in edge cases like sorted or reversed arrays, where the pivot divides the array poorly. Any discrepancies in time were small but consistent. Minor variations in Python’s memory management and system load during execution likely explain small timing differences across repeated runs.

Part 2:  
  
Implementation

A hash table is a data structure that stores key–value pairs and allows efficient insertion, search, and deletion operations.However, when two keys map to the same hash index (a collision), we need a collision resolution strategy. In this implementation, we use chaining, where each table slot stores a linked list (or dynamic list) of all key-value pairs that hash to the same index.

Implementation Details

1. Structure

Hash Table: An array (or list) of size m, where each element is a list (or linked list) of key–value pairs.

Hash Function: A function h(k) that maps a key k to an index between 0 and m−1.

1. Hash Function Choice

A universal hash function helps minimize collisions even under adversarial input. A simple version from the universal hashing family can be defined as:

h(k) = ((a ⨉ k + b) mod p) mod m

Where:

* p is a large prime number greater than the maximum key value,
* a and b are random integers such that 1 ≤ a < p and 0 ≤ b < p,
* m is the number of buckets in the hash table.

This ensures a near-uniform distribution of keys across buckets.

1. Operations

Insert(key, value)

* Compute the hash index: index = h(key)
* Access the linked list at that index.
* If key already exists → update its value.
* Else → append (key, value) to the list.

Time complexity (expected): O(1 + ⍺)

Search(key)

1. Compute index = h(key)
2. Traverse the linked list at that index.
3. If key is found → return value, else return null.

Time complexity (expected): O(1 + ⍺)

Delete(key)

1. Compute index = h(key)
2. Traverse the linked list to find the key.
3. Remove the corresponding (key, value) node.

Time complexity (expected): O(1 + ⍺)

2. Analysis

Assumption: Simple Uniform Hashing

Under simple uniform hashing, every key is equally likely to be hashed into any of the m slots, independently of other keys.

Let:

* n = number of keys stored,
* m = number of slots (buckets),
* Load factor \alpha = \frac{n}{m}.

This represents the average number of elements per bucket.

Expected Time Complexities

|  |  |  |
| --- | --- | --- |
| Operation | Expected Time | Explanation |
| Insert | O(1 + ⍺) | Hash computation takes O(1); insertion into linked list takes expected O(1) if α is small. |
| Search | O(1 + ⍺) | Expected number of elements per chain is α; so average chain length = α. |
| Delete | O(1 + ⍺) | Same reasoning as search. |

When ⍺ = O(1) (i.e., number of keys n is proportional to the number of buckets m), all operations run in expected constant time.

Effect of Load Factor

* Low α (α ≈ 1 or less):
  + - Most chains are short.
    - Search, insert, and delete operations are near constant time.
* High α (α ≫ 1):
  + - Chains become long.
    - Average lookup time increases linearly with α.
    - Degrades performance toward O(n) in the worst case.

3. Strategies to Maintain Low Load Factor and Minimize Collisions

a) Dynamic Resizing

When the load factor α exceeds a threshold (commonly 0.75 or 1.0):

1. Double the table size (m ← 2m),
2. Rehash all existing keys using the new hash function and table size.

This keeps α bounded and maintains expected O(1) performance. Similarly, if α becomes very low (after many deletions), the table size can be halved.

b) Good Hash Function Design

* Use a universal hash function family to ensure near-uniform key distribution.
* Avoid simple modular hash functions like h(k) = k % m for patterned keys.
* Choose m as a prime number to reduce clustering.

c) Separate Chaining with Linked Lists or Balanced Trees

* Linked lists: simple and efficient for small α.
* For large α, use balanced BSTs (like red-black trees) for each chain to reduce worst-case search to O(\log n).

Conclusion:

Using chaining with a well-chosen universal hash function, the hash table provides:

* Expected constant-time operations when the load factor is low,
* Predictable performance under uniform hashing,
* Scalability through dynamic resizing.

Maintaining a low load factor and minimizing collisions are crucial to sustaining high efficiency.