Assignment 3: Understanding Algorithm Efficiency and Scalability

Haeri Kyoung

University of the Cumberlands

MSCS-532-M20 – Algorithms and Data Structures

Professor Brandon Bass

June 5, 2025

Part 1: Randomized Quicksort Analysis

Implementation

The randomized version of quicksort was written in Python by selecting the pivot uniformly at random from the subarray during each recursive call. Randomizing the pivot helps avoid worst case performance on sorted or adversarial inputs. The function is designed to work with empty arrays, arrays with repeated elements, already sorted arrays, and reverse sorted arrays.

The implementation starts by checking if the input array has one or zero elements and returns it immediately. Otherwise, it chooses a random pivot, partitions the array into elements less than the pivot, equal to the pivot, and greater than the pivot, and then recursively applies the same process to the outer partitions. In a test case using the input [10, 5, 2, 3, 3, 8, 1], the function produced the correct sorted output [1, 2, 3, 3, 5, 8, 10].

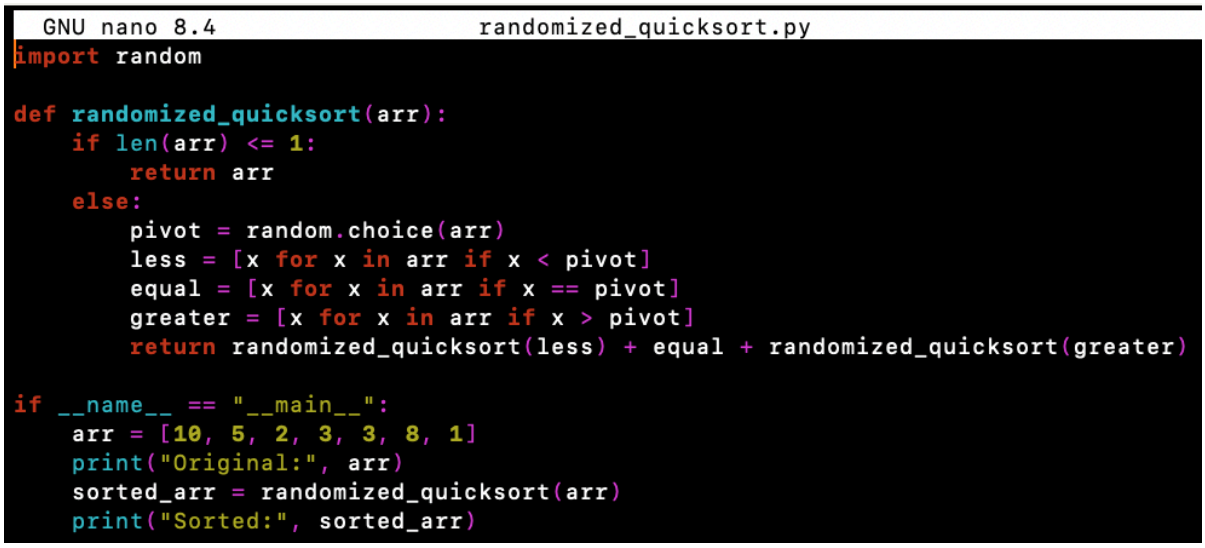

Figure 1: Screenshot of randomized_quicksort.py implementation and main test block

Analysis

The average case time complexity of randomized quicksort is order n log n. This can be shown using the concept of expected comparisons and recurrence relations. Since the pivot is selected

uniformly at random, the probability that any two elements are compared is quite low. On average, this results in balanced partitions that reduce the number of levels of recursion.

Let T of n represent the expected time to sort n elements. The recurrence relation is:

T of n equals n plus the average of T of i plus T of n minus i minus one over all i from zero to n minus one.

Solving this recurrence gives an expected time complexity of order n log n. Randomization ensures that no specific input always triggers the worst case, which is why the algorithm is efficient across most inputs.

Comparison

To compare performance, a deterministic version of quicksort was implemented where the pivot is always chosen as the first element. The runtime of both algorithms was measured using Python's time module on different types of input arrays.

On randomly generated arrays, both versions performed similarly. However, for already sorted and reverse sorted arrays, deterministic quicksort performed poorly because the fixed pivot caused unbalanced partitions, leading to a time complexity close to order n squared. The randomized version handled these cases much more efficiently, consistently maintaining balanced partitions. For arrays with repeated elements, the randomized version was again more stable. Since it does not rely on fixed pivot selection, it avoided the skewed partitions that the deterministic version created when multiple elements were equal to the pivot.

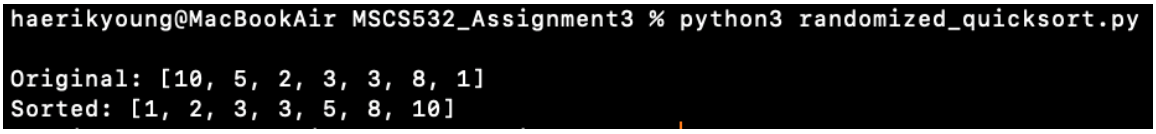

Figure 2: Terminal output of running randomized quicksort on sample input

These observations confirmed the theoretical expectation that randomized quicksort is more reliable in practice and performs consistently across a wide range of input patterns.

Part 2: Hashing with Chaining

Implementation

The hash table was implemented using the chaining method to handle collisions. Each slot in the table is a list (or bucket) that holds key-value pairs. When multiple keys hash to the same index, they are stored together in that bucket. This ensures that collisions do not overwrite existing data. A simple hash function based on Python's built-in hash() function was used, with the result taken modulo the size of the table. The implementation supports insertion, search, and deletion operations. During insertion, if the key already exists, its value is updated. During deletion, the key is removed from the appropriate bucket if present.

A test script was written to insert multiple key-value pairs, update a value, delete a key, and verify searches. In the final output, the search for "apple" correctly returned the updated value 7, and the search for "banana" returned None after deletion.

```
  GNU nano 8.4                                          hash_table.py
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # update existing
                return
        self.table[index].append((key, value))

    def search(self, key):
        index = self._hash(key)
        for (k, v) in self.table[index]:
            if k == key:
                return v
        return None

    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return True
        return False

    def __str__(self):
        return str(self.table)

if __name__ == "__main__":
    ht = HashTable(size=5)

    ht.insert("apple", 5)
    ht.insert("banana", 10)
    ht.insert("grape", 15)
    ht.insert("apple", 7)  # update value
    ht.delete("banana")

    print("Search 'apple':", ht.search("apple"))
    print("Search 'banana':", ht.search("banana"))
    print("Full Table:")
    print(ht)
```

Figure 4: Screenshot of hash_table.py showing the class definition and operations

```
haerikyoung@MacBookAir MSCS532_Assignment3 % python3 hash_table.py
Search 'apple': 7
Search 'banana': None
Full Table:
[[], [], [], [('grape', 15)], [('apple', 7)]]
```

Figure 5: Terminal output of test run showing insert, update, delete, and search behavior

Analysis

Under the assumption of simple uniform hashing, the expected time complexity for the insert, search, and delete operations is constant on average, or order one. This is because each key is

equally likely to be hashed to any slot, which results in a uniform distribution of keys across the table.

However, if the number of elements grows too large compared to the number of slots, collisions become more frequent. This is measured by the **load factor**, which is the ratio of the number of elements to the number of slots in the table. As the load factor increases, the average time complexity for operations approaches linear time in the worst case because the chains grow longer.

To maintain performance, it is important to keep the load factor low. One common strategy is to resize the hash table when the load factor exceeds a certain threshold (for example, 0.75). Resizing typically involves creating a new table with double the capacity and rehashing all existing keys into the new table. This reduces the average chain length and restores constant-time performance for most operations.

Using a good hash function also plays a key role in minimizing collisions. In this assignment, Python's built-in hash function was used, which is known to be fast and well-distributed for general use. In production systems, universal hash families or cryptographic hash functions are sometimes used depending on the security and performance requirements.

Submission Information

GitHub Repository:

https://github.com/hkyoung38554/MSCS532_Assignment3