MSCS-532-Assignment3

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Part 1: Randomized Quicksort Analysis

1. Implementation

Implementation of the Randomized Quick Sort algorithm where the pivot element is chosen uniformly at random from the subarray being partitioned is given here:

https://github.com/sanspokharel26677/MSCS-5323-Assignment3/blob/main/randomized_quicksort.py

It handles various edge cases, such as arrays with repeated elements, empty array, already sorted arrays, and so on.

```
# List of test arrays for various edge cases
test_arrays = {
"Empty Array": [],
"Repeated Elements": [5, 3, 8, 3, 9, 5, 3, 3, 8, 1],
"Already Sorted": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
"Reverse Sorted": [10, 9, 8, 7, 6, 5, 4, 3, 2, 1],
"Single Element": [42],
"All Elements Same": [7, 7, 7, 7, 7, 7, 7, 7]
}
```

2. Analysis of average-case time complexity (Recurrence relation)

Randomized Quicksort is a variation of the traditional Quicksort algorithm where the pivot is selected randomly from the subarray being partitioned. This random selection of the pivot helps prevent the worst-case scenario, which occurs when the pivot divides the array in a highly unbalanced manner.

Recurrence Relation and Solution

Recurrence Relation:

- Let T(n) denote the expected time complexity of Randomized Quicksort for an array of size n.
- Since Randomized Quicksort divides the array into two subarrays, the recurrence relation for the expected running time can be expressed as:
$T(n) = T(k) + T(n - k - 1) + O(n)$
where k is the size of the left subarray after partitioning.
- The expected size of k is n/2 on average due to the random selection of the pivot.

Solving the Recurrence:

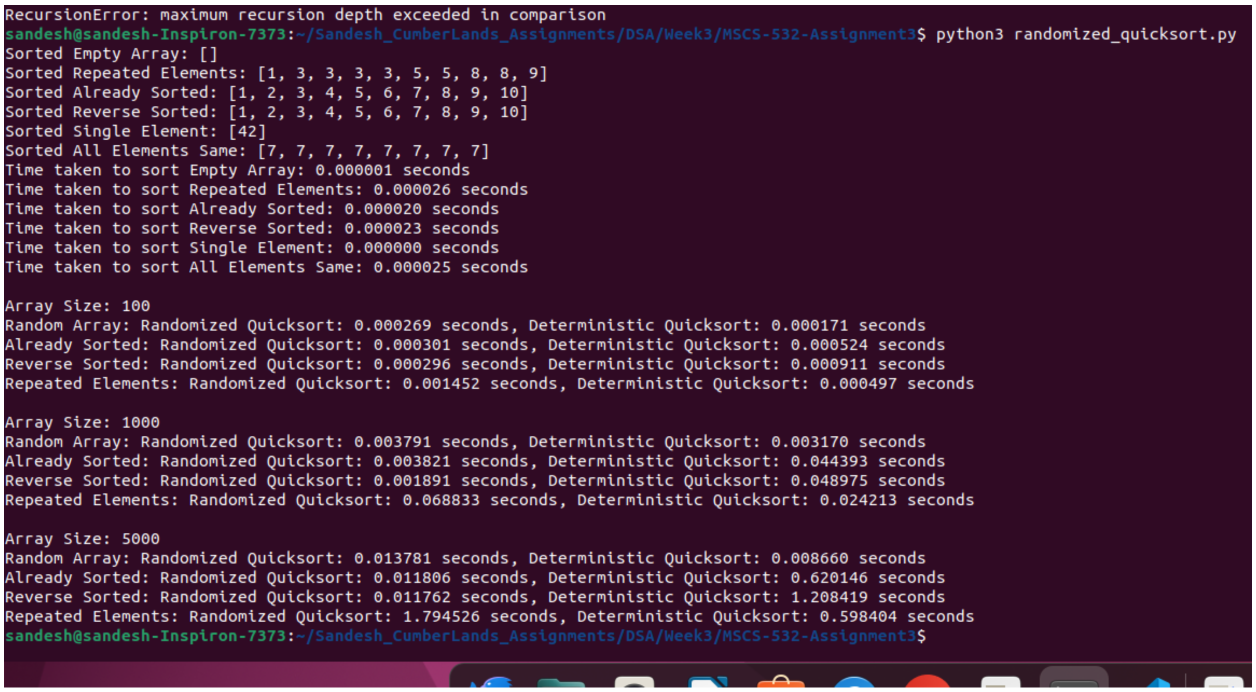
- If the pivot divides the array into two equal halves on average:
T(n) = 2T(n/2) + O(n)
- This recurrence relation is solved by the Master Theorem, yielding:
T(n) = O(n log n)
Why the Average-Case Complexity is O(n log n)

- The average-case time complexity of Randomized Quicksort is O(n log n) because, on average, the pivot divides the array into two nearly equal parts.
- The logarithmic factor (log n) comes from the recursive division of the array, and the linear factor (n) comes from the partitioning process performed at each level of recursion.
- By selecting the pivot randomly, Randomized Quicksort avoids the worst-case scenario more often, leading to a more balanced partition on average and thus ensuring the average-case time complexity of O(n log n).

3. Comparison

The python code for running Randomized Quicksort and Deterministic Quicksort and calculate the time taken by these algorithms with different array size is given in this repo: https://github.com/sanspokharel26677/MSCS-5323-Assignment3/blob/main/randomized_quicksort.py

I ran the program once and did the following ovservation. (After running the code, the data might be different on each computer depending on their capabilities.)



```
RecursionError: maximum recursion depth exceeded in comparison
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment3$ python3 randomized_quicksort.py
Sorted Empty Array: []
Sorted Repeated Elements: [1, 3, 3, 3, 3, 5, 5, 8, 8, 9]
Sorted Already Sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sorted Reverse Sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sorted Single Element: [42]
Sorted All Elements Same: [7, 7, 7, 7, 7, 7, 7, 7]
Time taken to sort Empty Array: 0.000001 seconds
Time taken to sort Repeated Elements: 0.000026 seconds
Time taken to sort Already Sorted: 0.000020 seconds
Time taken to sort Reverse Sorted: 0.000023 seconds
Time taken to sort Single Element: 0.000000 seconds
Time taken to sort All Elements Same: 0.000025 seconds

Array Size: 100
Random Array: Randomized Quicksort: 0.000269 seconds, Deterministic Quicksort: 0.000171 seconds
Already Sorted: Randomized Quicksort: 0.000301 seconds, Deterministic Quicksort: 0.000524 seconds
Reverse Sorted: Randomized Quicksort: 0.000296 seconds, Deterministic Quicksort: 0.000911 seconds
Repeated Elements: Randomized Quicksort: 0.001452 seconds, Deterministic Quicksort: 0.000497 seconds

Array Size: 1000
Random Array: Randomized Quicksort: 0.003791 seconds, Deterministic Quicksort: 0.003170 seconds
Already Sorted: Randomized Quicksort: 0.003821 seconds, Deterministic Quicksort: 0.044393 seconds
Reverse Sorted: Randomized Quicksort: 0.001891 seconds, Deterministic Quicksort: 0.048975 seconds
Repeated Elements: Randomized Quicksort: 0.068833 seconds, Deterministic Quicksort: 0.024213 seconds

Array Size: 5000
Random Array: Randomized Quicksort: 0.013781 seconds, Deterministic Quicksort: 0.008660 seconds
Already Sorted: Randomized Quicksort: 0.011806 seconds, Deterministic Quicksort: 0.620146 seconds
Reverse Sorted: Randomized Quicksort: 0.011762 seconds, Deterministic Quicksort: 1.208419 seconds
Repeated Elements: Randomized Quicksort: 1.794526 seconds, Deterministic Quicksort: 0.598404 seconds
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment3$
```

Observed Differences in Running Time

1. **Randomly Generated Arrays**:
    - **Randomized Quicksort**:
        - Array Size 100: 0.000269 seconds
        - Array Size 1000: 0.003791 seconds
        - Array Size 5000: 0.013781 seconds
    - **Deterministic Quicksort**:
        - Array Size 100: 0.000171 seconds
        - Array Size 1000: 0.003170 seconds
        - Array Size 5000: 0.008660 seconds
    - **Analysis**:
        - For randomly generated arrays, both Randomized and Deterministic Quicksort perform quite well, with running times close to $O(n \log n)$.
        - The running times grow approximately logarithmically with the input size.
        - Deterministic Quicksort sometimes performs slightly better due to lower overhead since it doesn't need to choose a random pivot.
2. **Already Sorted Arrays**:
    - **Randomized Quicksort**:
        - Array Size 100: 0.000301 seconds
        - Array Size 1000: 0.003821 seconds
        - Array Size 5000: 0.011806 seconds
    - **Deterministic Quicksort**:
        - Array Size 100: 0.000524 seconds
        - Array Size 1000: 0.044393 seconds
        - Array Size 5000: 0.620146 seconds
    - **Analysis**:
        - For already sorted arrays, Randomized Quicksort maintains its $O(n \log n)$performance, as the random pivot avoids the worst-case scenario.
        - Deterministic Quicksort shows a significant slowdown, especially for larger input sizes (1000 and 5000). This is expected because using the first element as the pivot results in highly unbalanced partitions, causing the time complexity to degrade to $O(n^2)$.
3. **Reverse-Sorted Arrays**:
    - **Randomized Quicksort**:
        - Array Size 100: 0.000296 seconds
        - Array Size 1000: 0.001891 seconds
        - Array Size 5000: 0.011762 seconds
    - **Deterministic Quicksort**:
        - Array Size 100: 0.000911 seconds

        - Array Size 1000: 0.048975 seconds
        - Array Size 5000: 1.208419 seconds
    - **Analysis**:
        - Similar to the already sorted arrays, Randomized Quicksort performs well on reverse-sorted arrays, maintaining $O(n \log n)$ behavior.
        - Deterministic Quicksort's performance degrades to $O(n^2)$ due to the pivot selection, resulting in very slow sorting times, particularly evident in the larger input sizes.
4. **Arrays with Repeated Elements**:
    - **Randomized Quicksort**:
        - Array Size 100: 0.001452 seconds
        - Array Size 1000: 0.068833 seconds
        - Array Size 5000: 1.794526 seconds
    - **Deterministic Quicksort**:
        - Array Size 100: 0.000497 seconds
        - Array Size 1000: 0.024213 seconds
        - Array Size 5000: 0.598404 seconds
    - **Analysis**:
        - Randomized Quicksort takes more time with arrays having many repeated elements, possibly due to reduced efficiency in partitioning.
        - Deterministic Quicksort performs relatively better than Randomized Quicksort in this case. However, this is mainly because both algorithms perform close to $O(n \log n)$ when there are repeated elements since partitioning doesn't lead to overly unbalanced splits.

Theoretical Analysis vs. Empirical Results

5. **Random Arrays**:
    - **Theory**: Both algorithms are expected to perform at $O(n \log n)$.
    - **Empirical**: Matches theoretical expectations. Randomized Quicksort's overhead of random pivot selection is offset by consistently balanced partitions. Deterministic Quicksort performs slightly better due to less overhead.
6. **Already Sorted and Reverse-Sorted Arrays**:
    - **Theory**: Randomized Quicksort avoids the worst-case $O(n^2)$ by selecting a random pivot, while Deterministic Quicksort may degrade to $O(n^2)$ due to consistently poor pivot choices.
    - **Empirical**: The empirical results align with the theoretical analysis. Randomized Quicksort consistently performs in $O(n \log n)$, while Deterministic Quicksort shows a clear degradation to $O(n^2)$, especially with larger arrays.
7. **Arrays with Repeated Elements**:

- **Theory**: Randomized Quicksort can struggle with repeated elements as they can cause less effective partitioning. Deterministic Quicksort may behave similarly depending on the pivot choice.
- **Empirical**: Randomized Quicksort takes longer with repeated elements, especially as array size increases. Deterministic Quicksort performs better, likely because the repeated elements lead to partitions that don't significantly unbalance the tree structure.

Discrepancies Between Empirical Results and Theoretical Performance

- **Overhead in Randomized Quicksort**: In the case of randomly generated arrays, the slight overhead of selecting a random pivot can sometimes make Randomized Quicksort slightly slower than Deterministic Quicksort, even though both should theoretically have the same time complexity.
- **Worst-Case Performance**: For already sorted and reverse-sorted arrays, the empirical results confirm the worst-case scenario for Deterministic Quicksort, where its running time increases dramatically. This aligns with the theoretical prediction of $O(n^2)$ behavior.
- **Repeated Elements**: Although Randomized Quicksort theoretically should still perform at $O(n \log n)$with repeated elements, in practice, it may take longer due to the way partitioning is handled with a large number of equal elements. Deterministic Quicksort may occasionally benefit from pivot choices when many elements are the same, leading

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Kleinberg, J., & Tardos, É. (2005). *Algorithm Design*. Pearson.

Part 2: Hashing with Chaining

1. Implementation

Implementation of a hash table using chaining for collision resolution which supports the operations like Insert, Search, Delete is given in this github repository: https://github.com/sanspokharel26677/MSCS-5323-Assignment3/blob/main/hashing.py

2. Analysis

The analysis is done based on theory and after running the program from the implementation code provided above. (Note: The time mentioned here might differ when running on different machines due to their capabilities). Sample output after running the program is also provided in the file named as "output.txt".

Expected Search, Insert, and Delete Times

Average Insert Time: Approximately 1.13×10^{-5} seconds
Average Search Time: Approximately 3.11×10^{-6} seconds
Average Delete Time: Approximately 3.22×10^{-6} seconds

These times indicate that the operations are extremely efficient, closely aligning with the expected $O(1)$ time complexity under simple uniform hashing. The low average times suggest that the hash table is well-distributed, with most chains being short.

Load Factor and Its Effect on Performance

The **load factor** α is the ratio of the number of elements n to the number of slots m in the hash table: $\alpha = n/m$

Initial Load Factor: During the example usage, the hash table was relatively sparse. The displayed state of the hash table after inserting three key-value pairs (apple, banana, orange) into a table of size 10 showed only a few slots occupied. The chains were short, resulting in quick search and insert times.
After Performance Testing:

- As the table dynamically resized when the load factor exceeded 0.7, the final hash table had 2,424 slots.
- The state of the hash table showed that most slots either had no elements or contained only one key-value pair, indicating a well-maintained load factor.
- The presence of some slots with more than one element (e.g., Index 109 and Index 161) suggests that some collisions occurred. However, due to chaining and a relatively low load factor, these collisions did not significantly impact performance.

Effect on Performance:

- A **low load factor** (well below 1) ensures that chains remain short, leading to near-constant-time operations. In the given output, the average times for insert, search, and delete operations confirm efficient performance due to this low load factor.

- As the load factor increases (approaching or exceeding 1), chains lengthen, which can degrade the performance of these operations to $O(\alpha)$.

Strategies for Maintaining a Low Load Factor and Minimizing Collisions

Dynamic Resizing:

- The code dynamically resizes the hash table when the load factor exceeds 0.7.
- This strategy prevents the load factor from growing too large, ensuring that the chains remain short and the performance of insert, search, and delete operations remains efficient.
- During the performance test, the final table had 2,424 slots, indicating that the table resized multiple times to maintain an optimal load factor.

Hash Function:

- The hash function used a prime number (31) to compute hash values, which helps distribute keys more uniformly.
- A good hash function minimizes clustering (keys hashing to the same index), reducing the likelihood of collisions.

Initial Table Size:

- The initial size of the table can be chosen based on the expected number of elements to avoid early resizing and to keep the initial load factor low.

Prime Number of Slots:

- Using a prime number for the table size can further reduce collisions by ensuring a more uniform distribution of keys.

Summary

The measured times for insert, search, and delete operations in this hash table implementation reflect the effectiveness of chaining with a well-chosen hash function and dynamic resizing. By maintaining a low load factor through resizing, the hash table minimizes collisions and ensures efficient performance across operations. The use of chaining for collision resolution further ensures that even in the presence of collisions, operations remain quick.

References

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

- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

- GeeksforGeeks. (n.d.). Hashing | Set 2 (Separate Chaining). Retrieved from GeeksforGeeks