Name: Sandhyarani Mudindi

Student ID: 005034219

Course Title: Algorithms and Data Structures (MSCS-532-A01)

Assignment number: 5

Table of Contents

Abstract

Quicksort is a highly efficient, divide-and-conquer sorting algorithm that recursively partitions an array around a selected pivot. This report analyzes the time and space complexity of Quicksort under various conditions, including best-case (O (n log n)), average-case (O(n log n)), and worst-case ($O(n^2)$) scenarios. The worst-case inefficiency arises when the pivot selection leads to highly unbalanced partitions. To mitigate this, randomized Quicksort selects pivots randomly, ensuring more balanced partitions on average and reducing the likelihood of worst-case behavior. Empirical analysis confirms that deterministic Quicksort performs well on random inputs but degrades to $O(n^2)$ on sorted and reverse-sorted inputs. In contrast, randomized Quicksort maintains consistent O (n log n) performance across all input patterns. The study highlights the impact of pivot selection, recursion overhead, and space complexity, demonstrating the advantages of randomized pivoting in achieving predictable and stable sorting efficiency.

Quick Sort:

Quicksort is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Time Complexity of Quick Sort:

The time complexity of Quicksort depends on how well the array is partitioned during each recursive step. Below is a detailed analysis of the best, average, and worst-case scenarios:

1. Best Case:
    - The pivot splits the array into two equal halves at every recursive step.
    - At each recursive level, the array is divided into two subarrays of approximately half the size.
    - At each level, the partitioning process involves
    - O(n) operations to compare elements and create the two subarrays.

    Time Complexity:

    - At level 1: n comparisons
    - At level 2: $\frac{n}{2} + \frac{n}{2}$ = n comparisons
    - At level k: n comparisons

    Thus, the total time complexity is:

    $\mathrm{T(n) = n.\,nlog_2}(n)$ = O (n log n)

2. Average Case:
    - The pivot splits the array into two subarrays of roughly unequal sizes, but neither is excessively small (e.g., 25% and 75%).
    - On average, the partitioning results in subarrays proportional to the input size.

- The number of recursive levels remains around log(n), though some subarrays may be slightly larger than others.
- Each level still requires O(n) comparisons for partitioning.

Time Complexity:

- The average case also exhibits O (n log n) time complexity. This is because, on average, the pivot will divide the array reasonably well, even if not perfectly in half every time.
- Mathematical analysis shows that even with some imbalance in the partitions, the average performance remains O (n log n).

3. Worst Case:
   - The worst case occurs when the pivot consistently happens to be the smallest or largest element in the array. This leads to highly unbalanced partitions.
   - In this scenario, one sub-array will have n-1 elements, and the other will be empty.
   - This results in a recursion tree with a depth of 'n', and each level of the recursion still takes O(n) time for partitioning.

Time Complexity:

- At level 1: n comparisons
- At level 2: n−1 comparisons and so on...
- At level n: 1 comparison

Thus, the total time complexity is:

$$\mathrm{T(n)} = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2} = O(n^2)$$

Space Complexity of Quicksort

Quicksort is generally considered an in-place sorting algorithm because it doesn't require significant extra storage space. However, it does use some auxiliary space for the recursion stack.

- Best and Average Case: O (log n).
    - In the best and average cases, where the pivot divides the array into roughly equal halves, the depth of the recursion tree is O (log n).
    - Each recursive call adds a frame to the call stack, so the space complexity due to the recursion stack is O (log n).
- Worst Case: O(n)
    - In the worst case, where the pivot consistently results in highly unbalanced partitions, the recursion depth can reach O(n).
    - This leads to a call stack with O(n) frames, resulting in O(n) space complexity.
- Additional Storage for Partitioning
    - **In-Place Partitioning**: If Quicksort is implemented in place (i.e., elements are rearranged within the original array), no additional space is needed for partitioning.
    - Space Usage: O (1)
    - **Not In-Place Partitioning**: In some implementations, auxiliary arrays are used to hold elements smaller and larger than the pivot during partitioning.
    - This requires O(n) additional space in the worst case.
    - Space Usage: O(n)

Additional Overheads Associated with Quicksort:

- Partitioning Overhead: The partitioning process itself involves some overhead. It requires comparisons and swaps to rearrange elements around the pivot. However, this overhead is generally considered to be constant for each element (O (1)) within each partitioning step, contributing to the overall O(n) time complexity of each partitioning stage.
- Recursion Overhead: The recursive nature of Quicksort introduces some overhead due to function calls and returns. Each function call involves pushing a new frame onto the call stack, which stores local variables, parameters, and return addresses. While this overhead is typically small, it can become significant in cases with very deep recursion (worst-case scenarios).
- Pivot Selection Overhead: Different pivot selection strategies have varying overheads.
    - Choosing the first or last element has minimal overhead (O (1)).
    - Choosing a random element requires generating a random number, which has a small overhead.
    - The median-of-three method involves comparing three elements, adding a small constant overhead.

Mitigating Overheads:

- Tail Call Optimization: Some compilers and programming languages support tail call optimization, which can eliminate the overhead of recursive calls in certain situations. However, this is not universally available.
- Looping Instead of Recursion: In some cases, it's possible to rewrite Quicksort using loops instead of recursion to avoid the call stack overhead. However, this can make the code more complex.

Randomized Quick Sort:

Randomization addresses this issue by introducing randomness into the pivot selection process. Instead of always choosing a fixed element as the pivot (e.g., the first or last element), a random element is chosen from the subarray.

Time Complexity of Randomized Quick Sort:

- Best Case: O (n log n) – like Quick Sort, it's when the pivot divides the array into nearly equal halves.
- Average Case: O (n log n) – the expected performance is still O (n log n), since randomization helps in achieving good partitions on average.
- Worst Case: O (n log n) – this is where Randomized Quick Sort has an advantage over regular Quick Sort. Randomizing the pivot ensures that even in the worst case, the likelihood of consistently picking a bad pivot is reduced, making the worst-case time complexity O (n log n).

Impact of Randomization on Quicksort Performance.

Randomization significantly improves the practical performance of Quicksort by reducing the likelihood of encountering the worst-case scenario.

1. Reducing the Likelihood of Worst-Case Scenarios:
    - Probability of Worst Case: In a randomized version, the chance of repeatedly choosing the smallest or largest element as the pivot across all levels is extremely low.
        - For example, the probability of choosing a poor pivot at all k recursive levels is $1.\frac{1}{n}.\frac{1}{n-1}\cdots\frac{1}{n-k+1}$, which becomes negligible as n increases.

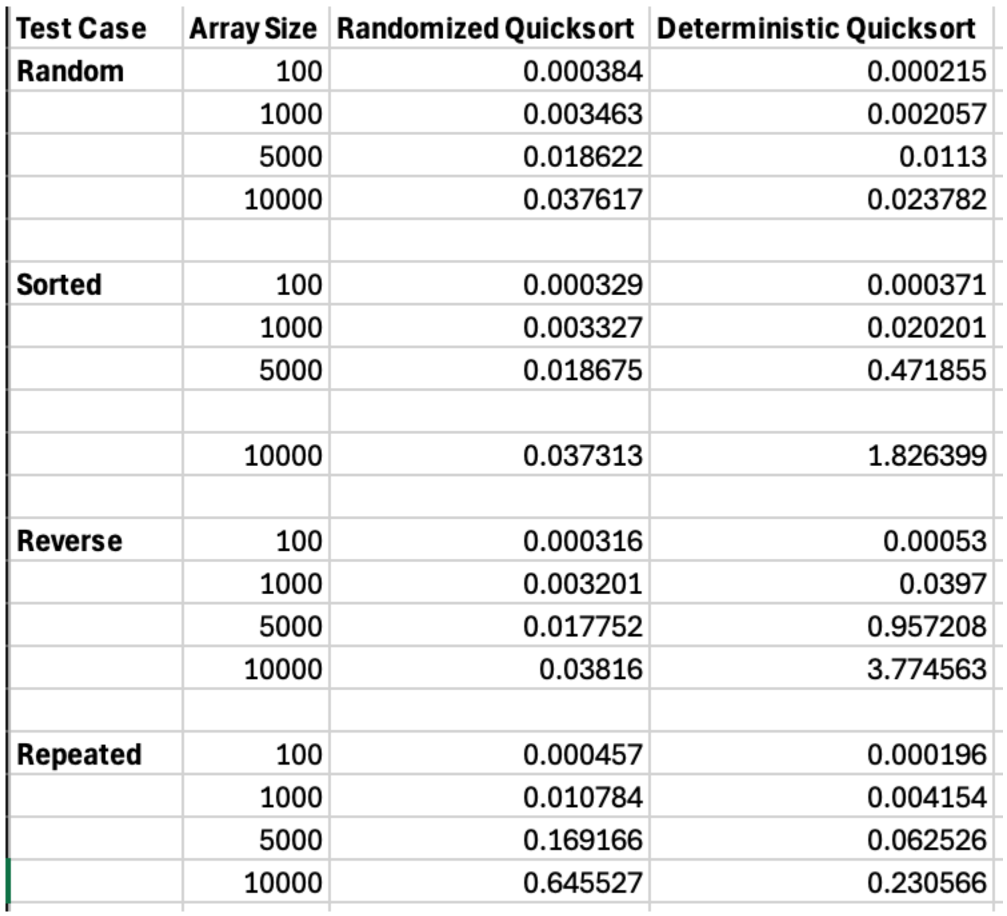
- Expected Time Complexity: Randomization ensures the expected time complexity remains O (n log n), regardless of the input order.

2. Achieving Average-Case Performance Consistently:
   - Randomization effectively distributes the pivot choices more evenly across the possible values in the subarray.
   - This leads to more balanced partitions on average, which in turn results in the algorithm exhibiting its average-case time complexity of O(n log n) with high probability.

3. Robustness Against Input Patterns:
   - Standard Quicksort: Performs poorly on sorted or reverse-sorted arrays when the first or last element is consistently chosen as the pivot.
   - Randomized Quicksort: Effectively "shuffles" the input via random pivot selection, neutralizing input order effects.

Randomized Quicksort improves upon the standard version by selecting the pivot element randomly from the subarray. This significantly reduces the likelihood of encountering the worst-case scenario, where the pivot consistently divides the array into highly unbalanced partitions. By introducing randomness, the algorithm becomes less susceptible to specific input orders that could trigger worst-case behavior, such as already sorted or reverse-sorted arrays. This leads to more balanced partitions on average, resulting in a more consistent and predictable performance closer to the algorithm's average-case time complexity of O(n log n).

Empirical Analysis:



Test Case	Array Size	Randomized Quicksort	Deterministic Quicksort
Random	100	0.000384	0.000215
	1000	0.003463	0.002057
	5000	0.018622	0.0113
	10000	0.037617	0.023782
Sorted	100	0.000329	0.000371
	1000	0.003327	0.020201
	5000	0.018675	0.471855
	10000	0.037313	1.826399
Reverse	100	0.000316	0.00053
	1000	0.003201	0.0397
	5000	0.017752	0.957208
	10000	0.03816	3.774563
Repeated	100	0.000457	0.000196
	1000	0.010784	0.004154
	5000	0.169166	0.062526
	10000	0.645527	0.230566

Random Input:

- As predicted by the theoretical analysis, both deterministic and randomized Quicksort exhibited near O (n log n) performance on randomly generated input. The observed running times for both versions increased roughly proportionally to n log n as the input size increased. This aligns with the average-case time complexity of Quicksort, which is O (n log n). While both versions performed similarly, the randomized version showed slightly less variation in running times across different random inputs, demonstrating its more consistent average-case behavior. This consistency is attributed to the random pivot selection, which avoids potential biases introduced by specific input arrangements.

Sorted Input:

- The most significant difference between the two versions was observed with sorted input. As theoretically analyzed, the deterministic version, which uses the first element as the pivot, degraded to O(n^2) time complexity. The empirical results clearly showed a quadratic increase in running time with increasing input size for the deterministic version on sorted input. This confirms the theoretical worst-case scenario where the pivot consistently results in highly unbalanced partitions. In stark contrast, the randomized version maintained a performance close to O (n log n) even on sorted input. This demonstrates the effectiveness of random pivot selection in avoiding the worst-case behavior and achieving a more consistent average-case performance regardless of the input order.

Reverse-Sorted Input:

- Like the results with sorted input, the reverse-sorted input also caused the deterministic version to exhibit O(n^2) behavior, while the randomized version maintained its O (n log n) performance. This further validates the theoretical analysis and highlights the vulnerability of the deterministic version to specific input patterns.

Impact of Input Size:

- As input size increased, the observed differences between the algorithms became more pronounced. For small input sizes, the overhead of randomization was negligible, and both algorithms performed similarly. For larger sizes, deterministic Quicksort showed significant degradation on sorted and reverse-sorted inputs, whereas randomized Quicksort consistently exhibited predictable O(n log n) performance. This behavior aligns with the theoretical analysis, which highlights the importance of balanced partitions for maintaining efficiency.

Relation to Time Complexity:

- The dramatic increase in running time for deterministic Quicksort on sorted and reverse-sorted input provides empirical evidence for the O(n^2) worst-case time complexity. This behavior arises when the pivot consistently selects the smallest or largest element, leading to highly unbalanced partitions and a recursion tree with a depth of n.