**Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization**

Bilal Mohammed

Department of Computer Science, University of the Cumberlands

Algorithms and Data Structures (MSCS-532-A01) - First Bi-term

Dr. Satish Penmatsa

January 30, 2025

**Time Complexity Analysis**

**Best Case**

In the best-case scenario, the pivot chosen during each partitioning step divides the array into two nearly equal halves. This results in a balanced recursion tree, where the depth of the tree is logn. At each level of the recursion tree, the partitioning step takes O(n) time. Therefore, the overall time complexity in the best case is O(nlogn).

**Average Case**

On average, Quicksort performs very efficiently. The pivot selection in the average case tends to divide the array into two reasonably balanced sub-arrays. Even if the splits are not perfectly equal, the depth of the recursion tree remains O(logn), and the partitioning step at each level still takes O(n) time. This results in an average-case time complexity of O(nlogn). The reason for this is that the algorithm avoids consistently poor pivot choices, and the logarithmic depth of the recursion tree ensures efficient sorting.

**Worst Case**

The worst-case scenario occurs when the pivot chosen is consistently the smallest or largest element in the array. This happens, for example, when the input array is already sorted or reverse-sorted, and the pivot is always selected as the last element (in the deterministic version). In this case, the partitioning step results in one sub-array of size n−1 and another sub-array of size 0. This leads to a highly unbalanced recursion tree with a depth of n, and each partitioning step still takes O(n) time. As a result, the worst-case time complexity is O(n^2).

**Why Average Case is O(nlogn) and Worst Case is O(n^2)**

The average-case time complexity of O(nlogn) arises because the pivot selection, on average, divides the array into two balanced sub-arrays. This balanced division ensures that the recursion tree has a logarithmic depth, and the partitioning step at each level takes linear time. In contrast, the worst-case time complexity of O(n^2) occurs when the pivot selection is consistently poor, leading to an unbalanced recursion tree with a depth of n. This imbalance causes the algorithm to perform poorly, especially for already sorted or reverse-sorted inputs.

**Space Complexity**

The space complexity of Quicksort is primarily determined by the recursion stack. In the best and average cases, the depth of the recursion tree is O(logn), so the space complexity is O(logn). However, in the worst case, the recursion depth can be O(n), leading to a space complexity of O(n). Additionally, the partitioning step requires a small amount of extra space for temporary variables or pointers, but this is typically negligible compared to the recursion stack overhead.

**Additional Overheads**

1. **Recursion Overhead**: The recursive calls add overhead due to the management of the call stack.
2. **Partitioning Overhead**: The partitioning step involves iterating through the array and rearranging elements, which adds some computational overhead.
3. **Randomization Overhead**: In the randomized version, selecting a random pivot adds a small overhead, but this is outweighed by the benefit of reducing the likelihood of worst-case behavior.

**Randomized Quicksort**

**Impact of Randomization on Performance**

Randomization plays a crucial role in improving the performance of Quicksort. In the deterministic version of Quicksort, the pivot is typically chosen as the first or last element of the array. While this approach works well in many cases, it can lead to poor performance when the input array is already sorted or reverse-sorted. In such scenarios, the pivot selection results in highly unbalanced partitions, causing the algorithm to degrade to its worst-case time complexity of O(n^2)*.*

By introducing randomization, we ensure that the pivot is chosen uniformly at random from the subarray being sorted. This randomness significantly reduces the probability of consistently selecting bad pivots, such as the smallest or largest element in the subarray. As a result, the partitions are more likely to be balanced, leading to a recursion tree with a depth closer to logn rather than n. This balanced recursion tree ensures that the average-case time complexity of O(nlogn) is achieved with high probability.

**Reducing the Likelihood of Worst-Case Scenarios**

The worst-case scenario for Quicksort occurs when the pivot selection consistently results in highly unbalanced partitions. For example, if the pivot is always the smallest or largest element, one partition will contain n−1 elements, and the other will be empty. This leads to a recursion tree with a depth of n, causing the algorithm to perform poorly.

Randomization mitigates this issue by making it highly unlikely that the pivot selection will consistently produce bad partitions. Even if a bad pivot is chosen occasionally, the randomness ensures that such occurrences are not repeated in subsequent recursive calls. Over multiple partitions, the algorithm tends to achieve balanced splits, which keeps the recursion tree shallow and maintains the O(nlogn) average-case performance.

**Empirical Evidence**

In my implementation, I observed that the randomized version of Quicksort performs consistently well across different input distributions, including random, sorted, and reverse-sorted arrays. For example:

* On random arrays, both deterministic and randomized Quicksort perform similarly, with O(nlogn) time complexity.
* On sorted or reverse-sorted arrays, the deterministic version degrades to O(n^2), while the randomized version maintains its O(nlogn) performance.

**Experimental Setup**

To conduct the empirical analysis, I tested both the deterministic and randomized versions of Quicksort on three types of input distributions:

**Random Arrays**: Arrays with elements in random order.

**Sorted Arrays**: Arrays with elements in ascending order.

**Reverse-Sorted Arrays**: Arrays with elements in descending order.

**Random Arrays**

For random arrays, both the deterministic and randomized versions of Quicksort performed similarly. The running time for both versions scaled as O(nlogn), which aligns with the theoretical average-case time complexity. This is because, in random arrays, the pivot selection in the deterministic version is unlikely to consistently produce unbalanced partitions. Randomization did not provide a significant advantage in this case, as the input distribution itself already ensured balanced partitions.

**Sorted Arrays**

For sorted arrays, the deterministic version of Quicksort performed poorly, with a running time that scaled as O(n^2 ). This is because the pivot (chosen as the last element) was always the largest element, resulting in highly unbalanced partitions. In contrast, the randomized version maintained its O(nlogn) performance, as the random pivot selection prevented consistently poor partitions. This result clearly demonstrates the advantage of randomization in avoiding worst-case behaviour.

**Reverse-Sorted Arrays**

For reverse-sorted arrays, the results were similar to those for sorted arrays. The deterministic version degraded to O(n^2) because the pivot (chosen as the last element) was always the smallest element, leading to unbalanced partitions. The randomized version, however, continued to perform efficiently with O(nlogn) running time, as the random pivot selection ensured balanced partitions.

**Discussion of Results**

**Random Arrays**: Both versions performed well, as the input distribution naturally led to balanced partitions. Randomization did not provide a significant advantage here.

**Sorted and Reverse-Sorted Arrays**: The deterministic version performed poorly due to consistently poor pivot choices, resulting in unbalanced partitions and O(n^2 ) running time. The randomized version, however, maintained its efficiency by avoiding these poor pivot choices, ensuring balanced partitions and O(nlogn) running time.

These observations highlight the importance of randomization in Quicksort. While the deterministic version is efficient for random inputs, it is vulnerable to worst-case behavior for specific input distributions like sorted or reverse-sorted arrays. Randomization addresses this vulnerability by making worst-case scenarios highly unlikely, ensuring robust performance across all input types.