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

Shreya Sapkota

Algorithms and Data Structures

Vanessa Cooper

11/17/2024

Quicksort

Quicksort is a comparison-based sorting algorithm that sorts lists or arrays of elements using a divide-and-conquer strategy. The fundamental concept of Quicksort is to divide the array into two subarrays, with the items in the right subarray being larger than the pivot and the elements in the left subarray being smaller. This procedure is then applied recursively to the subarrays, eventually resulting in the sorting of the full array. The effectiveness of the algorithm depends on the pivot selection. In the worst-case scenario, a poor pivot choice can result in inefficiencies.

Performance Analysis

Time Complexity

In the best-case scenario, the pivot creates a balanced recursion tree by splitting the array into two equal sections, which results in O(nlogn). Even for average-case scenarios, the pivot selection often produces partitions that are well-balanced. Each division results in a logarithmic recursion depth because it divides the array in half. The overall complexity in such a case is O(nlogn) as partitioning the array takes O(n) and the recursion depth is O(logn).

When the pivot is the smallest or largest member repeatedly, O(n2) happens, resulting in a skewed recursion tree with almost all the elements on one side of the partition. The time complexity in this worst case scenario is quadratic since each split requires O(n) time and there would be n recursive calls.

Space Complexity

Since the recursion depth is logarithmic in the best or average scenario, Quicksort's space complexity for the recursive call stack is O(logn). But in the worst situation, the depth of recursion can exceed O(n), resulting in a complexity of space of O(n).

Randomized Quicksort

A variant of the traditional Quicksort algorithm called Randomized Quicksort was created to address the issue of inefficient pivot selection. The pivot in Randomized Quicksort is selected at random from the array, making it unlikely that it will always be the smallest or largest element. This approach lessens the possibility of the worst-case time complexity of O(n2), which happens when the pivot is badly selected and produces extremely unbalanced divisions, by adding randomness. By avoiding worst-case scenarios, Randomized Quicksort enhances performance and typically has a time complexity of O(nlogn).

Empirical Analysis

After executing the implementation of both quicksort and randomized quicksort on arrays of varying sizes and distributions, the following observations were made:

Size	Distribution	Quicksort Time (s)	Randomized Quicksort Time (s)
100	Random	0.000095	0.000094

100	Sorted	0.000056	0.000082
100	Reverse Sorted	0.000053	0.000082
1000	Random	0.001	0.001027
1000	Sorted	0.000658	0.00104
1000	Reverse Sorted	0.000647	0.001093
10000	Random	0.007229	0.007453
10000	Sorted	0.007357	0.010399
10000	Reverse Sorted	0.007378	0.010991

According to the table, Quicksort's traditional and randomized implementations perform similarly for random distributions of all input sizes, suggesting that both algorithms run close to O(nlogn). However, the additional overhead brought about by random pivot selection is reflected in the small timing changes for randomized Quicksort.

Traditional Quicksort typically performs slightly quicker than the randomized version for sorted and reverse-sorted arrays, especially if input sizes are greater. Since traditional Quicksort eliminates the need for random number generation, this result can be attributed to its lower overhead. However, the randomized version still performs comparably and avoids the worst-case O(n2) behavior on highly skewed or ordered data.

Conclusion

In conclusion, Quicksort maintains O(nlogn) time complexity and works effectively on randomly distributed data for both deterministic and randomized Quicksort. On sorted and reverse-sorted data, however, randomized Quicksort exhibits higher robustness and avoids the

O(n2) worst-case situation that deterministic Quicksort typically encounters with these inputs. Randomized Quicksort has a minor overhead because it uses random pivot selection, but it is more dependable when dealing with unexpected data distributions.

References

GeeksforGeeks. (n.d.). Quicksort. Retrieved from https://www.geeksforgeeks.org/quick-sort/