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

Muneeb Mohiuddin

Algorithms and Data Structures

University of the Cumberland’s

September 22, 2024

**Introduction**

Sorting algorithms are among the basic elements of the theory of computation and are critical for data storage and retrieval. Of the many sorting algorithms, Quicksort can be considered as one of the most effective in terms of the number of operations and the variety of applications where it can be implemented effectively; starting from indexes and ending with data processing platforms. This paper gives a deeper understanding of the Quicksort algorithm and its deterministic version, performance evaluation of Quicksort, and comparison of deterministic Quicksort with the random Quicksort based on input data (Aftab et al., 2021).

This leads to a discussion of the theory behind the Quicksort algorithm and its deterministic and randomized versions as well as a performance analysis. The findings provide an understanding of the importance of algorithm optimization based on an algorithm and the importance of ensuring randomness in algorithms in exceptional input situations. The information derived from this research is indeed relevant to present day software engineering, particularly the distributed data processing systems of Apache Hadoop and Spark, and the search methodologies for Internet services and mobile platforms in which true resource optimization is imperative.

**Algorithm Overview**

The basic idea of the Quick sort algorithm is the divide and rule concept. The basic concept is to select an element as pivot from the array, and after that make the array divide into two subarrays in which left side is having elements lesser than pivot and right side is having elements greater than the pivot. The algorithm then proceeds to sort the left subarray and right subarray. After sorting all these subarrays, the arrays are merged to form the final sorted array as shown by Taiwo et al., (2020).

The steps in the deterministic version of Quicksort are as follows:

1. **Select a pivot -** The pivot can be chosen from any index in the given array (in a usual case, it has been chosen from 0th, the last, or an arbitrary middle index).
2. **Partition the array -** It involves partitioning the list such that all elements that are less than the chosen pivot is placed on its left, whereas all elements that are greater than the pivot form the right.
3. **Recursively sort -** Quicksort is again and again used on the left partition and right partition in decreasing order of the key value.
4. **Combine -** Upon sorting the subarrays, they are joined together with other subarrays to produce the resulting sorted array.

**Performance Analysis**

**Time Complexity**

Time complexity plays a decisive role when it comes to comparing the performances of the various algorithms, in this case, Quicksort is set back and boosted by the chosen pivot and the distribution of the input.

* **Best Case -** Quicksort has the worst-case time complexity of O(n2) which appears when repeatedly the pivot creates two subarrays each of size n/2 on average. In such cases, the depth of recursion tree formation is log n whereas the work done each level is proportional to n hence giving a total time complex of O (nlogn).
* **Average Case -** About the average, Quicksort works with a time complexity of O(n log n). This occurs because, as you can see, the pivot separates the array into two approximately equal parts on average. Having analyzed all the possible input data, it is possible to mention that most of them are random, which does not let the pivot cause unbalanced partitions in the majority of cases.
* **Worst Case -** Even the worst-case time complexity of Quicksort is O(n2) and this happens when the pivot always splits the array very unbalancing (for instance, if it sorts the array, and the pivot element is the number 1 or n). It is possible that recursion goes to O(n), and the sort degrades to a quadratic time sort.

**Space Complexity**

Space complexity of Quick sort is O(logn) since Quick sort requires a call stack to divide the array into subarrays. In the best and average cases, the values of Sc and T are proportional to logn, which means logarithmic space usage. However, in the worst case, or in other words, when the partitioning is unbalanced, the recursion depth can be as bad as O(n), hence the space complexity can also be as high as O(n).

**Randomized Quicksort**

In general, deterministic Quicksort can have very poor performance depending on what kind of data is being sorted, for example sorted or reversed sorted arrays. To this end, the use of a randomized version of the Quicksort algorithm can be adopted regardless of the choice of the pivot from the array. Through the addition of randomness, the algorithm greatly decreases the probability of having unbalanced partitions throughout and thus, increases the overall expected efficiency of the algorithm (Hossain et al., 2020).

The selection of the pivot being arbitrary, this version minimizes the chances of selection leading to the worst-case time complexity of O(n2). Randomization allows the pivot selection to result in good splits as frequently as in bad ones, which makes the scale of the situation less fluctuated depending on the types of input data.

**Empirical Analysis: Deterministic vs Randomized Quicksort**

In an effort to compare deterministic and randomized Quicksort productivity in a direct manner, we performed tests using arrays of different type and length. The arrays tested included:

* **Random Array -** An array of random integers.
* **Sorted Array -** An array which is pre-sorted in increasing order.
* **Reverse-Sorted Array -** An array arranged in the descending order.

**Test Data**

The arrays tested included sizes ranging from 1000 to 50,000 elements.

**Observations and Discussion**

The empirical results revealed the following trends:

* **Random Array Performance -** While programmed, deterministic and random Quicksort processed random arrays with similar efficiency; the execution time roughly equals O(nlogn). Any random input data will confirm that, even in the deterministic case, the approximation divides the array in half most of the time, as stated by Peters (2021).
* **Sorted and Reverse-Sorted Arrays -** Here we can see that the basic deterministic version of Quicksort showed a very poor behavior when the array was already sorted or reverse sorted. In the worst case, since the pivot does not split the array in half, this algorithm behaves as a comparison sort and has the time complexity of O(n²). However, the performance of the randomized version was way better specifically on these inputs given the fact that the pivots are chosen randomly which ensure that the partitions generated are relatively balanced.
* **Randomized Version Advantage -** The randomized version of Quicksort was invariably faster than the deterministic one for sorted and reverse sorted input arrays. This shows how incorporating randomization is beneficial on average since even worst-case occurrences in sorting times for various input types can be fixed.

**Conclusion**

The deterministic version, whilst being O(n) on average, can take quadratic time in the worst-case scenario when input is sorted in certain ways, for example by using sorted arrays. The problem with the Quicksort is its unpredictability since it sometimes performed poorly by choosing a wrong pivot, Randomized Quicksort rectify this by always choosing the pivot at random.

As we see it from theoretical considerations as well as experimental evaluation, randomized Quicksort appears to be the more favorable in terms of its use for a variety of data sets. The information that is obtained throughout this assignment is relevant to working with data and building algorithms in conditions that require high speed and efficiency.

**References**

Aftab, A., Ali, M. A., Ghaffar, A., Shah, A. U. R., Ishfaq, H. M., & Shujaat, M. (2021). Review on performance of quick sort algorithm. *Int. J. Comput. Sci. Inf. Secur.(IJCSIS)*, *19*(10.5281).

Hossain, M. S., Mondal, S., Ali, R. S., & Hasan, M. (2020, March). Optimizing complexity of quick sort. In *International conference on computing science, communication and security* (pp. 329-339). Singapore: Springer Singapore.

Peters, O. R. (2021). Pattern-defeating quicksort. *arXiv preprint arXiv:2106.05123*.

Taiwo, O. E., Christianah, A. O., Oluwatobi, A. N., & Aderonke, K. A. (2020). Comparative study of two divide and conquer sorting algorithms: quicksort and mergesort. *Procedia Computer Science*, *171*, 2532-2540.