Quicksort Algorithm: Implementation, analysis, and Randomization

The quicksort algorithm follows the divide-and-conquer technique for sorting the elements. It selects the pivot element, divides the array around the pivot element, and recursively sorts those partitions. Data processing, database management, network routing, and file systems use the quicksort algorithm because of its' average time complexity of O (n log n). That is comparatively faster than O(n^2) sorting algorithms for large data sets. However, quicksort is challenging if the selected pivot divides the array into unbalanced subarrays. That will lead to the worst-case of O (n ^2). A randomized version of quicksort will reduce the probability of unbalanced partitions by choosing pivot elements randomly.

Time Complexity Analysis

- **Best case:** It will excel when selected pivot elements divide the array into equal subarrays. The execution time complexity will be O (n log n). Recurrence relation will be T(n) = 2T(n/2) + O (n).
- **Average-case:** On average, pivot elements partition the array into two halves. That will match the time complexity of the best case, i.e., O (n log n).
- **Worst case:** The challenges will be when it always selects small or large pivot elements that further partition the array into unbalanced subarrays. It will increase the time complexity of O(n^2). The recurrence relation will be T(n) = T (n-1) + O(n) and leads to quadratic complexity.

Space Complexity

- The in-place implementation uses only O (log n) extra space for the recursion stack.
- Non-in-place versions use O(n) extra space

Randomized Quicksort

A randomized version of quicksort will pick a pivot element randomly from the array and divide it into sub-arrays. Randomizing the quicksort technique will impact performance and reduce the probability of hitting the worst-case scenario. It will ensure the average execution time of O (n logn) is retained. It will maintain consistency with performance on sorted or reversed-sorted inputs.

Empirical Analysis:

Deterministic and randomized quicksort will have different execution times on distributed inputs. Deterministic quicksort will not perform consistently on random, sorted, and reverse-sorted inputs because the determined pivot element will divide the array into unbalanced partitions. On the other hand, randomized quicksort will perform better on sorted and reverse-sorted inputs as it picks random pivot elements to partition the array reasonably in balance. Randomization outperforms deterministic because it will reduce the probability of the worst-case scenario. The Quicksort algorithm is powerful and efficient with an average O-case complexity (n logn). Furthermore, we can mitigate the worst-case scenario by implementing a random version of quicksort.

References

Blum, A. (2011). Probabilistic analysis and randomized Quicksort. Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0906.pdf

Price, E. (2021). Randomized algorithms: Lecture 3 - Randomized Quicksort. University of Texas at Austin. Retrieved from https://www.cs.utexas.edu/~ecprice/courses/randomized/fa21/scribe/lec3.pdf

Smid, M. (2019). Discrete structures for computer science. Retrieved from https://dokk.org/library/discrete_structures_computer_science_2019_Smid