Name: Shimon Kumar Bhandari

ID: 005030903

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

Assignment 5

Performance Analysis of Quick Sort

The sorting algorithm Quicksort finds widespread practical usage because it delivers excellent performance in the average case while performing in-place sorting. The time required by Quicksort depends on the input data along with the selected pivot point. The best outcome for Quicksort produces an O(nlogn) time complexity when the pivot element creates two equal parts of the array. The balanced partition process creates a recursion tree that reaches O(logn) height while performing O(n) operations at each level which results in an O(nlogn) complexity.

The average partitioning process results in O(nlogn) time complexity since the pivot element tends to split the array into similar parts on average which supports a balanced recursion tree. The time complexity of Quicksort becomes O(n^2) when it faces the most unfavorable conditions. The recursion tree reaches a maximum height of O(n) when the pivot remains consistently smallest or largest thus leading to highly unbalanced partitions and O(n) work at each level. The pivot's normal behavior creates O(nlogn) performance because it divides the array into balanced subgroups yet the O(n ^ 2) worst-case happens when partitions stay consistently unbalanced. The worst-case behavior risks can be reduced through pivot selection strategies that include randomized methods or the "median-of-three" technique (Cormen et al., 2009).

The space requirements of Quicksort as an in-place algorithm increase because its recursive nature requires additional memory usage. During best and average scenarios the recursion depth reaches O(logn) thus leading to O(logn) space complexity. When the recursion depth reaches its worst possible state of O(n) then the recursion stack becomes responsible for O(n) space complexity (Sedgewick & Wayne, 2011). The performance of the algorithm heavily relies on the partitioning step that requires element swapping operations because this step is both crucial and

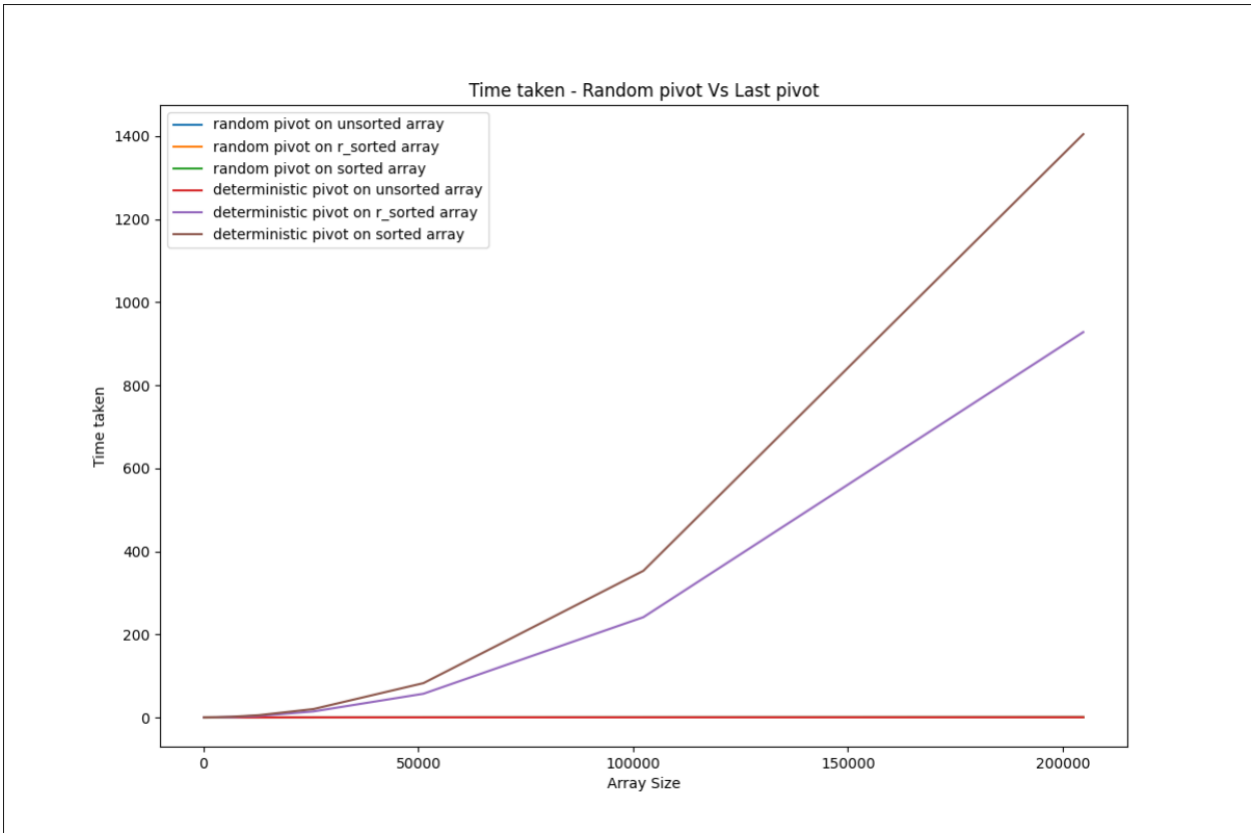
computationally demanding. The combination of practical efficiency and avoidance methods for worst-case situations makes Quicksort an ideal selection for sorting substantial datasets.

Randomized Quick Sort Analysis

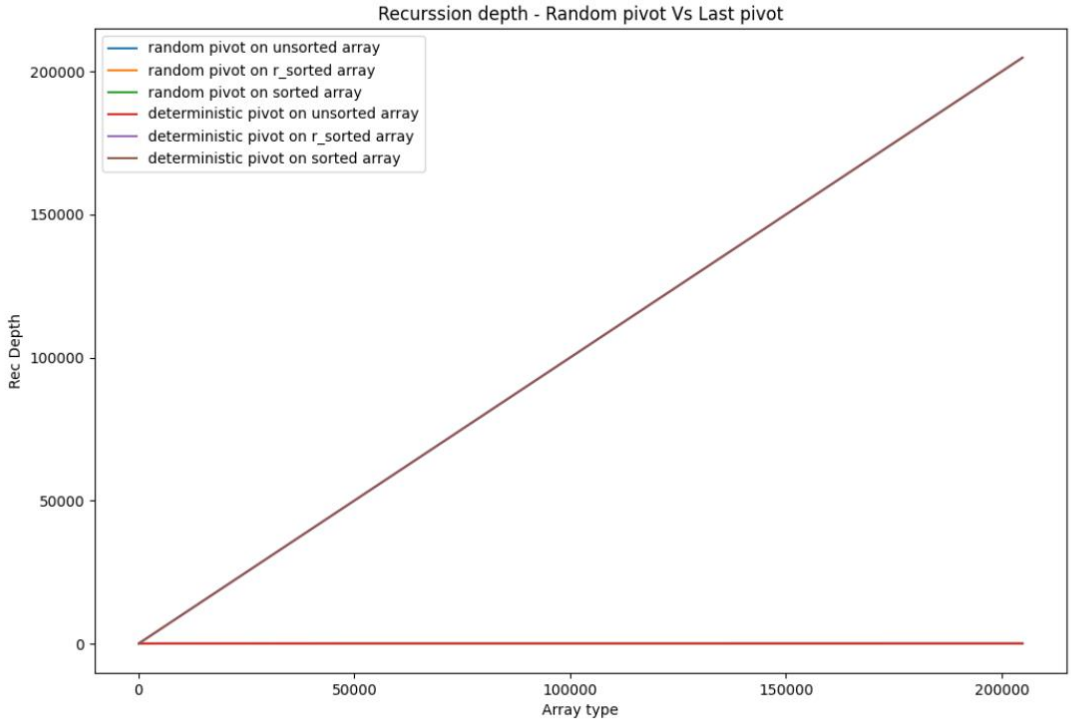
Randomization serves as a vital element for Quicksort because it minimizes the probability of reaching the O(n^2) worst-case performance. Standard Quicksort performs poorly when the pivot element chooses the smallest or biggest value repeatedly which causes severely unbalanced divisions in the sorting process. The worst-case behavior occurs when the input array contains sorted data because selecting the pivot element from either the first or last position causes the partition to be one sided. Random selection of the pivot from the array according to a uniform distribution reduces the likelihood of continuously picking a poor pivot. Randomized Quicksort uses random pivot selection to prevent specific inputs from causing worst-case scenarios thus making the algorithm more practical and dependable.

Randomized Quicksort maintains an average-case time complexity of O(nlogn) because randomization selects pivot elements which split arrays into evenly balanced sections on average. Each level of the recursion tree contains O(n) work while its height reaches O(logn) because of balanced partitioning which results in an efficient O(nlogn) complexity. Large datasets benefit from exponential reduction of unbalanced partition occurrences which virtually eliminates the worst-case scenario. Randomization techniques eliminate the theoretical significance of O(n^2) worst-case time complexity because it becomes irrelevant to most practical applications. Randomized Quicksort preserves in-place sorting characteristics and exhibits average-case space complexity of O(logn) because its recursion depth stays logarithmic.

Randomization as a performance enhancer for Quicksort creates balanced partitions while reducing the chance of worst-case scenarios thus it is considered a dependable sorting method.



As we can see above, the time taken by the randomized Quicksort is always similar against all types of arrays (Randomly generated array, already sorted arrays, Reverse-sorted arrays) of different sizes i.e. the time complexity is O(nlogn). But the time taken by the deterministic (last element pivot) Quicksort is different for different types of arrays. It does good on random array where the time complexity is O(nlogn). It does worst for already sorted and reverse sorted array. This is because the subarrays at each recursion do not have any element in one side of pivot. In this case the time complexity goes to O(n^2) because the recursion goes to O(n) and for each recursion it takes O(n) time to organize elements.



The same goes for space complexity, for randomized Quicksort the maximum depth of recursion is always similar for all types of arrays. As the array size increases the recursion also increases as O(logn). This is because the randomized version avoids the worst case, and it will always have a similar number of elements on the left and right side of pivot. But for the last element pivot Quicksort, the recursion depth is equal to the size of array for reverse sorted and sorted array. This is also because there are no elements on one side of pivots at each recursive call. Thus, the recursion increases linearly as per the array size i.e. O(n).

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.