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

Shrisan Kapali

Student Id: 005032249

MSCS 532 – Algorithms and Data Structures

Satish Penmatsa

February 02, 2025

Quicksort Analysis

A quicksort algorithm is where an array is sorted by selecting a pivot element, partitioning the array according to the pivot, and recursively sorting the subarrays (Kitthu, 2024). The time complexity of quicksort is given as follows.

Best Case Time Complexity

The best-case time complexity of a quicksort is O (nlogn). The best-case scenario occurs when the selected pivot element divides the given array roughly into two equal-sized sub-arrays at every partition (Kitthu, 2024). In the case where each partition divides the array into two roughly equal sub-arrays, the depth of the recursion is O(logn), and at each level, O(n), work is done to partition the array (Kitthu, 2024). Thus, this gives the overall time complexity of O(nlogn). The recurrence relation of the best case is given as

$T(n) = 2\,T(n/2) + O(n)$

Solving this recurrence relationship using Master Theorem,

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ [Master theorem]

Comparing the recurrence relation with master theorem, get a=2, b=2, k=1, p=0,

From master theorem cases where a=b,

$T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$

$T(n) = \Theta(n \log n)$

Average Case Time Complexity

The average time complexity of quicksort is the same as that of the best case, which is O (nlogn). In an average case, the pivot is selected at random, which, on average, divides the array into almost two equal sub-arrays. So, the depth of the recursion is O (logn), and O(n) work is done at each level, giving the total time complexity of O (nlogn). The recurrence relation is given as

$T(n) = T(n/c) + T((c-1)n/c) + O(n)$, where c is random pivot split ratio,

Assuming the chosen pivot divides the array into two equal subarrays, let assume c=2

$T(n) = T(n/2) + T((2-1)n/2) + O(n)$

$T(n) = T(n/2) + T(n/2) + O(n)$

$T(n) = 2T(n/2) + O(n)$

Solving as above gives the time complexity of O(nlogn).

Worst Case Time Complexity

The worst-case time complexity of quicksort is $O(n^2)$. The worst case occurs when the selected pivot at each partitioning step is always the smallest or largest element in the array (Kitthu, 2024). When the array is already in sorted order, and the first element is chosen as the pivot, at each partition level, the array only divides into a single subarray. The recurrence relation is given as

$T(n) = T(n-1) + O(n)$,

We solve this by expansion as master theorem is not applicable,

$T(n-1) = T(n-2) + (n-1)$

$T(n-2) = T(n-3) + (n-2)$

If we continue to expand,

$T(n) = T(1) + 2 + 3 + 4 + \ldots + (n - 1) + n$

$T(n) = n(n+1) / 2$

$T(n) = (n^2 /2) + (n/2)$

$T(n) = O(n^2)$

Space complexity of quicksort

Quicksort does not require any additional array or memory space for sorting. The space complexity of quicksort for best and average cases is O (logn), as the max recursion depth in these cases is given by O (logn). For the worst case, the recursion depth can be the length of the array, so the worst-case space complexity is O(n) (Kitthu, 2024).

Overheads associated with quicksort

One of the disadvantages of quicksort is that if the pivot element is chosen poorly, it can have the worst time complexity. This can be solved by using strategies such as randomized pivot selection, tail recursion elimination, hybrid algorithms, multi-threading, and median-of-three partitioning. Another overhead is related to maintaining the relative order of the elements. Quicksort is considered unstable as the relative order of the elements is not maintained while swapping them. Quicksort also has a high recursive overhead while sorting small data sets.

Randomized quicksort

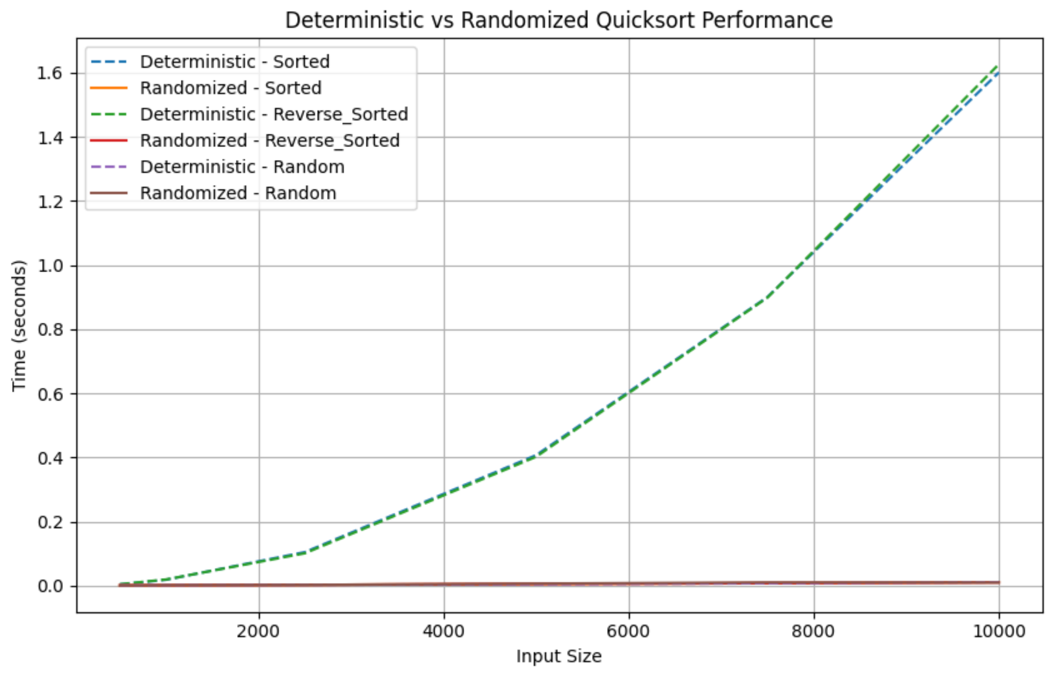
A randomized quicksort is a technique where the pivot element is selected randomly. As the pivot is selected at random, it avoids cases where the pivot at each partition is always selected poorly. It is highly unlikely that the smallest or largest element is selected as the pivot in each partition; thus, randomization avoids the worst case. It has the best and average time complexity of O(nlogn); however, the worst case is still $O(n^2)$.

Implementation of deterministic and randomized quicksort

The partition method of the deterministic quicksort was implemented by selecting the lowest element as the pivot. We had two pointers, left and right, where the pointer left moved from left to right position to find the element that was greater than the pivot, and the pointer right moved from right to left, finding the element that was smaller than the pivot. The pointer left stops when it finds the element greater than the pivot, and the pointer right stops when it finds the element less than the pivot. We then swap elements at positions left and right and repeat this step until left <=right.

The partition method of the randomized quicksort was implemented by selecting the pivot index at random, and the pivot index falls between the low and high elements. Once the pivot index is selected, we swap the array element at the pivot index with the highest element and assign the pivot as the element at the highest position. We start the left pointer at a low -1 index and loop the right pointer from low to high. If the array at the right pointer index is less than the pivot, we increment the left pointer and swap the value of the element at the left and right pointer index. Once the for loop is completed, we swap the remaining element of the left pointer with the highest element.

To analyze the execution time and performance between the deterministic and randomized quicksort, arrays of different sizes and distributions were created in sorted, reverse sorted, and random order. Each test case array was sorted using the deterministic and randomized quicksort. When the array size was small, the difference in performance was negligible. However, as the array size increased, the time it took for the deterministic quicksort to sort the arrays that were already in sorted and reverse sorted order grew much higher compared to the time it took for randomized quicksort to sort the same data sets. This also aligns with the theoretical concept of quicksort worst case as in the worst-case scenario, in the test case is the case where the array is already in sorted or reverse sorted order, the time complexity of a deterministic quicksort is O (n^2) and as shown in the graph the time to sort these data sets grew much higher. While, in theory, randomized quicksort is said to avoid the worst case, in the Python test cases, the randomized quicksort had an average time complexity in all the data variants. Thus, we can observe that randomization does help avoid the worst time complexity and delivers higher performance regardless of the data size and distributions.



References

Kitthu, H. A. (2024, December 24). *Quick sort algorithm: Time complexity and practical uses*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/quick-sort-algorithm