**Assignment 5: Quicksort Algorithm**

**Implementation, Analysis, and Randomization**

**Quicksort Implementation and Analysis:**

Quicksort is a fast-sorting algorithm that works by picking a pivot element, then dividing the array into two parts: one with elements smaller than the pivot and one with elements greater. This process is repeated recursively for each part. The time complexity of Quicksort is usually O(nlogn) on average, but can be O(n^2) in the worst case if the pivot is poorly chosen (Hoare, 1962). Randomizing the pivot can help avoid the worst-case scenario, improving performance. Quicksort is efficient for large datasets but may have higher space usage due to recursion.

The code snippet given below shows the Quicksort implementation:

A computer screen with text

Description automatically generated

**Performance Analysis:**

**Best Case:** In the best case, Quicksort picks a pivot that splits the array into two nearly equal parts each time. This leads to a recursion depth of O(logn)), and each partition step takes O(n) time. So, the overall time complexity is O(nlogn).

**Average Case:** In the average case, Quicksort’s pivot usually divides the array into balanced parts. This keeps the recursion depth at O(logn), and each partition step still takes O(n) time. Therefore, the average case time complexity is also O(nlogn).

**Worst Case:** The worst case occurs when the pivot choice is bad, like picking the smallest or largest element. This results in uneven splits and a recursion depth of O(n), making the time complexity O(n^2)

**Space Complexity:**

Quicksort uses extra space mainly for the recursion stack. In the best and average cases, the recursion depth is O(logn), so the space complexity is O(logn). In the worst case, with deep recursion, the space complexity is O(n).

**Additional Overheads:**

The pivot selection method can affect performance. Simple methods, like choosing the first or last element as the pivot, can cause poor performance on sorted data. More advanced methods, like picking a random pivot or using the "median of three," reduce this risk but add a bit of extra work.

**Randomized QuickSort**

Randomizing the pivot in Quicksort helps avoid the worst-case scenario (O(n^2) by making it less likely to pick a bad pivot. Without randomization, if the pivot is always the smallest or largest element, it can cause uneven splits, leading to deep recursion and a time complexity of O(n^2) (Sedgewick, 2011). With randomization, the pivot is chosen randomly, which lowers the chances of bad splits and improves performance. Although the worst case can still happen, it becomes much less likely. In practice, randomization helps Quicksort run in O(nlogn) time on average, even on inputs where the deterministic version might struggle.

The picture given below shows the code snippet for Randomized Quicksort:

A computer screen with text on it

Description automatically generated

**Impirical Analysis:**

Randomized QuickSort consistently shows better performance across various input types and sizes. Its randomization strategy helps it avoid the issues that affect Standard QuickSort, especially with reversed and random data. As input sizes grow, the efficiency of Randomized QuickSort becomes more evident, making it a more reliable and scalable option for diverse situations (Sedgewick, 2011).

Given below is the performance overview for different data size and patterns:

* **Random Inputs**: Randomized QuickSort generally outperforms Standard QuickSort with random data, regardless of the input size. The randomization process helps avoid the worst-case scenarios that can significantly slow down Standard QuickSort (Sedgewick, 2011).
* **Sorted Inputs**: Both algorithms perform well with sorted data, but Randomized QuickSort has a slight edge. This indicates that even when the data is already in order, randomization still offers a performance benefit (Sedgewick, 2011).
* **Reversed Inputs**: The performance difference between the two algorithms is most noticeable with reversed data. Standard QuickSort struggles with reversed inputs, resulting in slower processing times. In contrast, Randomized QuickSort handles these inputs more efficiently, demonstrating its robustness and ability to avoid worst-case scenarios (Sedgewick, 2011).

The table below shows the time taken by Quicksort and Randomized Quicksort for different data sizes and datta patterns:

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset Size** | **Pattern Type** | **Quicksort Time (s)** | **Randomized Quicksort Time (s)** |
| 100 | sorted | 0.000585 | 0.000076 |
| 100 | reverse\_sorted | 0.000287 | 0.000067 |
| 100 | random | 0.000066 | 0.000064 |
| 100 | repeated | 0.000076 | 0.000075 |
| 1000 | sorted | 0.026364 | 0.00089 |
| 1000 | reverse\_sorted | 0.026524 | 0.00099 |
| 1000 | random | 0.000869 | 0.000924 |
| 1000 | repeated | 0.002631 | 0.001865 |
| 10000 | sorted | 2.635135 | 0.012757 |
| 10000 | reverse\_sorted | 2.464587 | 0.011772 |
| 10000 | random | 0.01092 | 0.012832 |
| 10000 | repeated | 0.218594 | 0.126125 |

The table presents the execution times of Standard Quicksort and Randomized Quicksort for different dataset sizes (100, 1000, and 10,000) and patterns (sorted, reverse\_sorted, random, and repeated). For smaller datasets (size 100), the times for both algorithms are very small and fairly similar. As the dataset size increases, the performance difference becomes more noticeable. Randomized Quicksort consistently performs better than Standard Quicksort, especially for sorted and reverse-sorted data, where Standard Quicksort takes significantly longer. However, for random and repeated data, the performance difference is smaller. In general, Randomized Quicksort's time remains much more consistent across various input patterns compared to Standard Quicksort.

**References:**

Hoare, C. A. R. (1962). *Quicksort*. *Computer Journal, 5*(1), 10-16. <https://doi.org/10.1093/comjnl/5.1.10>

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