Quicksort is one of the most efficient and widely used sorting algorithms due to its average case time

complexity of O(n log n) and in-place sorting capabilities. This report explores the deterministic and

randomized versions of Quicksort, analyzing their performance across different input distributions.

1. **Implementation**

<https://github.com/nhemani33090/MSCS532_Assignment5>  
**Deterministic Quicksort**: Uses a fixed pivot selection strategy (middle element) for partitioning the array.

**Randomized Quicksort:** Uses a random pivot selection to reduce the likelihood of worst-case behavior.

1. **Performance Analysis**

**Time Complexity**

* **Best Case (O(n log n)):** Splitting takes O(n) time because each level scans the entire array to partition around the pivot, and since the pivot evenly splits the array into two halves, the recursion depth is O(log n), leading to a total time of O(n log n).
* **Average Case (O(n log n)):** Splitting still takes O(n) time, but since the pivot generally divides the array into reasonably balanced parts, the recursion depth remains O(log n), resulting in O(n log n) total time.
* **Worst Case (O(n²)):** Each partitioning step takes O(n) time, but when the pivot consistently selects the smallest or largest element, the array is reduced by only one element per step, leading to O(n) recursive levels and a total time of O(n²).

**Space Complexity**

Quicksort does not need extra memory to store elements, but it uses memory for recursive function calls and partitioning. The space complexity depends on how deep the recursion goes, which is based on how well the pivot splits the array.

* **Best & Average Case (O(log n))** – If the pivot splits the array evenly, the recursion only goes log n levels deep, so it uses O(log n) space in the call stack.
* **Worst Case (O(n))** – If the pivot is always the smallest or largest element, the recursion keeps going n levels deep, using O(n) space because each call adds to the stack.

Since the number of recursive calls decides both how much space and time Quicksort uses, space complexity follows a similar pattern to time complexity.

1. **Randomized Quicksort**

A computer screen with text

Description automatically generated

Randomized Quicksort works the same way as regular Quicksort but chooses the pivot randomly instead of using a fixed position like the first, last, or middle element. This random selection helps prevent the worst-case scenario, where a bad pivot choice leads to unbalanced partitions and O(n²) time complexity. By picking pivots randomly, the algorithm usually creates more balanced partitions, keeping the sorting process efficient with an expected time complexity of O(n log n). Although selecting a random pivot adds a small extra computation step, it significantly reduces the chances of poor performance, making Randomized Quicksort a better choice when the input order is unknown or could be adversarial. This ensures that the sorting process remains efficient and avoids cases where a fixed pivot might consistently lead to bad splits.

1. **Empirical Analysis:**

**A screenshot of a computer

Description automatically generated**

**Observations**

* **Random Data:** Both deterministic and randomized Quicksort performed similarly, achieving O(n log n) time. Randomized Quicksort introduced a slight overhead due to the additional step of selecting a random pivot.
* **Sorted Data:** Deterministic Quicksort was faster because selecting the middle element ensured balanced partitions and avoided worst-case behavior. Randomized Quicksort was slightly slower due to the extra step of choosing a random pivot but still maintained O(n log n) time.
* **Reverse-Sorted Data:** Similar to the sorted case, deterministic Quicksort performed efficiently as the middle pivot maintained balance. Randomized Quicksort was slightly slower due to the random pivot selection overhead.

**Conclusion**

Both deterministic and randomized Quicksort achieve O(n log n) time complexity in practice when a good pivot strategy is used. If deterministic Quicksort had chosen the first or last element as the pivot, sorted input would have resulted in O(n²) time complexity due to highly unbalanced partitions. Randomized Quicksort is particularly useful when the input distribution is unknown or potentially structured in a way that could lead to worst-case scenarios. However, when a well-chosen deterministic pivot (such as the middle element) is used, it can be slightly faster than randomizedQuicksort, as it avoids the minor overhead of random pivot selection.