**Overview**

Quicksort is one of the most famous sorting algorithms widely used in modern computing. It efficiently sorts large lists and influences how many real-world systems handle big data and fast lookups. In this assignment, I will implement a regular (deterministic) and a randomized version of Quicksort, analyze how they work in different situations, and see how randomizing the pivot helps avoid slowdowns.

**Implementation**

Here is how the basic Quicksort algorithm is implemented. Here the last element is used as the pivot.

A screen shot of a computer program

AI-generated content may be incorrect.

**Performance Analysis**

**Best Case:**

When the pivot splits the list evenly each time (such as with random data), Quicksort only needs about O(n logn) steps. This is because each time the list is split in half, there are log n levels of recursion.

**Average Case:**

Even with imperfect splits, Quicksort progresses on both sides at each step. The average-case time is also O(n logn) because the splits are reasonably balanced for most real-world data.

**Worst Case:**

If the pivot is always the smallest or largest item (like in a sorted or reverse-sorted array), every recursive call only removes one item, and the recursion depth becomes n. This leads to O(n2) time complexity, as every step repeatedly compares each item to the pivot.

**Why O (n log n) on Average:**

On average, Quicksort avoids worst-case behavior because the partitioning step usually creates subarrays smaller than the original. Additionally, the number of levels needed to break down the array is about logn. This means the total number of operations is about n log n.

**Space Complexity:**

Quicksort is an “in-place” algorithm, so it uses only a small, constant amount of extra memory for the sorting itself. However, it does use space for the call stack due to recursion. In the best and average case, this is about O(log n), but in the worst case (with bad pivots), it can use O(n) space.

**Randomized Quicksort**

Here is my code for **Randomized Quicksort**, which randomly chooses the pivot for each partition. This prevents the algorithm from continuously making the same poor choices when sorting data that happens to already be in order.

A screen shot of a computer program

AI-generated content may be incorrect.

Randomization helps by choosing the pivot element randomly. Randomization reduces the chance that the algorithm goes to the worst-case O(n2), even if the input has repeated patterns or is already sorted. In practice, randomized sort runs in O(n log n), making it very reliable for real-world data.

**Empirical Analysis**

When running both the version of Quicksort on arrays of different sizes and types including random, sorted and reverse sorted, results were different. Here is the sample table summarizing my results.

A screenshot of a computer program

AI-generated content may be incorrect.

**Observation**

Based on the results you can see in the table, both algorithms were fast when the data was random.  When the data is sorted and reverse-sorted for the sample sizes of 1000 and 2000, deterministic quicksort slowed drastically, resulting in a recursion error, while the randomized algorithm was still performant. While for the exact size of 500, though the deterministic algorithm did not error out, we can still see that it was much slower than the randomized version.  This table result matches the theoretical analysis that randomizing pivot points makes quick sort reliable, especially for larger data sets.

**Conclusion**

Quicksort is a fast and elegant algorithm, but its performance can suffer badly if the pivot choices are not random. Randomized Quicksort is a simple improvement that fixes this problem, making the algorithm efficient and reliable for a wide range of data. Understanding when and why to use randomized algorithms like this is an important skill for building robust, real-world software.

**GitHub**

Here is the GitHub link where the detailed code pasted above are present alongside the read me:

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison-Wesley. https://archive.org/details/ArtOfComputerProgrammingVol3/page/n1/mode/2up