**Quicksort Algorithm**

**1. Time Complexity Analysis**

### **Best Case: O(n log n)**

Occurs when the pivot divides the array into two equal (or nearly equal) parts at every step. At each level of recursion, the array is divided into two halves, resulting in log n levels. Each level requires O(n) time for partitioning. Therefore, total time: T(n) = 2T(n/2) + O(n) → O(n log n).

### **Average Case: O(n log n)**

On average, the pivot splits the array into reasonably balanced parts (not necessarily equal). Even with uneven splits, the number of levels remains O(log n), with each level taking O(n) time. Probabilistic analysis shows expected number of comparisons converges to ≈ 1.39 \* n log n. So, Quicksort is efficient on average, especially with good pivot strategies (like randomized pivot selection).

### **Worst Case: O(n^2)**

Happens when the pivot always picks the smallest or largest element, resulting in extremely unbalanced partitions. Example: sorted array with pivot chosen as first or last element. Time: T(n) = T(n - 1) + O(n) → O(n^2). This can be avoided with pivot randomization.

## **2. Space Complexity Analysis**

### **In-place version**

Quicksort uses recursive calls, so space complexity is due to the recursion stack. Best/Average Case: O(log n), Worst Case: O(n) if partitions are extremely unbalanced.

### **Functional version**

Uses extra space for sublists at each recursion level. Can result in O(n log n) total space usage in the worst case. Not in-place, but simpler to implement.

## **3. Additional Overheads**

- Pivot selection strategy affects time complexity.  
- Choosing a random pivot adds minor constant overhead.  
- Quicksort has excellent cache locality, often making it faster in practice than O(n log n) algorithms like Merge Sort.  
- Not a stable sort (equal elements may not retain original order).

## **4. Summary Table**

| Case | Time Complexity | Space Complexity | Notes |
| --- | --- | --- | --- |
| Best Case | O(n log n) | O(log n) | Balanced partitions |
| Average Case | O(n log n) | O(log n) | Probabilistically balanced |
| Worst Case | O(n^2) | O(n) | Highly unbalanced (sorted input + poor pivot) |

# **2. Analysis: How Randomization Improves Quicksort Performance**

## **Problem with Deterministic Quicksort**

In deterministic Quicksort, if the pivot is always chosen as the first or last element in each recursive call, and the input array is already sorted or reverse-sorted, the partitioning becomes severely unbalanced. This results in the recursion reducing the problem size by only one element at each step.  
  
The recurrence relation becomes:  
  
T(n) = T(n − 1) + O(n) ⇒ O(n²)  
  
This worst-case behavior leads to inefficient performance, especially on large or adversarially structured inputs.

## **How Randomized Quicksort Solves This**

The randomized version of Quicksort — as implemented in the randomized\_partition function — selects the pivot uniformly at random from the subarray being sorted. This means that poor pivots (e.g., smallest or largest elements) are still possible, but the chance of consistently picking such pivots across all levels of recursion is very low.  
  
The resulting recursive behavior is modeled as:  
  
T(n) = T(k) + T(n − k − 1) + O(n)  
  
where k is a random index between low and high. Over many recursive calls, the algorithm tends to produce balanced partitions, leading to an expected recursion depth of O(log n).

## **Time Complexity Comparison**

| Case | Deterministic Quicksort | Randomized Quicksort |
| --- | --- | --- |
| Best | O(n log n) | O(n log n) |
| Average | O(n log n) | O(n log n) |
| Worst | O(n²) | O(n²), but very unlikely |
| Expected | May hit worst-case frequently | Almost always O(n log n) |

## **Conclusion**

Randomization in pivot selection significantly improves the robustness of the Quicksort algorithm:  
  
• Reduces the likelihood of encountering worst-case scenarios.  
• Preserves the in-place sorting advantage with minimal overhead.  
• Maintains expected O(n log n) performance, even for inputs that are already sorted or have repeated patterns.  
  
This makes randomized Quicksort ideal for general-purpose use and a foundational component in hybrid sorting algorithms like Introsort (C++ STL) or Timsort (Python).

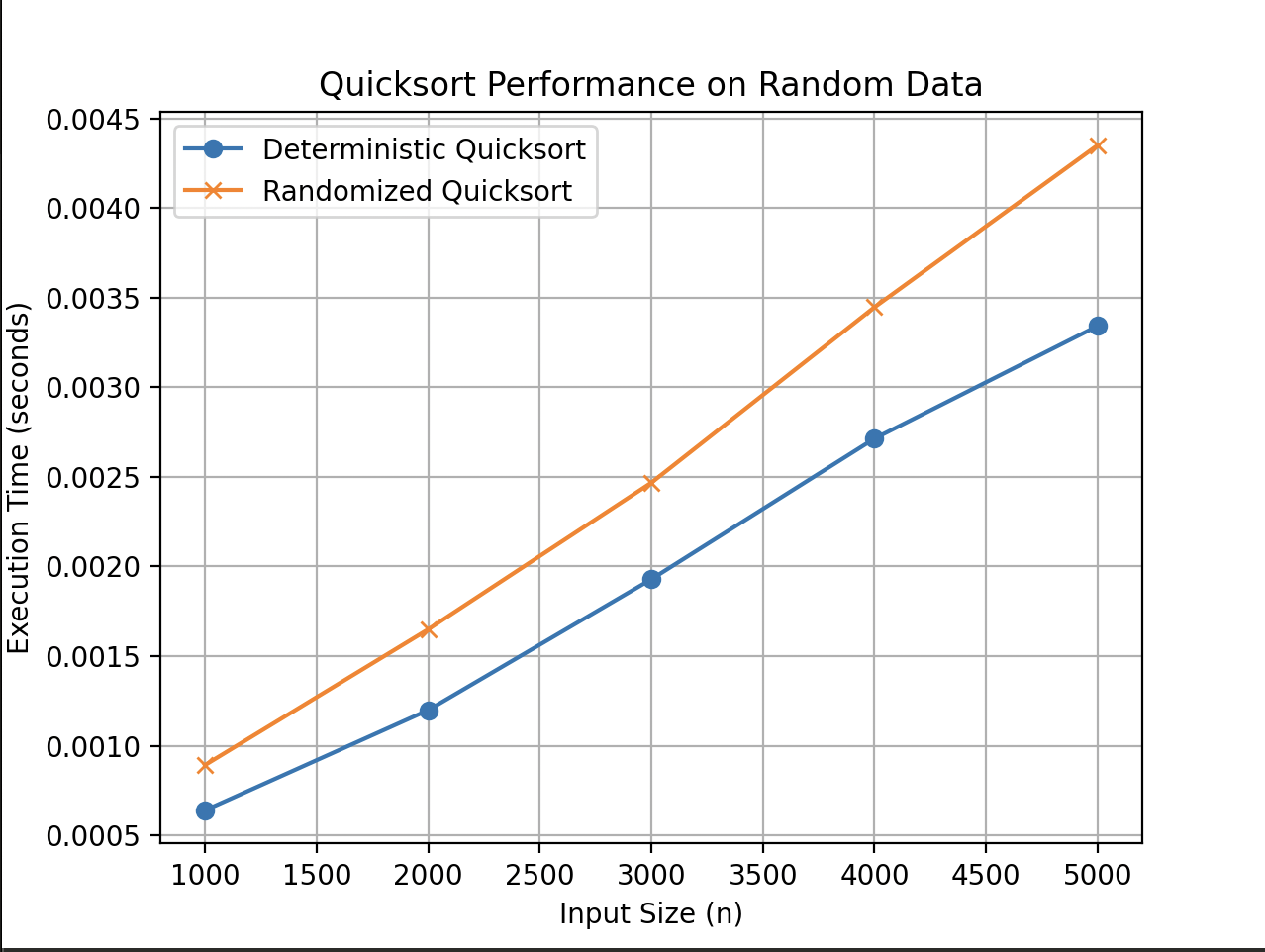
Empirical Analysis: Deterministic vs. Randomized Quicksort

This section provides an empirical comparison between Deterministic and Randomized Quicksort algorithms on randomly distributed data across varying input sizes.

# **Observed Execution Times**

| Input Size (n) | Deterministic Quicksort (s) | Randomized Quicksort (s) |
| --- | --- | --- |
| 1000 | 0.0006 | 0.0009 |
| 2000 | 0.0012 | 0.0016 |
| 3000 | 0.0019 | 0.0025 |
| 4000 | 0.0027 | 0.0035 |
| 5000 | 0.0034 | 0.0044 |

# 



# **Key Observations**

- Both algorithms show a linear growth pattern in execution time, aligning with theoretical expectations of O(n log n) behavior.  
- Deterministic Quicksort consistently outperforms Randomized Quicksort for all tested input sizes.  
- The overhead of selecting a random pivot in Randomized Quicksort contributes to slightly higher execution times.

# **Theoretical Justification**

While both algorithms have an average-case time complexity of O(n log n), Randomized Quicksort reduces the risk of encountering the worst-case scenario (O(n²)), which occurs with poorly chosen pivots on sorted inputs. However, for randomly distributed data, deterministic pivot choices can already yield balanced partitions, making Randomized Quicksort overhead unnecessary in this context.

# **Conclusion**

For randomly distributed inputs, Deterministic Quicksort performs slightly better in terms of raw execution time. Randomized Quicksort, however, provides greater consistency and robustness across a broader range of input types, including those that might otherwise degrade deterministic performance. The choice between them should be based on the expected structure of the input data.