**Randomized Quicksort — Time Complexity Analysis**

Randomized Quicksort improves over the deterministic version by choosing pivots **uniformly at random** from the subarray, which avoids consistently bad pivot choices (like always picking the smallest or largest element). On average, this produces balanced partitions and a recursion depth of . Each partition step processes all elements once, giving an overall expected time of .  
The 3-way partition used in this implementation further optimizes performance when **duplicates** are present by grouping equal elements together in a single partitioning step.  
However, in rare cases where the random pivot happens to be poor repeatedly, the algorithm may degrade to .

**Time Complexity Summary**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Remarks** |
| --- | --- | --- | --- | --- |
| **Randomized Quicksort (3-way)** | O(nlogn) | O(nlogn) |  | Efficient for duplicates; rare worst case |
| **Randomized Quicksort (2-way)** | O(nlogn) | O(nlogn) |  | Simpler but slower for repeated values |
| **Deterministic Quicksort (first pivot)** | O(nlogn) |  |  | Performs poorly on sorted/reverse inputs |

**Space Complexity**

* **Expected:** (recursion stack)
* **Worst case:**

**Comparison of Randomized and Deterministic Quicksort**

**Objective**

To empirically compare the running time of **Randomized Quicksort** (pivot chosen uniformly at random) with **Deterministic Quicksort** (first element as pivot) across different input types and array sizes.  
The goal is to observe how pivot choice and input order affect performance and verify the theoretical expectation for the randomized version.

**Experimental Setup**

* **Algorithms tested:**
  1. Randomized Quicksort (3-way partition)
  2. Deterministic Quicksort (first-element pivot)
* **Input sizes:** 200 – 15,000 elements
* **Distributions:** Random, Sorted, Reverse-sorted, Repeated values
* **Metric:** Mean running time over 3 trials for each case.
* **Observations**

| **Input Type** | **Randomized Quicksort** | **Deterministic (First Pivot)** | **Remarks** |
| --- | --- | --- | --- |
| **Random** |  |  | Both perform well; partitions are reasonably balanced. |
| **Sorted** |  |  | Deterministic picks the smallest pivot repeatedly, causing deep recursion. |
| **Reverse Sorted** |  |  | Same degradation due to unbalanced partitions. |
| **Repeated Elements** | (fast in practice) | but slower | Randomized 3-way partition groups duplicates efficiently. |

**Discussion**

The **randomized version** consistently maintained near-logarithmic growth across all input distributions, validating the theoretical analysis that expected time complexity is .  
The **deterministic first-pivot** algorithm matched on random inputs but degraded to on sorted and reverse-sorted arrays, where pivot choices were extremely unbalanced.  
In arrays with **many duplicates**, the 3-way randomized partition performed even better than standard 2-way partitioning, as it minimized unnecessary recursive calls on equal values.

Minor variations between empirical and theoretical results were due to:

* Random fluctuations in pivot selection,
* Python interpreter overhead, and
* System-level factors like caching and recursion cost.

**Hash Table Analysis**

In **simple uniform hashing**, each key is equally likely to be placed in any of the buckets.  
Let n be the number of elements and (load factor).

**Expected Time Complexity**

* **Insert:**
* **Search:**
* **Delete:**

When the load factor is small (close to 1), all operations run in **expected** time.

**Effect of Load Factor**

The load factor represents the **average number of elements per bucket**.

* A **low** means shorter chains → faster operations.
* A **high** leads to longer chains → slower performance.

To keep performance stable, most implementations maintain between **0.5 and 0.75**.

**Maintaining Low Load Factor**

To avoid long chains and high collision rates:

* **Dynamic resizing:** When , double the table size and rehash all keys.
* **Occasional shrinking:** When , halve the table size.
* **Universal hashing (used here):** Random coefficients minimize clustering and make collisions less likely.