# Average-Case Time Complexity Analysis of Randomized Quicksort

## Introduction

Randomized Quicksort is a sorting algorithm that selects a pivot element uniformly at random, partitions the array around the pivot, and recursively sorts the subarrays. Its average-case time complexity is known to be O(n log n). Below, we rigorously analyze this result using two methods: indicator random variables and recurrence relations.

## Method 1: Indicator Random Variables Approach

### Setup

Let the array have n distinct elements.  
Let T(n) be the expected number of comparisons Randomized Quicksort makes.  
Define an indicator random variable X\_{i,j} for each pair (a\_i, a\_j) where:  
X\_{i,j} = 1 if a\_i and a\_j are compared during execution, 0 otherwise.

### Total Comparisons

The total number of comparisons:  
T(n) = Σ\_{1 ≤ i < j ≤ n} X\_{i,j}  
  
Taking expectation:  
E[T(n)] = Σ\_{1 ≤ i < j ≤ n} E[X\_{i,j}] = Σ\_{1 ≤ i < j ≤ n} Pr(a\_i is compared with a\_j)

### Probability of Comparing Two Elements

Elements a\_i and a\_j are compared only if one of them is chosen as a pivot before any element between them in the sorted order. The probability that either a\_i or a\_j is the first pivot among the elements a\_i, a\_{i+1}, ..., a\_j is:  
Pr(X\_{i,j} = 1) = 2 / (j - i + 1)

### Summation Over All Pairs

Summing over all pairs:  
E[T(n)] = Σ\_{1 ≤ i < j ≤ n} 2 / (j - i + 1)  
  
Reindexing the sum by k = j - i + 1, the total expected comparisons are bounded by:  
E[T(n)] ≤ 2n Σ\_{k=1}^n 1/k = 2n H\_n = O(n log n)  
  
where H\_n is the n-th harmonic number, asymptotically log n.

## Method 2: Recurrence Relation Approach

### Formulating the Recurrence

At each step, the pivot divides the array of size n into two subarrays of sizes k and n - k - 1 for some k, where k is uniformly distributed over {0, 1, ..., n - 1}.  
  
The expected runtime satisfies:  
T(n) = (1/n) Σ\_{k=0}^{n-1} (T(k) + T(n - k - 1)) + cn  
  
where cn accounts for the partitioning step.

### Solving the Recurrence

Rewrite as:  
T(n) = (2/n) Σ\_{k=0}^{n-1} T(k) + cn  
  
Assuming T(k) ≤ a k log k (inductive hypothesis), we get:  
T(n) ≤ (2a / n) Σ\_{k=1}^{n-1} k log k + cn  
  
Approximating the sum by an integral:  
Σ\_{k=1}^{n-1} k log k ≈ ∫\_1^n x log x dx = (n^2 / 2) log n - (n^2 / 4) + C  
  
Substituting back:  
T(n) ≤ a n log n + (c - a/2) n  
  
By choosing a sufficiently large, the inequality holds, confirming T(n) = O(n log n).

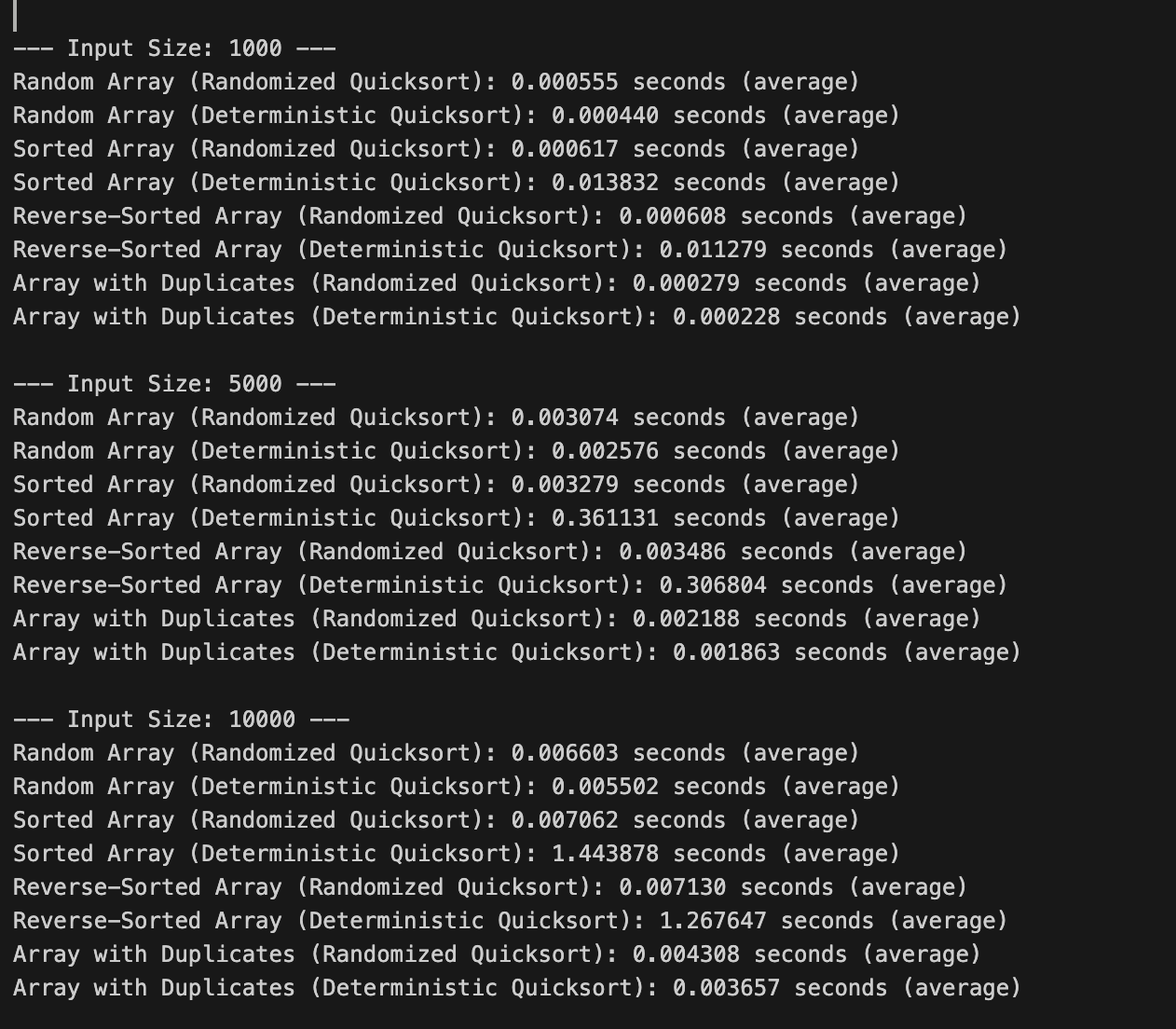
## Conclusion

Both methods demonstrate that the expected runtime of Randomized Quicksort is:  
  
T(n) = O(n log n)  
  
This is because:  
- Indicator variables show that each pair of elements is compared with probability inversely proportional to the size of the subarray they lie in.  
- The recurrence relation shows that, on average, the partition sizes balance out so that the recursion depth is O(log n) with O(n) work per level.

Empirical vs Theoretical Analysis of Randomized and Deterministic Quicksort

# Empirical Results Summary:

Results from running both the algorithms for diff input size.



- Random Array:  
 - Randomized Quicksort: Fast and scales well.  
 - Deterministic Quicksort: Slightly faster on small sizes.  
- Sorted Array:  
 - Randomized Quicksort: Consistently fast.  
 - Deterministic Quicksort: Very slow with large jump in time.  
- Reverse-Sorted Array:  
 - Randomized Quicksort: Consistently fast.  
 - Deterministic Quicksort: Very slow, similar to sorted array.  
- Array with Duplicates:  
 - Both perform well, deterministic slightly faster on small sizes.

# Theoretical Expectations:

- Randomized Quicksort: Average-case time complexity is O(n log n) for all input distributions due to random pivot selection which avoids worst-case splits.  
- Deterministic Quicksort (pivot as first element):  
 - Average-case O(n log n) for random inputs.  
 - Worst-case O(n^2) on already sorted or reverse sorted arrays, due to poor pivot choice causing unbalanced partitions.

# Detailed Discussion:

1. Random Arrays:  
 Both algorithms run close to O(n log n). Deterministic is slightly faster for small sizes due to less overhead.  
2. Sorted and Reverse-Sorted Arrays:  
 Deterministic quicksort suffers from worst-case O(n^2) behavior, while randomized quicksort maintains O(n log n) by balancing partitions on average.  
3. Arrays with Duplicates:  
 Both perform well, with deterministic sometimes slightly faster on small arrays because duplicates reduce recursion depth.  
  
The spikes in deterministic quicksort time for sorted and reverse sorted arrays clearly illustrate the worst-case scenario predicted by theory.

# Empirical and Theoretical Alignment:

- Big-O notation describes asymptotic behavior, ignoring constants and lower order terms which affect small input times.  
- On small inputs, overhead like recursion and random pivot selection can affect timing.  
- On large inputs, asymptotic differences dominate, highlighting randomized quicksort’s consistent performance.  
- Randomized quicksort’s results confirm its expected average O(n log n) complexity.  
- Deterministic quicksort’s poor performance on sorted inputs confirms its O(n^2) worst case.

# Summary and Recommendations:

- Randomized Quicksort is robust across input types due to pivot randomization.  
- Deterministic Quicksort with first element pivot is vulnerable to certain inputs and should be avoided in practice.  
- Using better pivot selection strategies (e.g., median-of-three) can improve deterministic quicksort.  
- Empirical results validate the theoretical analysis and highlight why randomization is preferred in practice.

# Hashing with Chaining

## 1. Expected Time Complexity Under Simple Uniform Hashing

Simple uniform hashing assumption:  
Every key is equally likely to be hashed to any of the available slots independently of other keys.

### Search Time

Successful search (key exists):  
Expected time is O(1 + α) where α = n/m is the load factor, n = number of elements, m = number of slots (buckets).  
Explanation: On average, you examine 1 slot plus the average length of the chain (for chaining), which is proportional to the load factor.

Unsuccessful search (key does not exist):  
Expected time is also O(1 + α) because you must scan the entire chain at the hashed slot and confirm the key is absent.

### Insert Time

Expected time is O(1) for inserting at the head or tail of a chain (typical implementation).  
However, to maintain performance, the load factor should be kept low to avoid long chains.

### Delete Time

Expected time is O(1 + α) since you need to locate the element in the chain before deleting it.

## 2. Impact of Load Factor (α = n / m)

Load factor α is the average number of elements per bucket.  
  
When α is small (<< 1):  
Chains are short; operations are close to O(1) constant time.  
  
When α increases:  
Chains get longer, and operations degrade toward O(α) linear time.  
  
Higher α means more collisions, more elements in each bucket chain.  
  
Typical practice:  
Keep α ≤ 0.7 for good performance.  
For open addressing (not chaining), α is kept even lower, around 0.5 or less.

## 3. Strategies for Maintaining Low Load Factor and Minimizing Collisions

### Dynamic Resizing (Rehashing)

When α exceeds a threshold (e.g., 0.7), resize the hash table:  
- Increase the number of buckets (usually doubling).  
- Pick a new prime number larger than the new size for the universal hash.  
- Re-insert all existing keys into the new larger table (rehashing).  
  
Benefits:  
- Keeps α low, maintaining expected O(1) operation times.  
- Reduces collision chains.

### Choosing a Good Hash Function

Universal hashing reduces the chance of collisions by randomly selecting hash parameters.  
Ensures no 'bad' key distribution causes worst-case performance.

### Collision Resolution Techniques

Chaining: Store all colliding elements in a linked list or dynamic array.  
Open addressing: Probe for next available slot (linear probing, quadratic probing, double hashing).  
Proper choice affects performance and load factor limits.

## Summary Table of Expected Times (Chaining)

| Operation | Expected Time (Simple Uniform Hashing) |
| --- | --- |
| Search (success) | O(1 + α) |
| Search (failure) | O(1 + α) |
| Insert | O(1) (amortized, ignoring resizing) |
| Delete | O(1 + α) |

## Final Notes

Load factor directly controls the average chain length in chaining and average probes in open addressing.  
Keeping the load factor low via dynamic resizing ensures that average operations remain efficient.  
Universal hashing helps randomize hash assignments, preventing clustering and uneven distributions.