Assignment 2: Understanding Algorithm Efficiency and Scalability

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

Date: 11/09/2025

Analysis of Randomized Quick sort:

This Quicksort implementation achieves high efficiency specifically in terms of time complexity by using two strategies. First, its randomized pivot selection virtually eliminates the $O(n^2)$ worst-case performance, which typically occurs on sorted or reversed data, ensuring an expected runtime of O (n log n). Second, it employs 3-way partitioning, a significant optimization for arrays with many duplicate elements. This method groups all items identical to the pivot into a central partition, which is then excluded from future recursive calls, avoiding the $O(n^2)$ degradation that standard 2-way partitioning faces in this scenario. The algorithm is also space-efficient, performing the sort in-place with an average call stack space of O (log n). This robust design inherently handles various edge cases: its base case manages empty and single-element arrays, while the random pivot and 3-way partitioning optimally address sorted/reversed arrays and duplicate values, respectively.

Coming to time complexity the average-case time complexity of Randomized Quicksort is O (n log n). This is because the random selection of a pivot makes it highly probable that the partitions will be reasonably balanced. While a single unlucky pivot choice could lead to a worst-case split, the average over all random choices results in an expected split that is good enough to achieve the O(n log n).

The total complexity is dependent on two factors one is Partitioning and the Comparisons. At each level of recursion the partition function does O(k) work on individual left and right subarray respectively of size k.The average case complexity 0(n log n) can be proved with two methods using Indicator Random Variable and Recurrence Relations, my analysis is using Recurrence Relations:

As mentioned, the partition function takes 0(n) time and in analysis, we are denoting it as constant c and E[T[n]] be the excepted running time for an array of size n. While partitioning of array of size n, the pivot selection is random and likely be the kth smallest element for any k from 0 to n-1 and subarrays are created as k and n-1-k sizes. The excepted time is the sum of the partitioning cost plus the expected time of the recursive calls, averaged over all possible pivot positions.

The actual equation for recurrence running time is $T(n) – 2T(n/2) + O(n)$ solving this gives $T(n) = O(n \log n)$.

$nE[T(n)] = (n+1) E[T(n-1)] + c(2n-1)$

Divide the entire equation by n(n+1) to set up a sum:

$E[T(n)/n+1 = E[T(n-1)]/n + c(2n-1)/n(n+1)$

Let $S(n) = E[T(n)]/n+1$ The recurrence becomes:

$S(n) = S(n-1) + c(2n-1)/n(n+1)$

$S(n) = S(0) + \sum_{i=1}^{n} c(2i-1)/i(i+1)$

Assuming $T(0)=0$ so $S(0)=0$

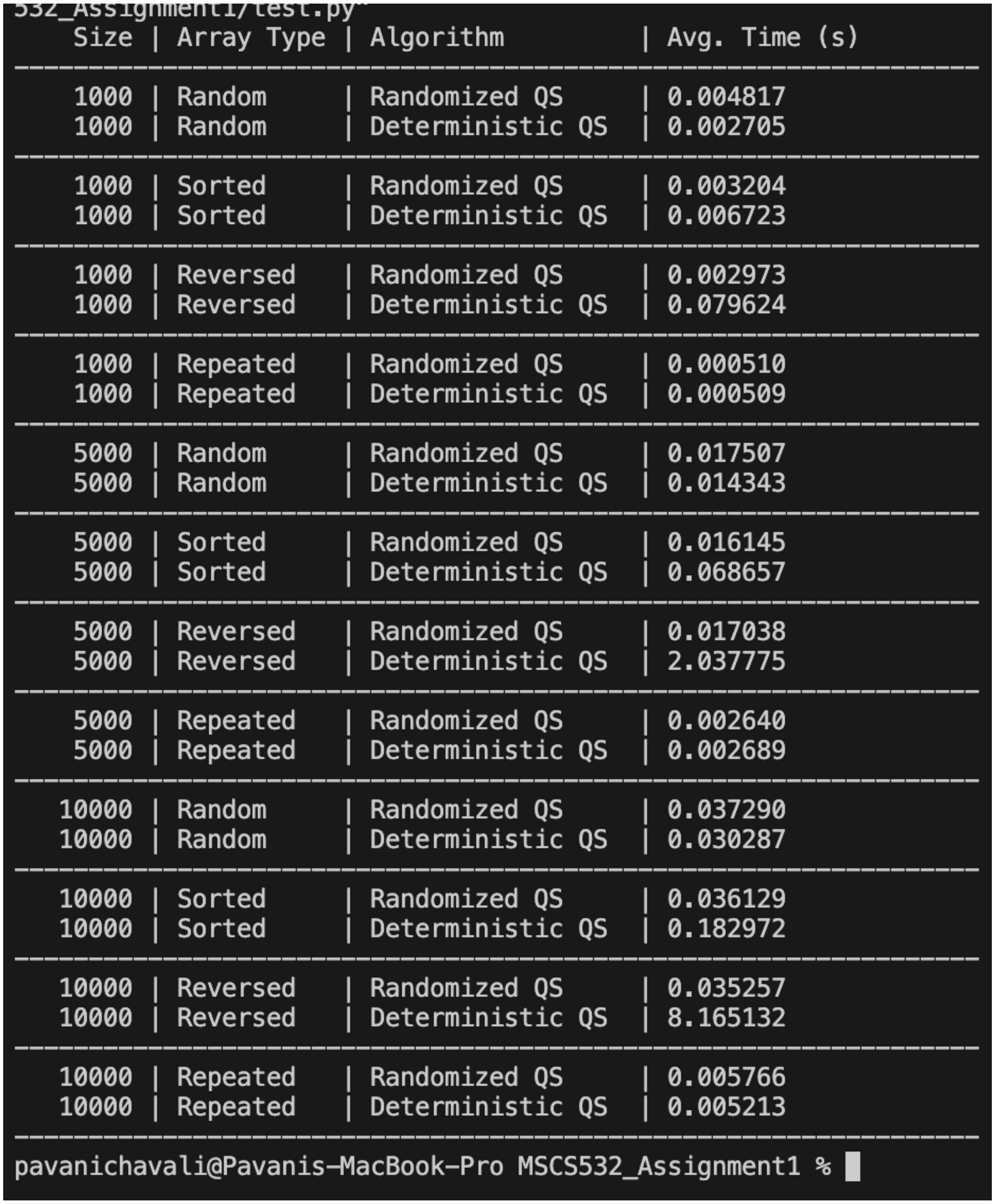
$C(2i-1)/i(i+1) < c(2i)/i(i+1) = 2c/i+1$ (since $H_{n+1} = O(\log(n+1))$)

$S(n) = O(\log n)$. Substituting back $S(n) = E[T(n)]/n+1$

$E[T(n)]/n+1 = O(\log n)$

$E[T(n)] = (n+1) . O(\log n) = O(n\log n) + O(\log n)$

Analysis: values for quick sort and randomized quick sort avg time



```
532_Assignment1/test.py"
    Size | Array Type | Algorithm          | Avg. Time (s)
------------------------------------------------------------------
    1000 | Random     | Randomized QS      | 0.004817
    1000 | Random     | Deterministic QS   | 0.002705
------------------------------------------------------------------
    1000 | Sorted     | Randomized QS      | 0.003204
    1000 | Sorted     | Deterministic QS   | 0.006723
------------------------------------------------------------------
    1000 | Reversed   | Randomized QS      | 0.002973
    1000 | Reversed   | Deterministic QS   | 0.079624
------------------------------------------------------------------
    1000 | Repeated   | Randomized QS      | 0.000510
    1000 | Repeated   | Deterministic QS   | 0.000509
------------------------------------------------------------------
    5000 | Random     | Randomized QS      | 0.017507
    5000 | Random     | Deterministic QS   | 0.014343
------------------------------------------------------------------
    5000 | Sorted     | Randomized QS      | 0.016145
    5000 | Sorted     | Deterministic QS   | 0.068657
------------------------------------------------------------------
    5000 | Reversed   | Randomized QS      | 0.017038
    5000 | Reversed   | Deterministic QS   | 2.037775
------------------------------------------------------------------
    5000 | Repeated   | Randomized QS      | 0.002640
    5000 | Repeated   | Deterministic QS   | 0.002689
------------------------------------------------------------------
   10000 | Random     | Randomized QS      | 0.037290
   10000 | Random     | Deterministic QS   | 0.030287
------------------------------------------------------------------
   10000 | Sorted     | Randomized QS      | 0.036129
   10000 | Sorted     | Deterministic QS   | 0.182972
------------------------------------------------------------------
   10000 | Reversed   | Randomized QS      | 0.035257
   10000 | Reversed   | Deterministic QS   | 8.165132
------------------------------------------------------------------
   10000 | Repeated   | Randomized QS      | 0.005766
   10000 | Repeated   | Deterministic QS   | 0.005213
------------------------------------------------------------------
pavanichavali@Pavanis-MacBook-Pro MSCS532_Assignment1 %
```

Overall, the Randomized Quicksort is far more robust. It delivers consistent, fast performance across all input types, confirming its expected O(n log n) time complexity.

Deterministic Quicksort is dangerously vulnerable. While slightly faster on random data, it suffers catastrophic, non-linear slowdowns on sorted and, most dramatically, on reverse-sorted arrays.

Analysis of Hash Table with chaining:

The important metric for a hash table is the load factor (λ), defined as:

$\lambda = n/m$ (where n = number of elements, m = number of slots)

Where each key is equally likely to land in any slot, which is Simple Uniform Hashing assumption, the load factor represents the average length of a chain.

All operations compute to a hash O(1) and then scan a chain. This gives a universal expected time complexity

Expected Time for Search, Insert and Delete: $O(1+\lambda)$

This formula directly shows how the load factor affects the performance:

Ideal (Low λ): If we keep λ small and constant ($\lambda=1$) the complexity is O(1).

Poor (High λ): If λ is large (meaning $n>>m$), the complexity becomes $O(\lambda)$ or O(n). The hash tables O(1) benefits is lost, and it performs no better than a simple list.

The primary strategy for a low load factor is dynamic resizing or releasing to ensure λ always stays in a constant, ideal range.

Define Thresholds: Max Threshold $\lambda>0.75$ The table is too full

Min Theshold $\lambda<0.25$ The table might be too empty and wasting space.

This is how the Resizing works as Table grows and shrinks, When the maximum threshold is hit, create a new, larger table and rehash all n elements into it. When the minimum threshold is hit, create a new, smaller table and rehash all elements. The process guarantees λ remains constant, so performance remains O (1). Resizing itself is an expensive, O(n) operation However, since it happens infrequently its high cost is averaged out over many cheap, O(1) operations. This makes the amortized cost of insertion and deletion O (1).

GithubLink:

https://github.com/PavaniChavali135/AlgoandDatastructures/tree/main/week_3/Asgmnt3