Richa Lahoti
005039141
MSCS-532-M20
Assignment 5

Implementation

Deterministic quicksort is implemented in the github repo under assignment5.py.

The code is in the github repo: https://github.com/rlahoti-cb/MSCS532_Assignment5

I will also quickly discuss the design choices made for the deterministic quicksort algorithm's implementation. The quicksort implementation allows supplying strategy which takes two values, random and middle. For deterministic quicksort, we choose the middle strategy that chooses the middle element as the pivot element. This deterministically chooses the pivot to implement the quicksort algorithm. This design choice allows us to minimize the amount of duplicated code since we can use the same quicksort algorithm for both random pivot and deterministic pivot (with the pivot selection helper function being the only difference).

Performance Analysis

1. Analysis of quicksort in worst, average, and best case

In this section, we will begin by providing a detailed analysis of the quicksort algorithm in the average, worst, and best cases.

We can begin with the worst case (O) scenario. The worst case occurs when we pick the worst pivot during the partition phase of the algorithm splitting the input array into two subarrays that we will recursively call quicksort on. In this case, choosing the worst pivot would mean splitting the input array (of size n) into two subarrays of size 1 and size n-1. This leads to the highest recursive depth called with the most minimal amount of work done at each level. This leads to a worst case performance of O(n^2).

In the average case, the pivot selection over the execution of the algorithm results in a combination of both good and bad pivots. In this case, this results in a performance of Θ(n log n). This can be observed in the real world by using a randomized pivot selection which will lead to both good and bad pivot selections over the course of the execution of the quicksort algorithm.

In the best case, the pivot selected at each recursive call will perfectly split the input array (size n) into two subarrays of equal size (n/2), This leads to the optimal performance of Ω(n log n).

As shown by the analysis of the runtime of the quicksort algorithm, the algorithm's performance is highly sensitive to the pivot selection. Pivot selection has a direct impact on the recursive depth of the algorithm and the work done therefore has a direct impact on the runtime of the algorithm.

2. Average time complexity vs worst time complexity of quicksort

We will now explain why the average time complexity of quicksort is O(n log n) and the worst case is O(n^2). In the worst case, we choose the worst pivot at every recursive layer. This leads to the algorithm splitting the input array of size n into two subarrays of size 1 and size n-1. This leads to n recursive levels with each level taking O(n) work. This leads to a worst case runtime of O(n^2). In the average case, we choose a random pivot which leads to a logarithmic reduction of the levels of work. This leads to log n levels that will occur. Each level takes O(n) work and therefore the average runtime of the quicksort algorithm is O(n log n).

3. Quicksort space complexity and additional overheads

In this section, we can discuss the space complexity and any additional overheads for the algorithm.

In terms of space complexity, there are many approaches that one can take when implementing the quicksort algorithm. In my approach and implementation, I choose to do in-place sorting which results in no additional space used by the algorithm. The algorithm updates elements in the input array in place allowing for an efficient usage of space. Although the algorithm does not use additional space to sort, the call stack does take space which in the average case takes O(log n) space since there are on average log n levels of recursion.

Randomized Quicksort

Randomized quicksort is implemented in the github repo linked below. This implementation of randomized quicksort is run by supplying the strategy argument as random. This randomly picks a pivot to partition the input array into subarrays for the recursive call.

https://github.com/rlahoti-cb/MSCS532_Assignment5

We can analyze from a theoretical perspective how randomized pivot selection reduces the likelihood of running into the worst case scenario of quicksort. From a theoretical perspective, the quicksort algorithm when choosing pivots at random enables the algorithm to usually avoid the worst case scenario. Since pivots are chosen at random, there is a lower likelihood that we run into choosing the worst pivot during each pivot selection leading to the worst case runtime. With random pivot selection, there will be a combination of both good and bad pivots chosen over all recursive calls which usually prevents the worst case runtime.

Empirical Analysis

In this section, we will run an experiment to capture data that will allow us to observe the performance of deterministic and randomized quicksort on varying input sizes (1000, 10000, 100000 elements) and varying input types (random data, sorted data, reversed sorted data). This will allow us to compare the real world performance of these algorithms and input data types with the theoretical expectations.

The results are displayed as a table. The raw data from the execution of the experiment is stored in the results.txt file in the code repository.

Pivot Selection	Data Size	Data Type	Time Taken (s)
random	1000	random	0.002894163
random	1000	sorted	0.002588034
random	1000	reversed sorted	0.002653122
random	1000	repeated random	0.021425009
deterministic	1000	random	0.001788139
deterministic	1000	sorted	0.001271725
deterministic	1000	reversed sorted	0.001579762
deterministic	1000	repeated random	0.019902945
random	10000	random	0.030177832
random	10000	sorted	0.028097153
random	10000	reversed sorted	0.029265165
random	10000	repeated random	1.953289986
deterministic	10000	random	0.023551941
deterministic	10000	sorted	0.017995119
deterministic	10000	reversed sorted	0.019322872
deterministic	10000	repeated random	1.885805130
random	100000	random	0.313586950
random	100000	sorted	0.302184820
random	100000	reversed sorted	0.291482925
random	100000	repeated random	194.269221067
deterministic	100000	random	0.286261082
deterministic	100000	sorted	0.210105896
deterministic	100000	reversed sorted	0.211861134
deterministic	100000	repeated random	183.298515081

Using the data captured from the experiment, we can compare the theoretical expectations with the empirical results.

We can see that the performance with the same input data size and input data types, deterministic and random seem to perform relatively the same. For example, we can compare random pivot selection with deterministic pivot selection across input data sizes for only random data. We see that with 1000 elements, random pivot selection takes 0.00289s and deterministic takes 0.00179s. We see that with 10000 elements, random pivot selection takes .03s and deterministic takes 0.024s. We see that with 100000 elements random pivot selection takes 0.314s and deterministic takes .286s. This clearly shows that deterministic pivot selection performs the same if not slightly better than random pivot selection. This real world performance does not align with the theoretical expectation that on the same input size, input data type that random pivot selection should perform better than deterministic pivot selection. I was very surprised by this result.

We can also run a comparison for the sorted data type which should theoretically perform better with random pivot selection than deterministic pivot selection. We can compare random pivot selection with deterministic pivot selection across input data sizes for only sorted data. We see that with 1000 elements, random pivot selection takes 0.00259s and deterministic takes 0.00127s. We see that with 10000 elements, random pivot selection takes .028s and deterministic takes 0.018s. We see that with 100000 elements random pivot selection takes 0.3s and deterministic takes .21s. Again, I am surprised to see that the performance of the deterministic pivot selection performs better than the random pivot selection on an input data type of sorted data.

In summary, we see that the real world performance does not match the theoretical performance. This was highly surprising for me. There could be a multitude of reasons which could explain this. This could be due to how my machine was caching data which I did not control for in my experiment. This could also be due to additional processes running that were impacting performance of the algorithm while it was executing (another aspect that I did not control for).

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

GeeksforGeeks. (2023, September 14). Quicksort using random pivoting. https://www.geeksforgeeks.org/dsa/quicksort-using-random-pivoting/

GeeksforGeeks. (2025, April 17). Quick Sort. https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/