MSCS-532-Assignment4

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Quicksort Algorithm: Implementation, Analysis, and Randomization

1. Implementation

Python implementation of the Quicksort algorithm is given here:

https://github.com/sanspokharel26677/MSCS-532-Assignment5/blob/main/quicksort.py

https://github.com/sanspokharel26677/MSCS-532-Assignment5

2. Performance Analysis

Quicksort is a very efficient sorting algorithm that uses the divide-and-conquer strategy. Let's break down its time complexity for the best, average, and worst cases.

1. Best-Case Time Complexity: O(n log n)

- The best case happens when the pivot divides the array into two nearly equal halves at every step. This way, the array keeps getting split evenly down the middle.
- In each split, the work needed to partition the array is linear, or $O(n)$, because every element is compared to the pivot.
- The process of splitting the array continues until we get subarrays of size 1, which happens in about $\log n$ levels of recursion (because halving an array repeatedly results in a logarithmic number of steps).
- So, the best-case time complexity is $O(n \log n)$, where $O(n)$ *is the time to partition, and* $\log n$ is the number of recursive calls.

2. Average-Case Time Complexity: O(n log n)

- The average case for Quicksort assumes that the pivot divides the array into two parts that are not necessarily equal but still reasonably balanced. For simplicity, let's assume on average the pivot splits the array into a ratio of 1/4 and 3/4.

- Again, partitioning the array takes *O(n)* time because each element is compared to the pivot.
- Now, since the array gets roughly halved each time, the number of recursive levels is around $\log n$ However, because the splits aren't perfect, this analysis gets a bit more complex but stays in the $O(n \log n)$ range.
- We can think of it like this: the array is still shrinking quickly enough with each recursive call that the total number of operations ends up being proportional to $n \log n$. Mathematically, if we sum up the work done at each level, it approximates to $O(n \log n)$.
- **Why** $O(n \log n)$**?**:
    - At each level of recursion, we do *O(n)* work to partition the array.
    - Since the array is split in a way that results in about $\log n$ levels, the total time is $O(n) \times O(\log n) = O(n \log n)$.

3. Worst-Case Time Complexity: O(n^2)

- The worst case occurs when the pivot chosen is always the smallest or largest element in the array. This happens if the array is already sorted (or reverse sorted) and the pivot is chosen poorly, such as always picking the first or last element.
- When this happens, the array isn't divided into two smaller subarrays. Instead, one subarray has nearly all the elements, and the other has none.
- For example, if the pivot is always the smallest element, the recursive call only reduces the problem size by 1 at each step.
- In this scenario, the partitioning step still takes *O(n)* time, but instead of having $\log n$ levels of recursion, we end up with n *levels.*
- **Why** $O(n^2)$**?**:
    - At the first level, we do $O(n)$ work.
    - At the second level, we do $O(n-1)$ work, then $O(n-2)$), and so on until we reach $O(1)$.
    - If we sum this series, $n+(n-1)+(n-2)+\ldots+1$, it results in $O(n^2)$

Space Complexity and Overheads of Quicksort

Alright, let's talk about the space complexity of Quicksort and any other bits and pieces that might add to its overhead.

1. Space Complexity

- The main thing to consider for Quicksort's space complexity is how it handles the recursion. Quicksort is an **in-place** sorting algorithm, which means it doesn't need extra

space proportional to the input size for sorting. It swaps elements directly within the array.

- **Auxiliary Space (In-Place Nature):**
  - Since Quicksort rearranges elements in the original array without using an extra data structure (like a copy of the array), its **auxiliary space** usage is generally low. In the best and average cases, the extra space mainly comes from the recursive calls.
- **Recursive Call Stack:**
  - Each recursive call adds a new layer to the call stack. In the best and average cases, the depth of the recursion is $\log n$, so the space complexity due to the call stack is $O(\log n)$.
  - In the worst case, if the array is already sorted and we pick the worst pivot (like the first element every time), the depth of the recursion can go up to n. This means the space complexity can be as bad as $O(n)$ the case because each recursive call has to be stored on the stack.

2. Additional Overheads

- **Partitioning Overhead:**
  - Even though Quicksort is in-place, there is some overhead involved in the partitioning step. This step involves swapping elements and comparing them with the pivot. While this doesn't require extra space, it does involve some computational overhead.
- **Pivot Selection:**
  - The method used to select the pivot can add a bit of complexity. For example, using the "median of three" method (choosing the median of the first, middle, and last elements) adds a few extra steps to the algorithm, but this overhead is generally negligible compared to the sorting process itself.
- **Performance Tweaks:**
  - Many practical implementations of Quicksort switch to simpler algorithms like Insertion Sort for small subarrays (usually of size less than 10) because they perform better in those cases. This switch introduces a bit of overhead but improves overall performance.

Summary

So, Quicksort is pretty efficient in terms of space in most scenarios, but it can hit that worst-case space complexity if the pivot choices are poor and the recursion goes too deep.

3. Randomized Quicksort

How Randomization Affects Quicksort Performance

Alright, so let's dive into how randomization actually helps improve Quicksort and why it's a big deal.

1. Why Randomization Helps

- The main reason Quicksort can hit that nasty $O(n^2)$ worst-case time complexity is due to poor pivot selection. If we're always picking the first or last element as the pivot and the array is already sorted or nearly sorted, Quicksort ends up making very uneven splits. This means we're not really dividing the array into two smaller parts but instead just peeling off elements one by one, which leads to way too many recursive calls.
- Randomization tackles this problem head-on by making the pivot selection unpredictable. By choosing a pivot randomly, we're basically saying, "Let's avoid being predictable in how we split the array." This randomness dramatically reduces the chance that we'll consistently pick a bad pivot and end up with those awful splits.

2. Reducing the Worst-Case Likelihood

- When we choose the pivot randomly, the odds of hitting the worst-case scenario (where the pivot is always the smallest or largest element) drop significantly. For example, in a large array, the chance of randomly picking the worst possible pivot every time is incredibly slim.
- Let's break it down a bit:
    - With a random pivot, each element in the array has an equal chance of being selected as the pivot.
    - Even if we end up picking a "bad" pivot once or twice, the randomness makes it unlikely that this will happen consistently across all recursive calls. Over many calls, we're likely to get a mix of good and not-so-good pivots, leading to more balanced splits on average.
- This randomness makes Quicksort's performance much more stable. Instead of worrying about worst-case scenarios, we can almost always count on Quicksort to run in its average time complexity of $O(n \log n)$, which is pretty efficient.

3. Why This Matters in Practice

- In real-world scenarios, data can have all sorts of patterns. Some datasets might be partially sorted, have a lot of duplicate elements, or have other quirks. If we're always picking the first or last element as the pivot, we're more likely to run into trouble with such data.
- By randomizing the pivot, Quicksort becomes more versatile and better suited to handle different kinds of input without degrading to $O(n^2)$ performance.

So, randomizing the pivot in Quicksort is a neat trick to get the most out of this algorithm, keeping it fast and efficient in most real-world scenarios.

4. Empirical Analysis

After running the code for the different kinds of quicksort algorithms, we were able to get some data related to time taken by those algorithms on different sizes of datasets. Below data is a sample data that I got after running which might be different depending on the computer specifications. Image of the following data has also been generated by the program implemented. Based on the results obtained here, we can observe the performance of both deterministic and randomized Quicksort across different input sizes and distributions. Here's a breakdown of the comparison:

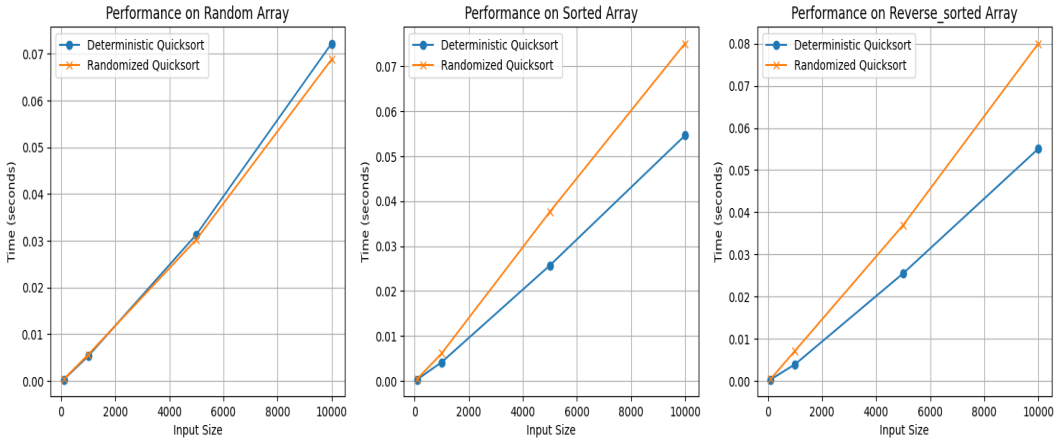
```
------------------------------------------------------------
Comparison and visulization of data
Input size: 100
  Random array - Deterministic: 0.000329s, Randomized: 0.000384s
  Sorted array - Deterministic: 0.000344s, Randomized: 0.000468s
  Reverse_sorted array - Deterministic: 0.000345s, Randomized: 0.000458s

Input size: 1000
  Random array - Deterministic: 0.005316s, Randomized: 0.005644s
  Sorted array - Deterministic: 0.004174s, Randomized: 0.006114s
  Reverse_sorted array - Deterministic: 0.003922s, Randomized: 0.007133s

Input size: 5000
  Random array - Deterministic: 0.031343s, Randomized: 0.030279s
  Sorted array - Deterministic: 0.025699s, Randomized: 0.037619s
  Reverse_sorted array - Deterministic: 0.025509s, Randomized: 0.036969s

Input size: 10000
  Random array - Deterministic: 0.072249s, Randomized: 0.068854s
  Sorted array - Deterministic: 0.054627s, Randomized: 0.075108s
  Reverse_sorted array - Deterministic: 0.055083s, Randomized: 0.080050s

Graph has been saved as quicksort_performance_comparison.png
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week4/MSCS-532-Assignment5$
```



Observation for random array:

For random arrays, both versions of Quicksort perform similarly, with the deterministic version being slightly faster for smaller input sizes. However, as the input size grows, the randomized version either matches or slightly outperforms the deterministic version. This aligns with the theoretical average-case time complexity of $O(n \log n)$ for both versions.

Observation for sorted arrays:

Now, this is interesting. On sorted arrays, the deterministic version is consistently faster than the randomized one. We might expect the deterministic version to struggle here since sorted input can be its worst enemy (if the pivot is chosen poorly), but because we're using the middle element as the pivot, it avoids that worst-case $O(n^2)$ behavior. The randomized version, on the other hand, has a bit more overhead because it randomly picks pivots, leading to slightly higher times.

Observation for reverse sorted array:

Similar story here with the reverse-sorted arrays. The deterministic version, using the middle pivot, performs slightly better than the randomized version. The randomized Quicksort has a bit more work to do with its random pivot selection, leading to higher execution times.

Discussion and Theoretical Analysis

Here's how these results line up with what we expected:

Random Arrays:

- The theory says that both deterministic and randomized Quicksort should be $O(n \log n)$ on average, and that's what we're seeing here. Both versions perform similarly, but the randomized version can slightly pull ahead on larger inputs. That's because its random pivot selection helps avoid any tricky input patterns that might mess with deterministic Quicksort.

Sorted and Reverse-Sorted Arrays:

- Here's where things get a bit surprising. We might think deterministic Quicksort would fall apart on sorted data (potentially hitting $O(n^2)$, but since we're using the middle element as the pivot, it sidesteps that issue. It actually performs pretty well on these inputs.
- The randomized version, meanwhile, introduces some extra overhead by picking random pivots, which can result in less-than-ideal splits, hence the slightly longer times. But it's designed to avoid the consistent worst-case scenarios that could happen with a poor pivot choice, making it a safer bet in less controlled environments.

Conclusion:

- **Random Input:** Both versions perform well and pretty much as expected.
- **Sorted and Reverse-Sorted Input:** The deterministic version does surprisingly well thanks to the middle pivot strategy. The randomized version has a bit more overhead but provides a safeguard against worst-case scenarios in general use.
- **Real-World Takeaway:** Using the middle element as a pivot in deterministic Quicksort can be a clever trick for structured data like sorted arrays. But randomized Quicksort is more flexible and generally avoids the pitfalls that could lead to $O(n^2)$ behavior.

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Motwani, R., & Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.