MSCS-532-Assignment6-Deterministic and Randomized Selection

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Implementation and Analysis of Selection Algorithms

1. Implementation

Python implementation of Deterministic algorithm for selection in worst-case linear time and Randomized algorithm for selection in expected linear time is given here:

https://github.com/sanspokharel26677/MSCS-532-Assignment6/blob/main/medians.py
https://github.com/sanspokharel26677/MSCS-532-Assignment6

2. Performance Analysis

Let's break down the time complexity for both the deterministic and randomized selection algorithms.

Deterministic Selection Algorithm (Median of Medians)

So, the deterministic algorithm is a bit more predictable since it guarantees *O(n)* time in the worst case. Here's why:
Dividing into groups: It splits the array into small groups of 5, sorts each group, and picks the median of each. Sorting these small groups takes *O(n)* time overall because sorting a group of 5 is constant, and we're just doing this for the whole array.
Finding the median of medians: Next, the algorithm takes these medians and recursively finds the median of that smaller set. This step also takes *O(n)* because it reduces the problem size pretty efficiently.
Partitioning: After that, the algorithm uses the median of medians to partition the array. Since we're always guaranteed a good split (like, at least 30% of the elements are smaller or larger than the pivot), this part is where the magic happens. It ensures the recursive steps work on progressively smaller portions of the array, and each partition is still *O(n)*.

So, even in the worst case, it's doing a pretty controlled partition and reduces the array size efficiently, which is why we get that sweet *O(n)* worst-case guarantee.

Randomized Selection Algorithm (Randomized Quickselect)

Now, the randomized algorithm (Quickselect) is kind of like gambling a bit, but with good odds! It only *expects O(n)*, but doesn't guarantee it.
Partitioning: It randomly picks a pivot and partitions the array around it. If we're lucky, the pivot will split the array into two equal halves (or close enough), and this partitioning takes *O(n)* in each step.
Recursive call: After partitioning, the algorithm recursively deals with one of the halves. On average, this will be half the array, meaning the recursion handles *n/2n/2*n/2 elements. So, in expectation, the recursion shrinks the problem quickly enough, giving us an expected *O(n)* overall.
However, since the pivot is random, we might occasionally get unlucky and keep picking bad pivots (like the largest or smallest element), leading to a worst-case scenario of $O(n^2)$. But generally, the random pivot works in our favor, which is why the average time is still linear.

Space Complexity and Overheads

Deterministic Algorithm:

Space complexity: This one uses more space because it has to store the medians of the groups (we're working with extra arrays here), which gives it *O(n)* space complexity.
Recursive depth: Since it keeps dividing the problem but does so efficiently, the recursive depth is $O(\log n)$, but that extra storage for medians means it's still *O(n)* space overall.

Randomized Algorithm:

Space complexity: This one's leaner! Since the partitioning is done in-place, the only space overhead comes from the recursion stack, and on average, the recursion depth is $O(\log n)$. So the space complexity is just $O(\log n)$, which is way smaller than the deterministic one.
Overhead: There's some overhead from generating random numbers to pick the pivot, but that's tiny compared to everything else.

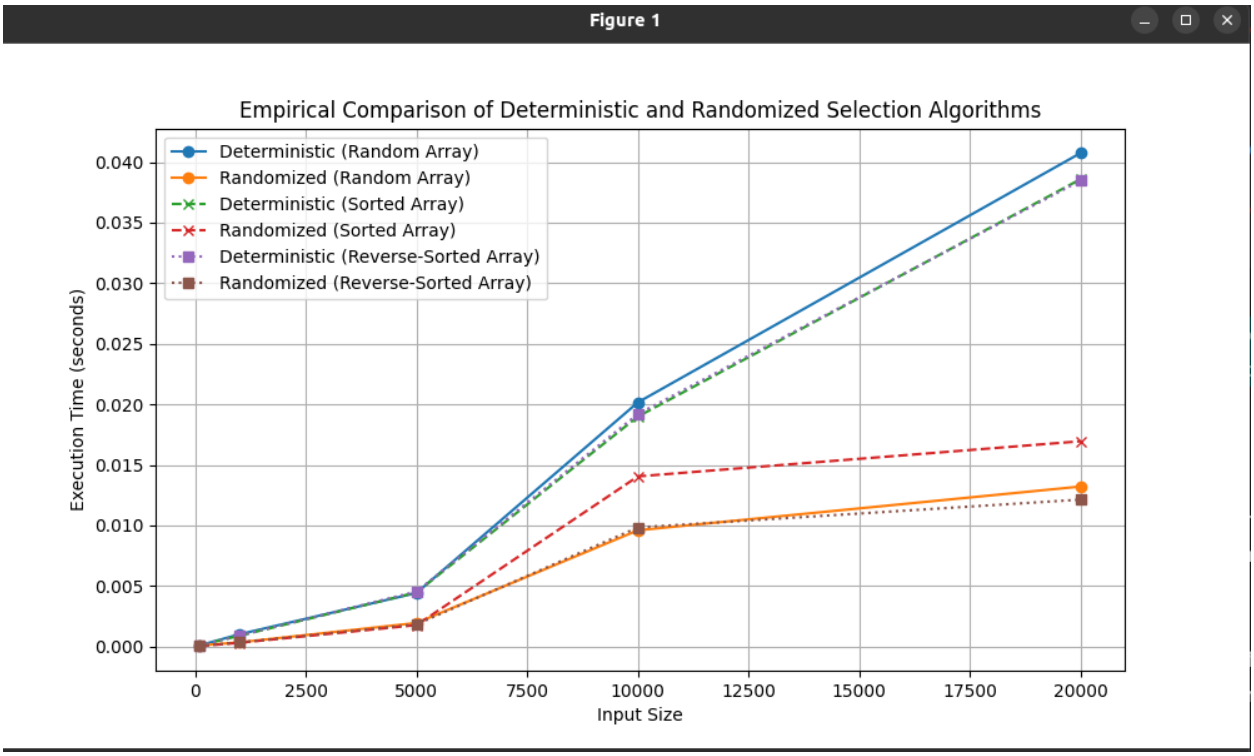
To sum it up:

Deterministic (Median of Medians): Guarantees *O(n)* time but uses more space *O(n)*.
Randomized (Quickselect): Has *O(n)* expected time and is more space-efficient with $O(\log n)$ space but can hit $O(n^2)$ in unlucky cases.

3. Empirical Analysis

Below is the discussion of the observed results and relating them to the theoretical analysis of both the deterministic (Median of Medians) and randomized (Quickselect) algorithms. For this, I ran the python code which I have also submitted and got some output. The output might be different while running from different machines due to their capabilities.

```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week5/MSCS-532-Assignment6$ python3 new.py
The 4th smallest element using deterministic implementation is 7
The 4th smallest element using randomized implementation is 7

Comparison of Deterministic and Randomized Algorithms:

Input Size: 100
Deterministic - Random: 0.000088s, Sorted: 0.000096s, Reverse: 0.000100s
Randomized - Random: 0.000085s, Sorted: 0.000032s, Reverse: 0.000056s

Input Size: 1000
Deterministic - Random: 0.001009s, Sorted: 0.000885s, Reverse: 0.000897s
Randomized - Random: 0.000340s, Sorted: 0.000305s, Reverse: 0.000341s

Input Size: 5000
Deterministic - Random: 0.004421s, Sorted: 0.004468s, Reverse: 0.004540s
Randomized - Random: 0.001945s, Sorted: 0.001766s, Reverse: 0.001829s

Input Size: 10000
Deterministic - Random: 0.020159s, Sorted: 0.018986s, Reverse: 0.019195s
Randomized - Random: 0.009595s, Sorted: 0.014046s, Reverse: 0.009829s

Input Size: 20000
Deterministic - Random: 0.040753s, Sorted: 0.038593s, Reverse: 0.038471s
Randomized - Random: 0.013211s, Sorted: 0.016948s, Reverse: 0.012129s
```



Observed Results and Analysis

For Input Size 100:

- Both algorithms are neck and neck here. The randomized algorithm is just a little faster, especially on sorted and reverse-sorted arrays. Pretty much what we'd expect for smaller inputs—no real surprises.

For Input Size 1000:

- Now we start seeing a bit more of a difference. The randomized algorithm is clearly faster, especially with sorted arrays, which it handles really well. The deterministic one is a bit slower but consistent across all array types (random, sorted, reverse-sorted).

For Input Size 5000:

- The gap is growing! The deterministic algorithm slows down slightly but keeps its steady performance across all types of inputs. On the other hand, the randomized algorithm is still faster overall but handles sorted arrays a bit better than random or reverse-sorted.

For Input Size 10000:

- The deterministic algorithm's time is climbing as expected, but it's still fairly even across all input types. The randomized algorithm is faster again, but you can see that it struggles a bit with sorted arrays, causing a jump in time.

For Input Size 20000:

- By this point, the deterministic algorithm is slower but stays consistent (whether it's random, sorted, or reverse-sorted, the times don't change much). The randomized algorithm, however, spikes with sorted arrays but still beats the deterministic one on most counts.

Relating to the Theory:

Deterministic Algorithm:

- **Theory says**: Worst-case linear time, $O(n)$, always.
- **What we see**: The deterministic algorithm behaves exactly like the theory predicts. Its time is almost identical whether the array is sorted, random, or reversed. This is because the median of medians approach guarantees good pivots, keeping everything linear—even in the worst cases (like sorted arrays).

Randomized Algorithm:

- **Theory says**: *O(n)* expected time, but can hit $O(n^2)$ in the worst case.
- **What we see**: The randomized algorithm generally performs better, especially with smaller inputs and random arrays. But, as predicted, it can struggle a bit with sorted arrays, where the pivot selection can sometimes lead to less balanced splits. This explains the jumps in time on sorted data as the input size grows.

So, in short: the randomized algorithm is faster on average, but the deterministic one is more predictable and steady, handling even the worst-case inputs without much fluctuation.

4. References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., & Tarjan, R. E. (1973). *Time bounds for selection*. Journal of Computer and System Sciences, 7(4), 448-461. https://www.sciencedirect.com/science/article/pii/S0022000073800339?via%3Dihub

Hoare, C. A. R. (1961). *Algorithm 65: Find*. Communications of the ACM, 4(7), 321-322.
https://dl.acm.org/doi/10.1145/366622.366647

Motwani, R., & Raghavan, P. (1995). *Randomized algorithms*. Cambridge University Press.