Implementation and Analysis of Selection Algorithms

This section provides a detailed theoretical analysis of the time and space complexity for the deterministic (Median of Medians) and randomized (Quickselect) selection algorithms.

Time Complexity

Deterministic Algorithm (Median of Medians)

The Median of Medians algorithm is specifically designed to guarantee a linear-time performance in the worst case.

- **Worst-Case Time Complexity:** O(n)
  The algorithm's brilliance lies in its pivot selection strategy, which ensures that the pivot is always "good enough." A good pivot is one that partitions the array into reasonably balanced subarrays, avoiding the worst-case scenarios seen in a naive Quickselect.
  The process guarantees that the chosen pivot is greater than at least $3(n/10 - 2)$ elements and smaller than at least $3(n/10 - 2)$ elements. For large n, this means the pivot is roughly in the 30th-70th percentile range. This prevents highly unbalanced partitions.
  The time complexity can be described by the recurrence relation:
  $T(n) \leq T(n/5) + T(7n/10) + O(n)$
  - $T(n/5)$: The time for the recursive call to find the median of the medians from the n/5 chunks.
  - $T(7n/10)$: In the worst case, the next recursive call will be on a subarray of size at most 7n/10.
  - $O(n)$: The time required for partitioning the array around the pivot and for creating the initial chunks and their medians.

Since $n/5 + 7n/10 = 9n/10 < n$, the work done at each level of recursion decreases geometrically. The sum of this series is a constant multiple of n, proving the worst-case linear time complexity of O(n).

Randomized Algorithm (Quickselect)

The Randomized Quickselect algorithm is simpler and often faster in practice, but its performance guarantee is probabilistic.

- **Expected (Average-Case) Time Complexity:** O(n)
  The key is the random pivot selection. On average, a randomly chosen pivot is expected to be near the true median. This means it will partition the array into two roughly equal-sized subarrays.

The recurrence for the average case is:
$T(n) \approx T(n/2)+O(n)$
- $T(n/2)$: The recursive call on one of the two partitions (on average, half the size).
- $O(n)$: The time for partitioning.

This recurrence solves to $O(n)$. The total work is $n+n/2+n/4+...$, which is a geometric series that sums to $2n$.

- **Worst-Case Time Complexity:** $O(n^2)$
  The worst case occurs if the random pivot selection consistently chooses the smallest or largest element in the current subarray. This forces the algorithm to recurse on a subarray of size $n-1$.
  The recurrence for the worst case is:
  $T(n)=T(n-1)+O(n)$

  This leads to a total time of $n+(n-1)+(n-2)+...+1$, which is an arithmetic series that sums to $n(n+1)/2$, resulting in a $O(n^2)$ complexity. While theoretically possible, the probability of this happening with a truly random pivot is extremely low, especially for large arrays.

Space Complexity and Overheads

- **Median of Medians:** The Python implementation uses list comprehensions (low, high, pivots, chunks, medians) at each step, which create new lists. This means the space complexity is dominated by the storage of these temporary lists, making it $O(n)$. The recursion depth also contributes, but it's less than the space used by the lists. The primary overhead is the complex pivot-finding mechanism, which involves multiple passes and a recursive call just to select the pivot.
- **Randomized Quickselect:** Similar to the deterministic version, our implementation uses list comprehensions for partitioning, leading to a space complexity of $O(n)$. The recursion depth is $O(\log n)$ on average but can be $O(n)$ in the worst case. The overhead is minimal, involving only a random number generation and a single pass for partitioning.

Empirical Analysis

This section analyzes the empirical results from the script, relating them to the theoretical concepts discussed above.

Comparison on Random Data

The performance table you provided clearly shows that for randomly distributed data, **Randomized Quickselect is significantly faster than Median of Medians**.

Input Size	Deterministic Time (s)	Randomized Time (s)
1000	0.001348	0.000338
5000	0.009581	0.002513
20000	0.032069	0.026340
50000	0.201824	0.026500

- **Observation:** At an input size of 50,000, the randomized algorithm is nearly **7.6 times faster** than the deterministic one.
- **Relation to Theory:** This result is expected. While both are O(n), the *constant factor* hidden in the Big-O notation is much larger for Median of Medians. The process of dividing into chunks, finding medians of each, and recursively calling the function just to find a pivot introduces significant overhead. Randomized Quickselect's pivot selection is a single, fast operation.

Performance on Different Input Distributions

- **Sorted or Reverse-Sorted Data:**
  - **Randomized Quickselect:** Because our implementation uses random.choice() to select the pivot, its performance is resilient to the initial order of the data. It would not degrade to its O(n2) worst case, as a naive implementation (e.g., always picking the first or last element) would. Its runtime would remain consistent with the random data case.
  - **Median of Medians:** The performance of this algorithm is largely independent of the input distribution. Its mechanical process of finding a pivot ensures a good partition regardless of whether the data is sorted, reverse-sorted, or random.
- **Data with Many Duplicates:**
  - Both implementations handle duplicates gracefully. By explicitly creating a pivots list for all elements equal to the pivot (pivots = [x for x in arr if x == pivot]), the algorithms correctly place the k-th element. If k falls within the

range of indices occupied by the pivot elements, the pivot is returned immediately. This prevents skewed partitions or infinite recursion that could occur if duplicates were handled improperly. The performance would not be negatively impacted.

In summary, the empirical results strongly support the theoretical analysis. **Randomized Quickselect** is the superior choice for general use due to its speed and simplicity, while **Median of Medians** serves as a crucial algorithm for applications where a worst-case performance guarantee is non-negotiable.

Graph [Deterministic vs Randomised]