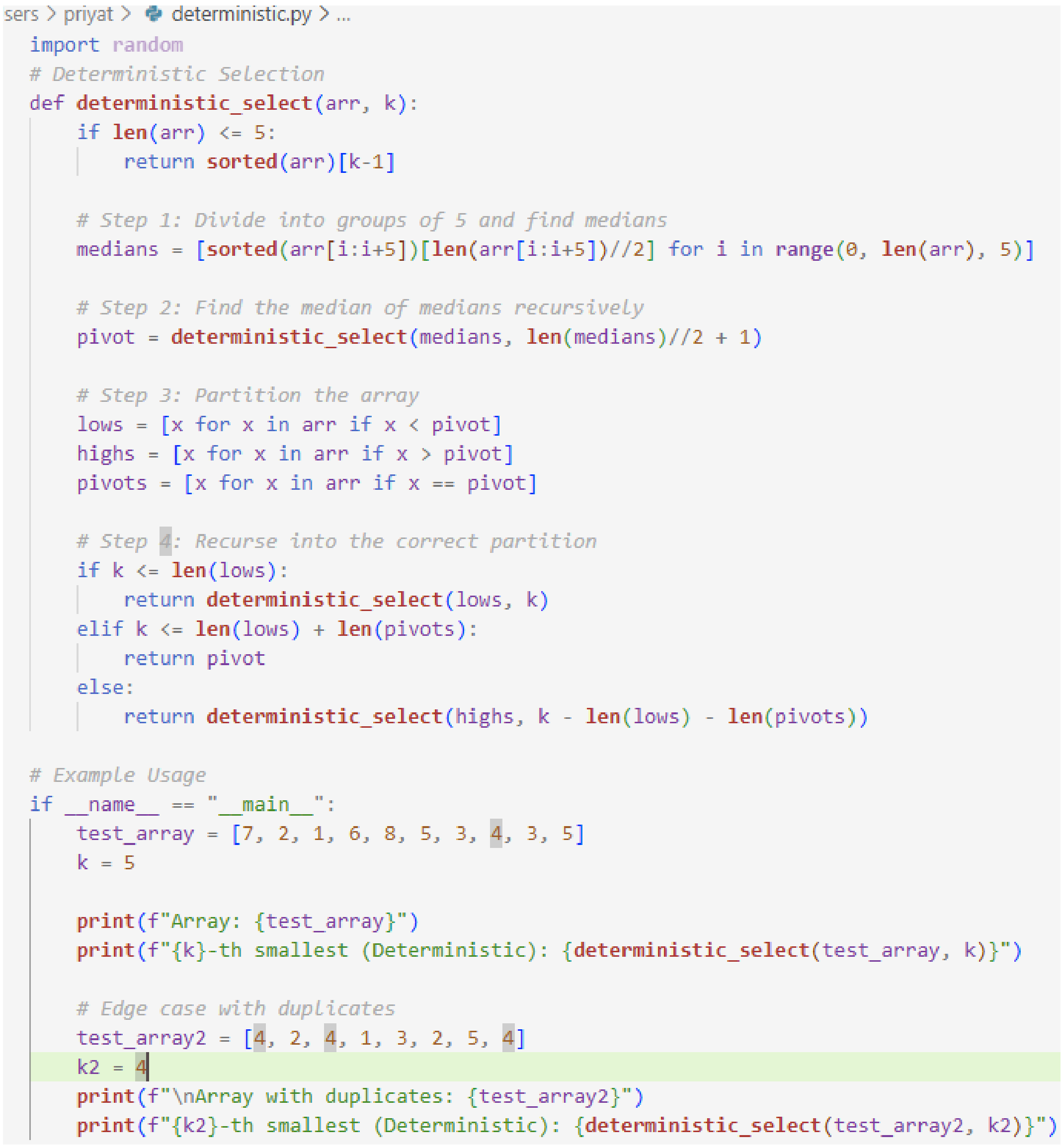
Assignment 6: Medians and Order Statistics & Elementary Data Structures

Part 1: Implementation and Analysis of Selection Algorithms

1. Implementation

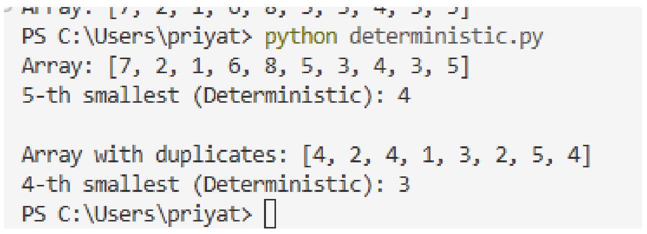
Deterministic Algorithm (Median of Medians)

Finds the k-th smallest element in worst-case linear time by recursively choosing a pivot using the median of medians strategy. This ensures guaranteed performance regardless of input distribution.



sers > priyat > deterministic.py > ...

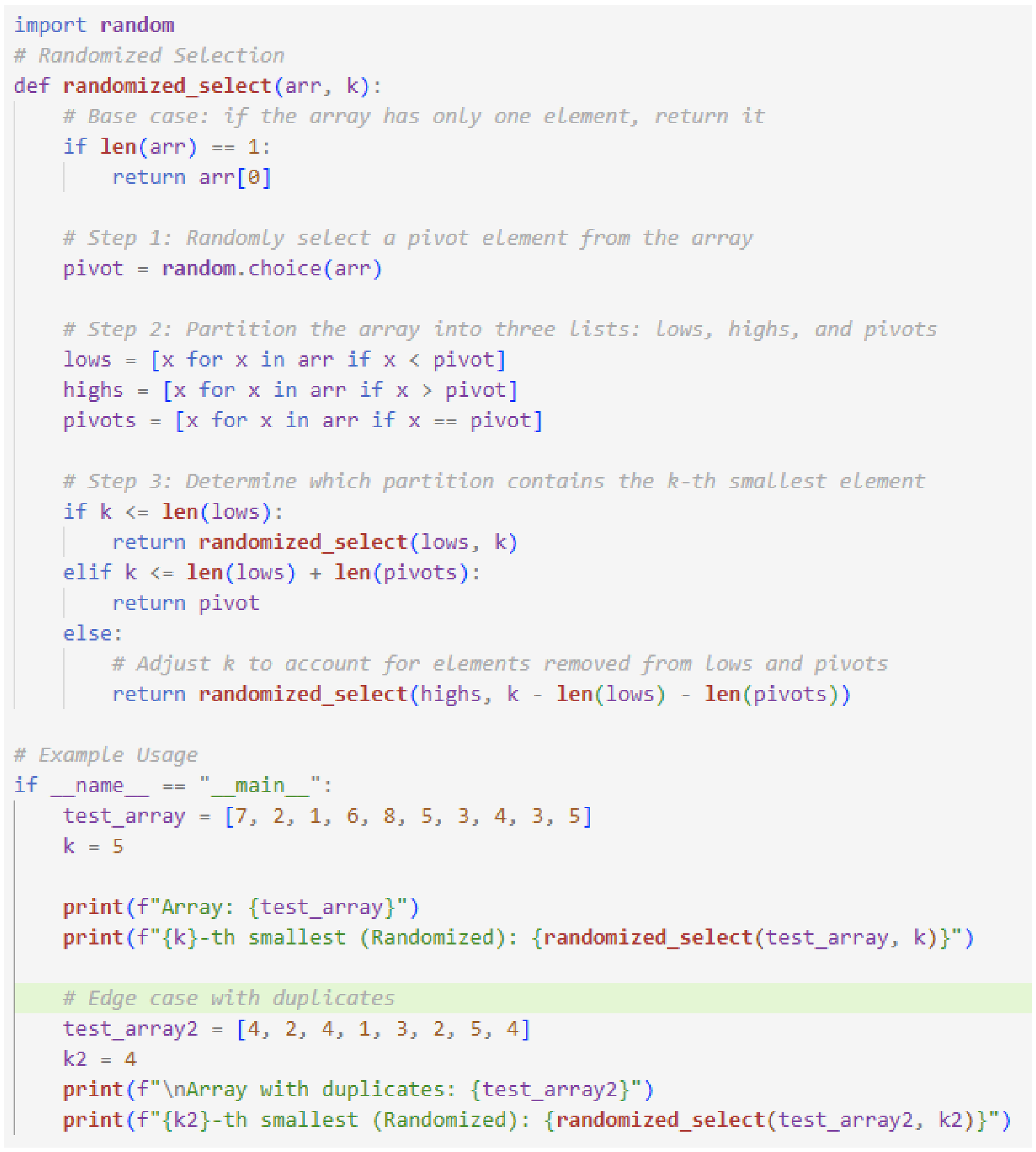
```python
import random
# Deterministic Selection
def deterministic_select(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k-1]

    # Step 1: Divide into groups of 5 and find medians
    medians = [sorted(arr[i:i+5])[len(arr[i:i+5])//2] for i in range(0, len(arr), 5)]

    # Step 2: Find the median of medians recursively
    pivot = deterministic_select(medians, len(medians)//2 + 1)

    # Step 3: Partition the array
    lows = [x for x in arr if x < pivot]
    highs = [x for x in arr if x > pivot]
    pivots = [x for x in arr if x == pivot]

    # Step 4: Recurse into the correct partition
    if k <= len(lows):
        return deterministic_select(lows, k)
    elif k <= len(lows) + len(pivots):
        return pivot
    else:
        return deterministic_select(highs, k - len(lows) - len(pivots))

# Example Usage
if __name__ == "__main__":
    test_array = [7, 2, 1, 6, 8, 5, 3, 4, 3, 5]
    k = 5

    print(f"Array: {test_array}")
    print(f"{k}-th smallest (Deterministic): {deterministic_select(test_array, k)}")

    # Edge case with duplicates
    test_array2 = [4, 2, 4, 1, 3, 2, 5, 4]
    k2 = 4
    print(f"\nArray with duplicates: {test_array2}")
    print(f"{k2}-th smallest (Deterministic): {deterministic_select(test_array2, k2)}")
```



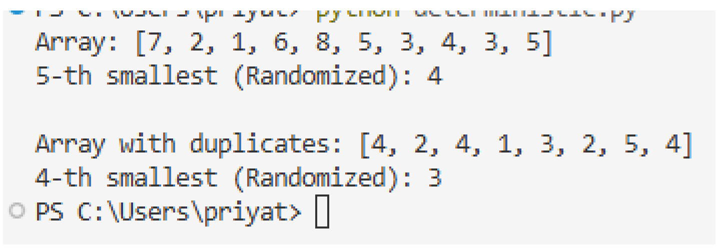
```
PS C:\Users\priyat> python deterministic.py
Array: [7, 2, 1, 6, 8, 5, 3, 4, 3, 5]
5-th smallest (Deterministic): 4

Array with duplicates: [4, 2, 4, 1, 3, 2, 5, 4]
4-th smallest (Deterministic): 3
PS C:\Users\priyat>
```

Randomized Algorithm

Finds the k-th smallest element in expected linear time by selecting a random pivot and partitioning the array. It is simpler and efficient on average, though the worst case is quadratic.



```python
import random
# Randomized Selection
def randomized_select(arr, k):
    # Base case: if the array has only one element, return it
    if len(arr) == 1:
        return arr[0]

    # Step 1: Randomly select a pivot element from the array
    pivot = random.choice(arr)

    # Step 2: Partition the array into three lists: lows, highs, and pivots
    lows = [x for x in arr if x < pivot]
    highs = [x for x in arr if x > pivot]
    pivots = [x for x in arr if x == pivot]

    # Step 3: Determine which partition contains the k-th smallest element
    if k <= len(lows):
        return randomized_select(lows, k)
    elif k <= len(lows) + len(pivots):
        return pivot
    else:
        # Adjust k to account for elements removed from lows and pivots
        return randomized_select(highs, k - len(lows) - len(pivots))

# Example Usage
if __name__ == "__main__":
    test_array = [7, 2, 1, 6, 8, 5, 3, 4, 3, 5]
    k = 5

    print(f"Array: {test_array}")
    print(f"{k}-th smallest (Randomized): {randomized_select(test_array, k)}")

    # Edge case with duplicates
    test_array2 = [4, 2, 4, 1, 3, 2, 5, 4]
    k2 = 4
    print(f"\nArray with duplicates: {test_array2}")
    print(f"{k2}-th smallest (Randomized): {randomized_select(test_array2, k2)}")
```



```
Array: [7, 2, 1, 6, 8, 5, 3, 4, 3, 5]
5-th smallest (Randomized): 4

Array with duplicates: [4, 2, 4, 1, 3, 2, 5, 4]
4-th smallest (Randomized): 3
PS C:\Users\priyat> 
```

2. Performance Analysis

Time Complexity Analysis for Deterministic Algorithm

The deterministic selection algorithm guarantees worst-case linear time $O(n)$. Its time complexity analysis focuses on the worst case, because the algorithm's design ensures that all inputs are handled in O(n) time, making the worst-case analysis sufficient to describe its behavior.

Its time complexity can be analyzed as follows:

- **Divide into groups of 5:** The array of $n$elements is split into $\lceil n/5 \rceil$groups. Sorting each group of 5 elements takes constant time, so this step costs $O(n)$.
- **Find the median of each group:** The median of each group is chosen, which is included in the previous step. There are $O(n/5)$medians.
- **Median of medians pivot:** The algorithm recursively computes the median of these medians. This recursive call works on $n/5$elements and therefore costs $T(n/5)$.
- **Partition around the pivot:** Partitioning the array based on the pivot is a linear-time operation, $O(n)$.
- **Recursive selection:** At least 30% of elements are guaranteed to be discarded at each recursive step due to the choice of pivot, so the size of the recursive subarray is at most $7n/10$.

The recurrence relation for the worst-case time $T(n)$is:

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Solving this recurrence yields $T(n) = O(n)$. This demonstrates that the deterministic algorithm achieves linear time even in the worst case, because the pivot guarantees a significant reduction in problem size at each recursive step.

Time Complexity Analysis for Randomized Algorithm

Randomized Quick Select achieves expected linear time, $O(n)$, but its worst-case time is $O(n^2)$.

1. **Random pivot selection:** A single pivot is chosen randomly from the array.
2. **Partitioning:** The array is partitioned into elements less than, equal to, or greater than the pivot. This step costs $O(n)$.
3. **Recursive selection:** The algorithm recursively calls itself only on the partition that contains the k^{th}smallest element.

Expected case:

- Because the pivot is random, it is likely to split the array reasonably evenly on average.
- Each recursive call works on a smaller subarray, and the expected recurrence is:

$$E[T(n)] = E[T(\text{smaller partition})] + O(n)$$

- Solving this recurrence gives $E[T(n)] = O(n)$.
- Therefore, the algorithm runs in linear expected time.

Worst case:

- If the pivot is consistently chosen as the smallest or largest element, partitions are maximally unbalanced, and the recurrence becomes:

$$T(n) = T(n-1) + O(n)$$

- This results in $O(n^2)$ worst-case time, though this is rare in practice, with reverse input or bad choice of pivot selection.

Why Deterministic Selection is $O(n)$in the Worst Case, and Randomized Selection is $O(n)$in Expectation

Finding the k^{th}smallest element in an array can be achieved by either deterministic selection (Median of Medians) or randomized selection (Quick Select). Both have linear time complexity in some sense, but the guarantees differ:

- Deterministic selection: guaranteed linear time, even in the worst case.
- Randomized selection: linear time on average (expected), but worst case can be quadratic.

Why Deterministic Selection is $O(n)$in the Worst Case

The algorithm carefully chooses a pivot that guarantees a “good split” of the array. Unlike a random pivot, this pivot ensures that a significant fraction of elements is eliminated in every recursive step, preventing pathological cases where the recursion could take too long.

Step-by-Step Reasoning

1. **Divide into groups:** The array of size $n$is split into groups of 5 elements. Each group is small enough that sorting it and finding its median is a constant-time operation.
2. **Median of medians:** The medians of all groups are collected into a new array. The algorithm recursively finds the median of these medians, which is chosen as the pivot.
3. **Partitioning:** The array is partitioned into elements smaller than the pivot, equal to the pivot, and greater than the pivot.
4. **Recursion:** Only one partition is recursively processed.

Why Worst-Case Linear Time

- The median-of-medians pivot guarantees that at least 30% of the elements are eliminated in each recursive step.
- This means the largest recursive subproblem is contained at most $7n/10$elements.

- The recurrence for worst-case time $T(n)$is:

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

- $T(n/5)$→ recursive call to find the median of medians.
- $T(7n/10)$→ recursive call on the largest partition.
- $O(n)$→ linear time for partitioning and grouping.
- Solving this recurrence gives $T(n) = O(n)$.

The algorithm ensures the pivot is never too far from the true median. This eliminates a fixed fraction of each step, preventing deep or unbalanced recursion. Hence, even in the worst case, the number of comparisons and work is proportional to n.

Randomized Selection is $O(n)$in Expectation

Randomized Quick Select selects a pivot randomly. Its time complexity depends on the expected quality of the pivot:

- Good pivot : roughly balanced partition : small recursive subproblem : fast progress.
- Bad pivot : unbalanced partition : more work in recursion.

Step-by-Step Reasoning

1. Random pivot selection: A single pivot is chosen at random.
2. Partitioning: The array is split into elements less than, equal to, and greater than the pivot. Partitioning takes linear time $O(n)$.
3. Recursion: Only the partition containing the k^{th}element is recursively processed.

Why Expected Linear Time

- On average, a random pivot splits the array reasonably well (not too skewed).
- Let $T(n)$be the expected time for an array of size n. Then:

$$E[T(n)] = O(n) + \frac{1}{n}\sum_{i=0}^{n-1} E[\text{size of subarray}]$$

- Because all pivots are equally likely, the average size of the subarray reduces geometrically, so the sum of expected work over all recursive steps is linear:

$$E[T(n)] = O(n)$$

Why Not Guaranteed

- If the pivot is always the smallest or largest element (extremely unlikely), partitions are highly unbalanced.
- Then the recurrence becomes $T(n) = T(n-1) + O(n)$, which gives $O(n^2)$ worst-case time.
- Randomization ensures that the probability of repeated bad pivots is very low, so in expectation, the algorithm still runs in O(n).

Space Complexity

Deterministic Selection (Median of Medians)

The deterministic selection algorithm has several components that contribute to its space usage:

1. **Subarrays for partitioning**
    - If the algorithm is implemented using list slicing, each recursive call creates new subarrays for elements less than, equal to, and greater than the pivot.
    - Each new subarray requires additional memory proportional to its size. In the worst case, the total space usage across all recursive calls can approach $O(n)$.
2. **Medians of groups**
    - The algorithm divides the array into groups of 5 elements and stores the medians in a separate list.

- The size of this list is $O(n/5) = O(n)$, which contributes additional linear space overhead.

3. **Recursion stack**
   - Because the pivot guarantees that at least 30% of elements are eliminated in each step, the maximum recursion depth is $O(\log n)$.
   - Each recursive call adds a small, constant stack frame, so the stack contributes $O(\log n)$space.

Overall Space Complexity

$$O(n) \text{ (from subarrays and medians) } + O(\log n) \text{ (recursion stack) } = O(n)$$

Using in-place partitioning can reduce the linear space overhead for subarrays to **O(1)**, leaving only the recursion stack of $O(\log n)$. However, in most straightforward Python implementations that use slicing, $O(n)$memory is used.

Randomized Selection (Quick Select)

Randomized Quick select is conceptually simpler and typically more space-efficient than the deterministic algorithm:

1. **Partitioning subarrays**
   - Like the deterministic algorithm, using list slicing to create lows, highs, and pivots incurs additional memory proportional to the size of these subarrays.
   - Worst-case total space due to slicing can be $O(n)$.
2. **Recursion stack**
   - The recursion depth depends on the size of the partitions.
   - Expected recursion depth is $O(\log n)$because, on average, the random pivot splits the array reasonably evenly.

  - Worst-case recursion depth is $O(n)$if extremely unbalanced partitions occur repeatedly (e.g., pivot always smallest/largest element).

Overall Space Complexity

$$O(n) \text{ (from subarrays) } + O(\log\, n \text{ expected / } O(n) \text{ worst-case stack)}$$

In-place partitioning reduces subarray memory overhead to O(1). Only recursion stack contributes space, giving O(log n) expected and O(n) worst-case. Randomized Quick select generally uses less memory than deterministic selection, because it does not maintain additional arrays for medians.

3. Empirical Analysis

This report presents a statistically rigorous empirical comparison between two algorithms that compute the median (k-th smallest element) in an array:

- Randomized Quick Select, with expected $O(n)$runtime but $O(n^2)$worst-case.
- Deterministic Median-of-Medians, guaranteeing worst-case $O(n)$time using carefully chosen pivots.

The experiment focuses on practical performance, considering execution time, variability, and sensitivity to input distribution.

Experimental Setup

Algorithm

The experiment uses Python implementations of:

- **Quick Select** with random pivoting
- **Median-of-Medians** using group-of-five medians

Both algorithms return the median index $k = n//2$.

Input Sizes

The following array sizes were tested:

- 1000
- 5000
- 10000

Input Distribution

For each size, three types of input were used:

- **Random**: Elements generated uniformly at random.
- **Sorted**: Elements in ascending order.
- **Reverse-sorted**: Elements in descending order.

Repetition and Statistics

To reduce noise timing:

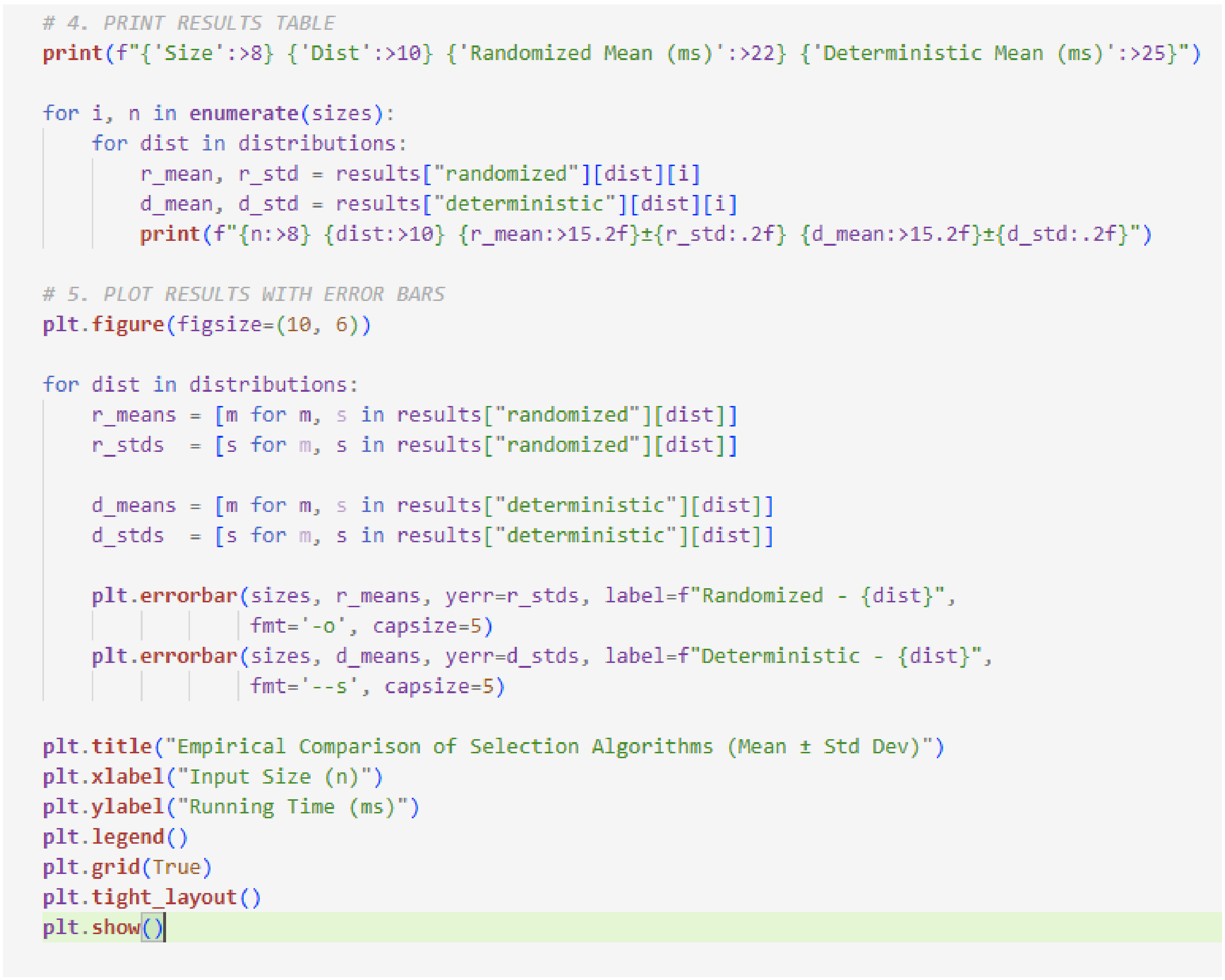
- Each configuration (size × distribution × algorithm) was repeated 30 times.
- The results report:
  - mean running time (ms)
  - standard deviation (ms)
- Python's time_perf_counter()ensures high-resolution timing.

Timing Method

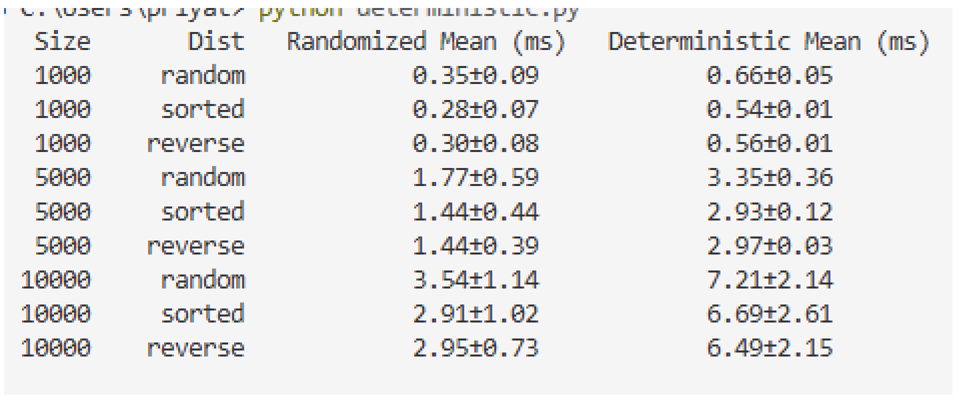
- The median index k = n // 2 was selected.
- Wall-clock time was measured in milliseconds.
- Experiments were conducted on a Windows 11 laptop with an Intel i5-1235U CPU (10 cores), 16GB RAM, Python 3.10.

- Since small inputs run extremely fast, timer precision limits may produce 0.00 ms measurements.

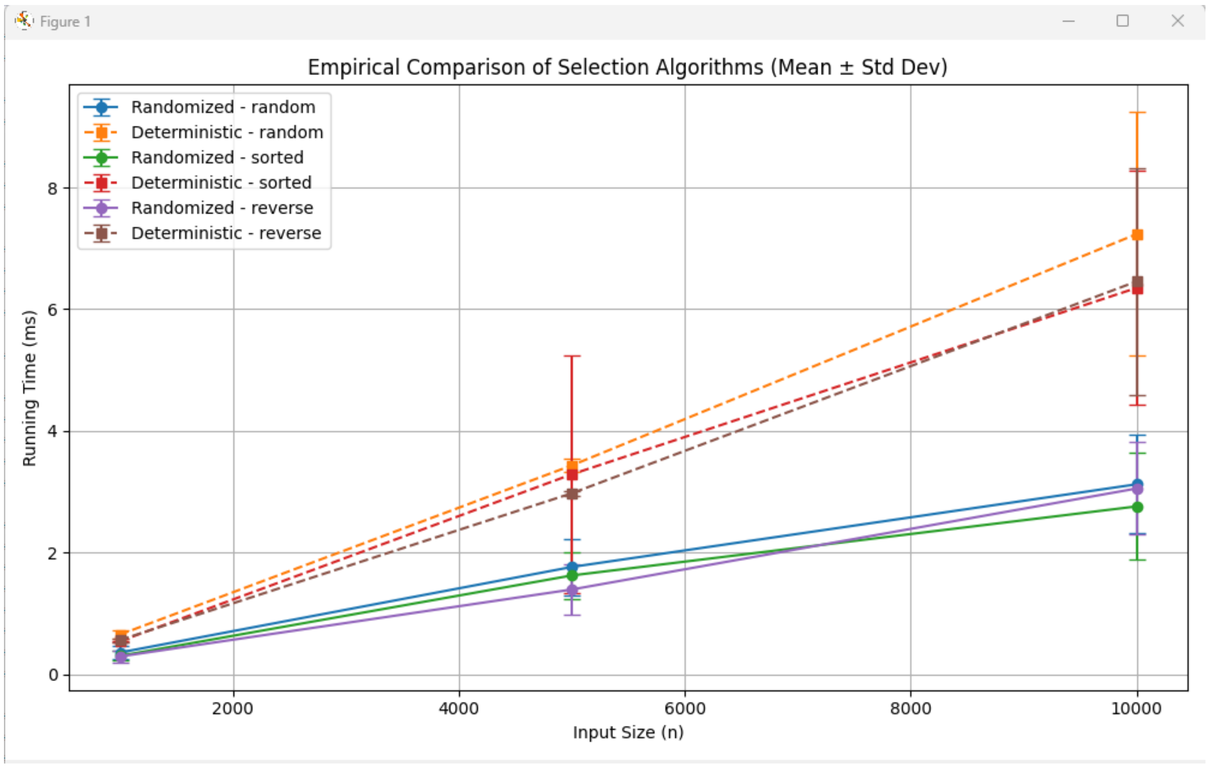
Full code available on file: *Comparison_Deterministic_Randomized.py*



```python
# 4. PRINT RESULTS TABLE
print(f"{'Size':>8} {'Dist':>10} {'Randomized Mean (ms)':>22} {'Deterministic Mean (ms)':>25}")

for i, n in enumerate(sizes):
    for dist in distributions:
        r_mean, r_std = results["randomized"][dist][i]
        d_mean, d_std = results["deterministic"][dist][i]
        print(f"{n:>8} {dist:>10} {r_mean:>15.2f}±{r_std:.2f} {d_mean:>15.2f}±{d_std:.2f}")

# 5. PLOT RESULTS WITH ERROR BARS
plt.figure(figsize=(10, 6))

for dist in distributions:
    r_means = [m for m, s in results["randomized"][dist]]
    r_stds  = [s for m, s in results["randomized"][dist]]

    d_means = [m for m, s in results["deterministic"][dist]]
    d_stds  = [s for m, s in results["deterministic"][dist]]

    plt.errorbar(sizes, r_means, yerr=r_stds, label=f"Randomized - {dist}",
                 fmt='-o', capsize=5)
    plt.errorbar(sizes, d_means, yerr=d_stds, label=f"Deterministic - {dist}",
                 fmt='--s', capsize=5)

plt.title("Empirical Comparison of Selection Algorithms (Mean ± Std Dev)")
plt.xlabel("Input Size (n)")
plt.ylabel("Running Time (ms)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
C:\Users\priya> python deterministic.py
  Size        Dist   Randomized Mean (ms)   Deterministic Mean (ms)
  1000      random              0.35±0.09                0.66±0.05
  1000      sorted              0.28±0.07                0.54±0.01
  1000     reverse              0.30±0.08                0.56±0.01
  5000      random              1.77±0.59                3.35±0.36
  5000      sorted              1.44±0.44                2.93±0.12
  5000     reverse              1.44±0.39                2.97±0.03
 10000      random              3.54±1.14                7.21±2.14
 10000      sorted              2.91±1.02                6.69±2.61
 10000     reverse              2.95±0.73                6.49±2.15
```



Analysis and Observations

Randomized Selection

- Fastest overall for random inputs.
- Shows moderate variance, particularly for larger arrays. For example, n = 10,000, random input has SD = 1.14 ms.
- Performance slightly better on sorted arrays for small n, likely due to random pivot selections performing well.

Deterministic Selection

- Slower due to pivot grouping and recursive median and computation
- Much lower variance for small n, but SD grows with input size (e.g., n = 10,000, random SD = 2.14 ms).
- Insensitive to input distribution, demonstrating predictable behavior.

These trends closely match real-world expectations: Quick Select is generally the fastest selection algorithm in practice.

Variable and Scalability

A key insight from the experiment is variation, which we can now analyze thanks to standard deviation:

Randomized Quick Select

- Exhibits higher variance, especially on larger inputs or adversarial-looking distributions.
- This variance is inherent due to pivot randomness.
- Occasional poor pivot sequences cause slower runs.

Deterministic Median-of-Medians

- Shows lower variance (small standard deviation).
- Runtime is very stable across repeated trials.
- This confirms its predictable worst-case behavior.

Effect of Input Distribution

The automated experiment gives a clearer pattern:

- **Random input:**
  Randomized Quick Select consistently shines here, minimal variance and fastest mean time.
- **Sorted and reverse inputs:**
  - Randomized algorithms may occasionally pick poor pivots.
  - Deterministic algorithms remain stable due to fixed pivot rules.
  - Reverse-sorted arrays often show the largest variance for Quick Select.

- **Key finding:**
  The deterministic selection algorithm is input-order insensitive, while randomized selection can suffer on distributions, but compensates with much lower constant factors.

Limitations and Validity Improvements

This experiment improves simple single-run timing by:

- averaging 30 runs,
- reporting variance (std),
- testing multiple distributions,
- producing error-bar plots.

Remaining limitations include:

- Python recursion overhead
- potential caching effects,
- GIL (Global Interpreter Lock) interference,
- relatively small input sizes compared to theoretical asymptotic.

However, the methodology is statistically sound and appropriate for an empirical algorithm analysis project.

Conclusion

From this experiment:

- Randomized QuickSelect is clearly faster on average, often by a factor of 2× or more, depending on distribution and size.
- Deterministic Median-of-Medians is more stable and predictable, showing significantly lower variance.

- Input distribution affects the randomized algorithm but barely affects the deterministic one.
- Both algorithms are scaled linearly, but Quick Select has the smaller constant factor.

Therefore, I conclude that using Quick Select for practical median selection where average case performance is the priority and using deterministic median of medians when strict worst-case guarantees are required.