Report

Summary

1. **Deterministic Quicksort**: Uses a fixed pivot strategy.
2. **Randomized Quicksort**: Uses a random pivot to reduce the chance of worst-case scenarios.
3. **Performance Analysis**: Measures and compares the running times of both implementations.

Step 1: Deterministic Quicksort Implementation

Step 1.1: Basic Structure

Start by defining the Quicksort function. It will take an array as input and sort it in place.

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # 1/2 = Middle = pivot
    left = [x for x in arr if x < pivot]  # Elements < pivot
    middle = [x for x in arr if x == pivot]  # Elements = pivot
    right = [x for x in arr if x > pivot]  # Elements > pivot
    return quicksort(left) + middle + quicksort(right)
```

Functions	Description
quicksort(arr)	Sorts an array using the deterministic Quicksort algorithm.

Step 1.2: Testing the Deterministic Quicksort

```python
# Test the deterministic quicksort
if __name__ == "__main__":
    sample_array = [3, 6, 8, 9, 1, 2, 1]
    sorted_array = quicksort(sample_array)
    print("Sorted array:", sorted_array)
```

```
python3 quicksort.py
Sorted array (deterministic): [1, 1, 2, 3, 6, 8, 9]
```

Step 2: Performance Analysis

You can analyze the performance using the `time` library to measure the execution time.

```python
import time

# Performance analysis function
def performance_analysis(arr):
    start_time = time.time()
    quicksort(arr)
    end_time = time.time()
    return end_time - start_time

# Test the performance
if __name__ == "__main__":
    import random
    random_array = [random.randint(1, 1000) for _ in range(1000)]
    time_taken = performance_analysis(random_array)
    print(f"Time taken to sort: {time_taken:.6f} seconds")
```

Functions	Description
performance_analysis(sort_function, arr)	Analyzes the performance of a sorting function.

Step 3: Randomized Quicksort Implementation

Step 3.1: Implementing Randomized Quicksort

Modify the `quicksort` function to randomly select a pivot.

```python
import random

def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot_index = random.randint(0, len(arr) - 1)  # Random = pivot
    pivot = arr[pivot_index] # Set = pivot
    left = [x for x in arr if x < pivot]  # Elements < pivot
    middle = [x for x in arr if x == pivot]  # Elements = pivot
    right = [x for x in arr if x > pivot]  # Elements > pivot
    return randomized_quicksort(left) + middle + randomized_quicksort(right)
```

Functions	Description
randomized_quicksort(arr)	Sorts an array using the randomized Quicksort algorithm.

```
python3 quicksort.py
Sorted array (deterministic): [1, 1, 2, 3, 6, 8, 10]
Sorted array (randomized): [1, 1, 2, 3, 6, 8, 10]
```

```
python3 quicksort.py
Sorted array (deterministic): [1, 1, 2, 3, 6, 8, 10]
Sorted array (randomized): [1, 1, 2, 3, 6, 8, 10]
Deterministic Quicksort time: 0.002722 seconds
Randomized Quicksort time: 0.004137 seconds
```

Step 3.2: Testing the Randomized Quicksort

```python
if __name__ == "__main__":
    sample_array = [3, 6, 8, 10, 1, 2, 1]
    sorted_array = randomized_quicksort(sample_array)
    print("Sorted array (randomized):", sorted_array)
```

Step 4: Empirical Analysis

You can compare the performance of both implementations.

```python
def compare_performance(arr):
    deterministic_time = performance_analysis(arr.copy())
    randomized_time = performance_analysis(arr.copy())
    print(f"Deterministic Quicksort time: {deterministic_time:.6f} seconds")
    print(f"Randomized Quicksort time: {randomized_time:.6f} seconds")

if __name__ == "__main__":
    random_array = [random.randint(1, 1000) for _ in range(1000)]
    compare_performance(random_array)
```

4.1 Performance Analysis

- **Time Complexity**:
  - Best Case: (O(n \log n)) when the pivot divides the array evenly.
  - Average Case: (O(n \log n)) due to the logarithmic number of partitions.
  - Worst Case: (O(n^2)) when the pivot is consistently the smallest or largest element.
- **Space Complexity**:
  - Average space complexity is (O(\log n)) due to recursion stack in the best case, but can reach (O(n)) in the worst case due to stack depth.

5. Randomized Quicksort

Mitigating Worst-Case Performance: Randomized Quicksort enhances the traditional deterministic approach by selecting the pivot randomly instead of using a fixed strategy (like always picking the middle element). This random selection significantly reduces the likelihood of encountering the worst-case scenario, which occurs when the pivot consistently divides the array in a highly unbalanced manner (e.g., always picking the smallest or largest element).

By introducing randomness, the chances of consistently poor partitions diminish. For example, if we were to sort an already sorted array, a deterministic approach might always select the middle element, leading to one side of the partition being empty, which results in (O(n^2)) time complexity. Randomization means that, on average, we will obtain a better distribution of elements across partitions, leading to a more balanced sort.

Empirical Evidence of Performance Improvements: Empirical studies and tests have shown that Randomized Quicksort often performs better in practice, especially on diverse datasets. When tested against different configurations (random, sorted, reverse-sorted), the performance improvements manifest in reduced execution times and fewer comparisons in average cases. Studies have demonstrated that randomized pivots consistently yield (O(n \log n)) performance in the average case, while the worst-case scenario becomes rare, demonstrating resilience against pathological inputs.

6. Empirical Analysis

Comparative Running Times: To analyze the performance of both implementations, we can run tests using both randomized and sorted datasets:

1. **Random Datasets:**

    - Generate large random arrays (e.g., 1,000 elements) and measure the time taken by both deterministic and randomized Quicksort.
    - Expect both implementations to show (O(n \log n)) behavior, with Randomized Quicksort slightly outperforming in most cases due to better pivot selection.

2. **Sorted Datasets:**

    - Test the algorithms with already sorted arrays to observe how each implementation reacts to this worst-case input scenario for deterministic Quicksort.
    - The deterministic version might show (O(n^2)) performance, while Randomized Quicksort should maintain (O(n \log n)) behavior because the random pivot selection avoids poor partitioning.

Analysis of Trends and Variances: From the tests, we can expect to see:

- **Randomized Quicksort** consistently performing well across varied datasets, maintaining efficiency even with sorted inputs due to its random pivot selection.
- **Deterministic Quicksort** may exhibit significant slowdowns with sorted or nearly sorted datasets, highlighting its sensitivity to input order.

Overall, the variance in performance can be attributed to how well each algorithm manages to balance partitions, with the randomized version exhibiting lower variance due to its pivot selection strategy.

7. Conclusion

In summary, both Quicksort implementations—deterministic and randomized—are effective sorting algorithms, but they exhibit different strengths and weaknesses:

Time Complexity Analysis

- **Deterministic Quicksort**:

    - **Best Case**: (O(n \log n)) - when the pivot splits the array into equal halves.
    - **Average Case**: (O(n \log n)) - average time taken over many inputs, still logarithmic partitions.
    - **Worst Case**: (O(n^2)) - when the smallest or largest elements are consistently chosen as pivots, resulting in unbalanced partitions.

- **Randomized Quicksort**:

    - **Best and Average Case**: Still (O(n \log n)) due to improved pivot selection.
    - **Worst Case**: Significantly reduced likelihood due to randomness, but theoretically could still be (O(n^2)) on rare occasions.

- **Efficiency**: Randomized Quicksort generally provides better performance across diverse datasets, maintaining (O(n \log n)) time complexity in most cases, while deterministic Quicksort can degrade to (O(n^2)) under specific conditions.

- **Practical Applications**:

- **Deterministic Quicksort** can be useful when data characteristics are well understood and when sorting small datasets where the overhead of randomization may not justify its benefits.
- **Randomized Quicksort** is preferred in scenarios involving large datasets or unknown distributions, where avoiding worst-case performance is critical. It is also beneficial in applications like databases or data processing frameworks, where efficiency and reliability are paramount.

Overall, the choice between these implementations should consider the input characteristics and the importance of performance consistency in the specific application context.

Submission Instructions

1. **GitHub Repository**:
   - https://github.com/baralsamrat/MSCS532_Assignment5
   - Upload `quicksort.py` and the report as `report.pdf`.
   - Include a `README.md` with instructions and a summary of your findings.
2. **Submit the Link**:
   - MSCS532_Assignment5
