Quicksort Algorithm: Implementation, Analysis, and Randomization

Introduction

The Quicksort algorithm is a widely used sorting algorithm that employs the divide-and-conquer strategy to organize data efficiently. Known for its efficiency and versatility, Quicksort is often favored in applications where large datasets require fast and in-place sorting. By selecting a pivot element and partitioning data around it, Quicksort recursively sorts smaller subarrays, achieving an average time complexity of O(nlogn).

This report examines both deterministic and randomized versions of the Quicksort algorithm. Through theoretical analysis and empirical testing, we aim to understand the performance characteristics of Quicksort and explore how randomization can mitigate the worst-case time complexity. This study provides insights valuable for optimizing algorithms in data processing, search systems, and resource-constrained environments, ultimately contributing to the development of scalable and efficient software solutions.

Deterministic Quicksort Implementation

The deterministic version of Quicksort selects a pivot in a predefined manner. In this implementation, we select the last element of the array as the pivot. This approach provides a consistent framework for partitioning and sorting, but it may lead to poor performance if the array is already sorted or nearly sorted.

1. **Selecting a Pivot**: The pivot selection is straightforward—choosing the last element of the array.
2. **Partitioning the Array**: The partition function arranges elements such that those less than the pivot come before it, while those greater than the pivot are positioned after it.
3. **Recursive Sorting**: Once partitioning is complete, the algorithm recursively sorts the subarrays on each side of the pivot.

Here’s the Python code for the deterministic Quicksort:

```python
def quicksort(arr, low, high):
  if low < high:
    # Partition the array and get the partition index
    pi = partition(arr, low, high)
    # Recursively sort elements before and after partition
    quicksort(arr, low, pi - 1)
    quicksort(arr, pi + 1, high)
```

```
def partition(arr, low, high):
  # Select the pivot (last element in this case)
  pivot = arr[high]
  i = low - 1  # Index of the smaller element
  for j in range(low, high):
     if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]
  arr[i + 1], arr[high] = arr[high], arr[i + 1]
  return i + 1
```

Explanation of the Code:

- **quicksort**: This function recursively sorts the array by calling the partition function to find the correct position of the pivot element.
- **partition**: This function rearranges the elements in the array to ensure that all elements less than the pivot are on the left and those greater are on the right.

This deterministic approach is efficient for most cases, but it can encounter the worst-case scenario when the array is already sorted, resulting in O(n^2) complexity.

```
from DeterminsticQuickSort import quicksort



import time

import random

import sys



# Increase recursion limit

sys.setrecursionlimit(20000)  # Adjust this value as needed based on array size



# Define a function to measure execution time of deterministic quicksort

def measure_sort_time(arr):
    start_time = time.time()

    quicksort(arr, 0, len(arr) - 1)

    end_time = time.time()

    return end_time - start_time


# Deterministic Quicksort functions (same as before)

def quicksort(arr, low, high):

    if low < high:

        pi = partition(arr, low, high)

        quicksort(arr, low, pi - 1)

        quicksort(arr, pi + 1, high)


def partition(arr, low, high):

    pivot = arr[high]

    i = low - 1

    for j in range(low, high):

        if arr[j] < pivot:

            i += 1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1


# Test arrays with different cases

array_sizes = [1000, 5000, 10000]  # Testing with different input sizes
```

```python
# 1. Empty array test
empty_array = []
time_empty = measure_sort_time(empty_array)
print(f"Empty array sorted in: {time_empty:.6f} seconds")


# Loop through other test arrays with increasing sizes
for size in array_sizes:
    # 2. Randomly generated array
    random_array = [random.randint(1, 10000) for _ in range(size)]
    time_random = measure_sort_time(random_array)
    print(f"Random array of size {size} sorted in: {time_random:.6f} seconds")


    # 3. Already sorted array
    sorted_array = list(range(1, size + 1))
    time_sorted = measure_sort_time(sorted_array)
    print(f"Sorted array of size {size} sorted in: {time_sorted:.6f} seconds")


    # 4. Reverse-sorted array
    reverse_sorted_array = list(range(size, 0, -1))
    time_reverse_sorted = measure_sort_time(reverse_sorted_array)
    print(f"Reverse sorted array of size {size} sorted in: {time_reverse_sorted:.6f} seconds")


    # 5. Array with repeated elements
```

```python
repeated_elements_array = [random.choice([1, 2, 3, 4, 5]) for _ in range(size)]

time_repeated = measure_sort_time(repeated_elements_array)

print(f"Array with repeated elements of size {size} sorted in: {time_repeated:.6f} seconds")



print("-" * 40)
```

```
python3 TestDeterministicQuickSort.py

Empty array sorted in: 0.000001 seconds

Random array of size 1000 sorted in: 0.001142 seconds

Sorted array of size 1000 sorted in: 0.037648 seconds

Reverse sorted array of size 1000 sorted in: 0.022093 seconds

Array with repeated elements of size 1000 sorted in: 0.002855 seconds

----------------------------------------

Random array of size 5000 sorted in: 0.004198 seconds

Sorted array of size 5000 sorted in: 0.730644 seconds

Reverse sorted array of size 5000 sorted in: 0.813689 seconds

Array with repeated elements of size 5000 sorted in: 0.071713 seconds

----------------------------------------

Random array of size 10000 sorted in: 0.008779 seconds

Sorted array of size 10000 sorted in: 2.875890 seconds

Reverse sorted array of size 10000 sorted in: 2.096908 seconds

Array with repeated elements of size 10000 sorted in: 0.248777 seconds
```

Empirical Analysis

To evaluate the performance of the deterministic Quicksort implementation, we measured execution times across different array types and sizes. The results are as follows:

1. **Empty Array**: The algorithm handled an empty array with negligible time due to the lack of elements to sort.
2. **Random Array**: For randomly generated arrays, the sorting time scaled efficiently with the array size. For instance, a 1,000-element array sorted in approximately 0.001 seconds, while a 10,000-element array completed in 0.009 seconds. This matches the expected average-case time complexity of O(nlogn).
3. **Sorted Array**: The algorithm showed significant increases in sorting time for already sorted arrays, taking 0.038 seconds for 1,000 elements and 2.88 seconds for 10,000 elements. This behavior is consistent with the worst-case time complexity O(n^2)), as the deterministic pivot choice (last element) leads to unbalanced partitions in pre-sorted data.
4. **Reverse Sorted Array**: Similar to sorted arrays, reverse-sorted arrays also incurred higher sorting times, particularly as the array size increased. This is due to the same unbalanced partitioning effect.
5. **Array with Repeated Elements**: Arrays with repeated elements demonstrated relatively efficient performance, as the presence of identical values allows quicker partitioning. For a 10,000-element array, the algorithm sorted in approximately 0.25 seconds.

Observations:
The performance results align with theoretical expectations:

- Random and repeated-element arrays perform close to the average-case O(nlogn).
- Sorted and reverse-sorted arrays approach the worst-case O(n^2), showing substantial performance degradation for larger inputs.

These findings underscore the deterministic Quicksort's sensitivity to input order. We expect the randomized Quicksort to mitigate the worst-case performance by diversifying pivot selection.

Performance Analysis

Quicksort is known for its efficiency, but its performance can vary depending on the order of input elements and pivot selection. Here, we analyze the time complexity in different cases and the algorithm's space complexity.

1. **Time Complexity**:
    - **Best Case**: The best-case time complexity occurs when the pivot divides the array into two nearly equal halves at each recursive step. In this scenario, the time complexity is

O(nlogn) because the recursive division only occurs log(n) times, and each level requires O(n) comparisons.

- **Average Case**: The average case also results in O(nlogn) time complexity. This efficiency stems from the probability that, on average, the pivot will partition the array fairly evenly. As a result, the recursive depth is approximately log(n) levels, with each level requiring O(n) operations.
- **Worst Case**: The worst-case time complexity of Quicksort is O(n^2) . This scenario occurs when the pivot always partitions the array into one large subarray and one empty subarray (e.g., when the array is already sorted). In this case, there are n levels of recursion, with each level requiring O(n) operations.

2. **Space Complexity**:
   - Quicksort is an in-place sorting algorithm, so it doesn't require extra space for data storage. However, it requires stack space for recursive function calls. The **space complexity** in the average and best cases is O(logn) because the recursive depth is approximately log(n).
   - In the worst case, the space complexity becomes O(n) due to the deep recursion stack when the array is already sorted or reverse-sorted.
3. **Additional Overheads**:
   - **Partitioning Overhead**: Partitioning the array around the pivot requires comparisons and swaps, with a cost of O(n) per recursive level.
   - **Recursion Overhead**: The recursive calls introduce a stack overhead. This is minimized by choosing pivots effectively or using a randomized approach to reduce the likelihood of encountering the worst-case scenario.

Randomized Quicksort Implementation

To improve the performance of Quicksort on already sorted or reverse-sorted arrays, a randomized version can be used. In randomized Quicksort, we select the pivot randomly from the current subarray, reducing the likelihood of encountering unbalanced partitions.

Code for Randomized Quicksort (already provided):

```
def randomized_quicksort(arr, low, high):
  if low < high:
     pivot_index = randomized_partition(arr, low, high)
     randomized_quicksort(arr, low, pivot_index - 1)
     randomized_quicksort(arr, pivot_index + 1, high)

def randomized_partition(arr, low, high):
  pivot_index = random.randint(low, high)
  arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
  pivot = arr[high]
  i = low - 1
  for j in range(low, high):
     if arr[j] < pivot:
         i += 1
         arr[i], arr[j] = arr[j], arr[i]
  arr[i + 1], arr[high] = arr[high], arr[i + 1]
  return i + 1
```

Testing Randomized Quicksort

Now, let's run the randomized Quicksort tests using the same test arrays you used for deterministic Quicksort. You can use the provided TestRandomizedQuickSort.py to check performance on empty, random, sorted, reverse-sorted, and repeated-element arrays.

```
from RandomizedQuickSort import randomized_quicksort

import time
import random
import sys

# Increase recursion limit
sys.setrecursionlimit(5000)  # Adjust this value as needed based on array size

# Define a function to measure execution time of randomized quicksort
def measure_sort_time(arr):
    start_time = time.time()
    randomized_quicksort(arr, 0, len(arr) - 1)
    end_time = time.time()
    return end_time - start_time

# Randomized Quicksort functions (same as before)
def randomized_quicksort(arr, low, high):
    if low < high:
        pivot_index = randomized_partition(arr, low, high)
        randomized_quicksort(arr, low, pivot_index - 1)
        randomized_quicksort(arr, pivot_index + 1, high)

def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Test arrays with different cases
array_sizes = [1000, 5000, 10000]  # Testing with different input sizes

# 1. Empty array test
empty_array = []
time_empty = measure_sort_time(empty_array)
print(f"Empty array sorted in: {time_empty:.6f} seconds")

# Loop through other test arrays with increasing sizes
for size in array_sizes:
    # 2. Randomly generated array
    random_array = [random.randint(1, 10000) for _ in range(size)]
    time_random = measure_sort_time(random_array)
    print(f"Random array of size {size} sorted in: {time_random:.6f} seconds")

    # 3. Already sorted array
    sorted_array = list(range(1, size + 1))
    time_sorted = measure_sort_time(sorted_array)
    print(f"Sorted array of size {size} sorted in: {time_sorted:.6f} seconds")

    # 4. Reverse-sorted array
    reverse_sorted_array = list(range(size, 0, -1))
    time_reverse_sorted = measure_sort_time(reverse_sorted_array)
    print(f"Reverse sorted array of size {size} sorted in: {time_reverse_sorted:.6f} seconds")

    # 5. Array with repeated elements
    repeated_elements_array = [random.choice([1, 2, 3, 4, 5]) for _ in range(size)]
    time_repeated = measure_sort_time(repeated_elements_array)
    print(f"Array with repeated elements of size {size} sorted in: {time_repeated:.6f} seconds")

    print("-" * 40)
```

```
python3 TestRandomizedQuickSort.py
Empty array sorted in: 0.000001 seconds
```

```
Random array of size 1000 sorted in: 0.001342 seconds
Sorted array of size 1000 sorted in: 0.001319 seconds
Reverse sorted array of size 1000 sorted in: 0.001293 seconds
Array with repeated elements of size 1000 sorted in: 0.004891 seconds
----------------------------------------
Random array of size 5000 sorted in: 0.007000 seconds
Sorted array of size 5000 sorted in: 0.006445 seconds
Reverse sorted array of size 5000 sorted in: 0.005823 seconds
Array with repeated elements of size 5000 sorted in: 0.067382 seconds
----------------------------------------
Random array of size 10000 sorted in: 0.011307 seconds
Sorted array of size 10000 sorted in: 0.010288 seconds
Reverse sorted array of size 10000 sorted in: 0.010567 seconds
Array with repeated elements of size 10000 sorted in: 0.251032 seconds
```

Empirical Analysis

The randomized version of Quicksort was tested under the same conditions as the deterministic version, measuring execution times across different array types and sizes. Here's a comparison of the results:

1. **Empty Array**: Like the deterministic version, the randomized Quicksort handles an empty array in negligible time, indicating no overhead when there are no elements to sort.
2. **Random Array**: Randomized Quicksort maintained efficient performance with random arrays. Sorting a 1,000-element array took approximately 0.0013 seconds, while a 10,000-element array completed in 0.011 seconds. This efficiency aligns with the expected average-case time complexity of O(nlogn).
3. **Sorted and Reverse-Sorted Arrays**:
   The randomized Quicksort showed significant improvements on already sorted and reverse-sorted arrays. For example, the 10,000-element sorted array completed in approximately 0.010 seconds with randomized Quicksort, compared to 2.88 seconds with deterministic Quicksort. This improvement is due to the randomized pivot selection, which reduces the likelihood of unbalanced partitions and mitigates the O(n^2)worst-case scenario common in the deterministic approach.
4. **Array with Repeated Elements**:
   Randomized Quicksort's performance with repeated elements was similar to that of the deterministic version for larger arrays. Sorting a 10,000-element array with repeated elements took approximately 0.25 seconds in both cases, as repeated values can still lead to balanced partitions.

Deterministic and randomized Quicksort

```
import random
```

```python
def randomized_quicksort(arr, low, high):

  if low < high:

      pivot_index = randomized_partition(arr, low, high)

      randomized_quicksort(arr, low, pivot_index - 1)

      randomized_quicksort(arr, pivot_index + 1, high)



def randomized_partition(arr, low, high):

  pivot_index = random.randint(low, high)

  arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

  pivot = arr[high]

  i = low - 1

  for j in range(low, high):

      if arr[j] < pivot:

          i += 1

          arr[i], arr[j] = arr[j], arr[i]

  arr[i + 1], arr[high] = arr[high], arr[i + 1]

  return i + 1
```

```
python3 CompareQuickSort.py

Empty array sorted in (Randomized): 0.000001 seconds

Empty array sorted in (Deterministic): 0.000000 seconds
```

```
---------------------------------------
Random array of size 1000 sorted in (Randomized): 0.001464 seconds
Random array of size 1000 sorted in (Deterministic): 0.001070 seconds
Sorted array of size 1000 sorted in (Randomized): 0.001269 seconds
Sorted array of size 1000 sorted in (Deterministic): 0.036064 seconds
Reverse sorted array of size 1000 sorted in (Randomized): 0.000914 seconds
Reverse sorted array of size 1000 sorted in (Deterministic): 0.021745 seconds
Array with repeated elements of size 1000 sorted in (Randomized): 0.003136 seconds
Array with repeated elements of size 1000 sorted in (Deterministic): 0.006022 seconds
---------------------------------------
Random array of size 5000 sorted in (Randomized): 0.005016 seconds
Random array of size 5000 sorted in (Deterministic): 0.004449 seconds
Sorted array of size 5000 sorted in (Randomized): 0.005103 seconds
Sorted array of size 5000 sorted in (Deterministic): 0.720127 seconds
Reverse sorted array of size 5000 sorted in (Randomized): 0.004711 seconds
Reverse sorted array of size 5000 sorted in (Deterministic): 0.519502 seconds
Array with repeated elements of size 5000 sorted in (Randomized): 0.065004 seconds
Array with repeated elements of size 5000 sorted in (Deterministic): 0.141356 seconds
---------------------------------------
Random array of size 10000 sorted in (Randomized): 0.010641 seconds
Random array of size 10000 sorted in (Deterministic): 0.008462 seconds
Sorted array of size 10000 sorted in (Randomized): 0.010720 seconds
Sorted array of size 10000 sorted in (Deterministic): 2.888985 seconds
Reverse sorted array of size 10000 sorted in (Randomized): 0.010720 seconds
Reverse sorted array of size 10000 sorted in (Deterministic): 2.064753 seconds

Array with repeated elements of size 10000 sorted in (Randomized): 0.251301 seconds

Array with repeated elements of size 10000 sorted in (Deterministic): 0.560851 seconds
```

Empirical Analysis

The performance comparison between deterministic and randomized Quicksort was conducted on various data distributions and sizes. Here is a summary of the findings:

1. **Empty Array**: Both deterministic and randomized versions handled the empty array in negligible time, as expected.
2. **Random Array**: Performance for random arrays was similar across both implementations, with minor differences. For example, a random array of 10,000 elements took approximately 0.008 seconds with deterministic Quicksort and 0.011 seconds with randomized Quicksort, both aligning with the O(nlogn) average-case complexity.
3. **Sorted and Reverse-Sorted Arrays**:
   The randomized Quicksort demonstrated a substantial improvement on sorted and reverse-sorted arrays, where the deterministic Quicksort struggled:
   - For a sorted array of 10,000 elements, deterministic Quicksort took 2.89 seconds, whereas randomized Quicksort completed the task in 0.010 seconds.
   - Similarly, for a reverse-sorted array of the same size, deterministic Quicksort took 2.06 seconds, while randomized Quicksort managed it in 0.010 seconds.
4. These results confirm that randomized pivot selection effectively mitigates the worst-case O(n^2) scenario in cases where input data is ordered or nearly ordered. This performance boost makes randomized Quicksort significantly faster and more reliable for such distributions.
5. **Array with Repeated Elements**:
   With arrays containing repeated elements, randomized Quicksort performed better but not as significantly as with sorted arrays. For example, sorting 10,000 repeated elements took 0.56 seconds with deterministic Quicksort versus 0.25 seconds with randomized Quicksort. While repeated elements can still lead to balanced partitions in both versions, random pivot selection remains advantageous.

Array Type	**Array Size**	**Deterministic Quicksort (seconds)**	**Randomized Quicksort (seconds)**
Empty Array	-	0.000000	0.000001
Random Array	1,000	0.001070	0.001464
	5,000	0.004449	0.005016
	10,000	0.008462	0.010641
Sorted Array	1,000	0.036064	0.001269
	5,000	0.720127	0.005103
	10,000	2.888985	0.010720
Reverse-Sorted Array	1,000	0.021745	0.000914
	5,000	0.519502	0.004711
	10,000	2.064753	0.010720
Array with Repeated Elements	1,000	0.006022	0.003136

	5,000	0.141356	0.065004
	10,000	0.560851	0.251301

Summary of Findings:

These results highlight the limitations of deterministic Quicksort in scenarios with ordered data, where it falls back to O(n^2) performance. In contrast, randomized Quicksort consistently performed closer to O(nlogn), making it a more efficient and versatile choice, especially for data with regular patterns or ordering.

Conclusion

This study explored both deterministic and randomized implementations of the Quicksort algorithm, analyzing their performance across various input distributions. While the deterministic Quicksort is simple and effective for random data, its fixed pivot selection can lead to significant performance degradation when applied to already sorted or reverse-sorted data, resulting in O(n^2)) time complexity in the worst case. In contrast, the randomized Quicksort mitigates this limitation by selecting a pivot randomly, thus reducing the likelihood of unbalanced partitions.

Empirical testing demonstrated that the randomized approach consistently achieved faster execution times for sorted and reverse-sorted arrays, maintaining performance close to the ideal $O(nlogn)$ time complexity. This improvement highlights the robustness of randomized Quicksort, making it a preferred choice when input data may exhibit order or patterns that could lead to worst-case scenarios.

In conclusion, randomized Quicksort provides a flexible and efficient solution for sorting diverse data distributions. Its balanced performance across different cases makes it a valuable tool in developing scalable and optimized algorithms, especially in applications requiring fast, in-place sorting with minimal memory overhead.

Conclusion

This study explored both deterministic and randomized implementations of the Quicksort algorithm, analyzing their performance across various input distributions. While the deterministic Quicksort is simple and effective for random data, its fixed pivot selection can lead to significant performance degradation when applied to already sorted or reverse-sorted data, resulting in $O(n^2)$) time complexity in the worst case. In contrast, the randomized Quicksort mitigates this limitation by selecting a pivot randomly, thus reducing the likelihood of unbalanced partitions.

Empirical testing demonstrated that the randomized approach consistently achieved faster execution times for sorted and reverse-sorted arrays, maintaining performance close to the ideal $O(nlogn)$ time complexity. This improvement highlights the robustness of randomized Quicksort, making it a preferred choice when input data may exhibit order or patterns that could lead to worst-case scenarios.

In conclusion, randomized Quicksort provides a flexible and efficient solution for sorting diverse data distributions. Its balanced performance across different cases makes it a valuable tool in developing scalable and optimized algorithms, especially in applications requiring fast, in-place sorting with minimal memory overhead.

References

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

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, *5*(1), 10–16. https://doi.org/10.1093/comjnl/5.1.10

Sedgewick, R. (1978). Implementing Quicksort Programs. *Communications of the ACM*, *21*(10), 847-857. https://doi.org/10.1145/359619.359631