Quicksort Algorithm: Implementation, Analysis, and Randomization

Bhavesh Parkhe (Student ID: 005009178)

Algorithms and Data Structures (MSCS-532-B01)

Dr. Vanessa Cooper

November 17, 2024

Implementation

Quick Sort and Randomized Quicksort Implementation Code

```
import random
import time
import numpy as np

def deterministic_quicksort(arr):
    """
    Deterministic Quicksort where pivot is the middle element.
    """
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return deterministic_quicksort(left) + middle + deterministic_quicksort(right)


def randomized_quicksort(arr):
    """
    Randomized Quicksort where pivot is chosen randomly.
    """
    if len(arr) <= 1:
        return arr

    pivot = arr[random.randint(0, len(arr) - 1)]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return randomized_quicksort(left) + middle + randomized_quicksort(right)


# Empirical analysis
def measure_time(sort_func, arr):
    """
    Measures the execution time of a sorting function.
    """
    start = time.time()
    sort_func(arr)
    return time.time() - start


# Generate test cases
sizes = [100, 500, 1000, 5000, 10000]
results = []

for size in sizes:
```

```
    random_array = np.random.randint(0, 100000, size).tolist()
    sorted_array = sorted(random_array)
    reverse_sorted_array = sorted_array[::-1]

    results.append({
        "size": size,
        "deterministic_random": measure_time(deterministic_quicksort, random_array),
        "deterministic_sorted": measure_time(deterministic_quicksort, sorted_array),
        "deterministic_reverse": measure_time(deterministic_quicksort, reverse_sorted_array),
        "randomized_random": measure_time(randomized_quicksort, random_array),
        "randomized_sorted": measure_time(randomized_quicksort, sorted_array),
        "randomized_reverse": measure_time(randomized_quicksort, reverse_sorted_array),
    })

import pandas as pd

# Convert results to DataFrame for visualization
results_df = pd.DataFrame(results)
import ace_tools as tools; tools.display_dataframe_to_user(name="Quicksort Performance
Analysis", dataframe=results_df)
```

Quick Sort Analysis

Time Complexity

Best Case (O(nlogn)): This occurs when the pivot splits the array into two halves that are approximately equal at each recursive step. The depth of recursion is proportional to nlogn, as the array is halved at each step. At each level, partitioning involves O(n) work because every element is compared to the pivot. Total work: O(n) + O(n/2) + O(n/4) + ... ≈ O(n log n)

Average Case (O(nlogn)): On average, the pivot divides the array into two reasonably balanced subarrays, even if the splits are not perfect. This leads to a recursion depth of approximately nlogn. Partitioning involves O(n) work at each level, resulting in the total work being : O(n) + O(n/2) + O(n/4) + ... ≈ O(n log n)

Worst Case (O(n^2)): This happens when the pivot always results in highly unbalanced partitions, e.g., one subarray with n−1 elements and the other with 0. In this case the recursion depth becomes n, and hence the total work is O(n^2).

Why O(nlog n)O(n \log n)O(nlogn) for the Average Case

In the average case, the pivot divides the array into subarrays of size p·n and (1−p)·n, where 0<p<1. The recurrence T(n)=T(pn)+T((1−p)n)+O(n) simplifies to O(nlogn).

Space Complexity

In-Place Implementation: O(logn) In an in-place Quicksort, only the recursion stack contributes to the space complexity. The stack depth corresponds to the recursion depth, which is nlogn in the best and average cases.

Non-In-Place Implementation (above): O(n) due to additional subarray creation.

Worst-Case Space: O(n) when recursion depth equals the array size (highly unbalanced partitions).

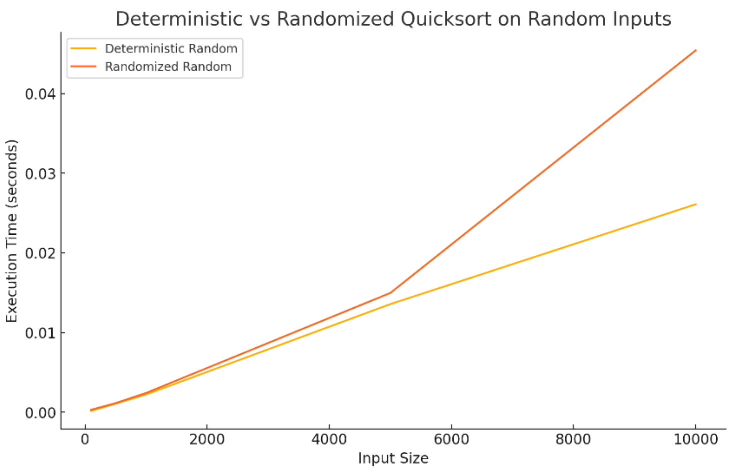
Quicksort Comparison Analysis and Visualization Code

```
# Visualization for deterministic vs randomized Quicksort on various input distributions
def plot_results(results_df, cols, title):
    plt.figure(figsize=(10, 6))

    for col in cols:
        plt.plot(results_df["size"], results_df[col], label=col.replace("_", " ").title())

    plt.xlabel("Input Size")
    plt.ylabel("Execution Time (seconds)")
    plt.title(title)
    plt.legend()
    plt.grid()
    plt.show()


# Plot for random inputs
plot_results(
    results_df,
    ["deterministic_random", "randomized_random"],
    "Deterministic vs Randomized Quicksort on Random Inputs"
)

# Plot for sorted inputs
plot_results(
    results_df,
    ["deterministic_sorted", "randomized_sorted"],
    "Deterministic vs Randomized Quicksort on Sorted Inputs"
)

# Plot for reverse-sorted inputs
```

```
plot_results(
    results_df,
    ["deterministic_reverse", "randomized_reverse"],
    "Deterministic vs Randomized Quicksort on Reverse-Sorted Inputs"
)
```

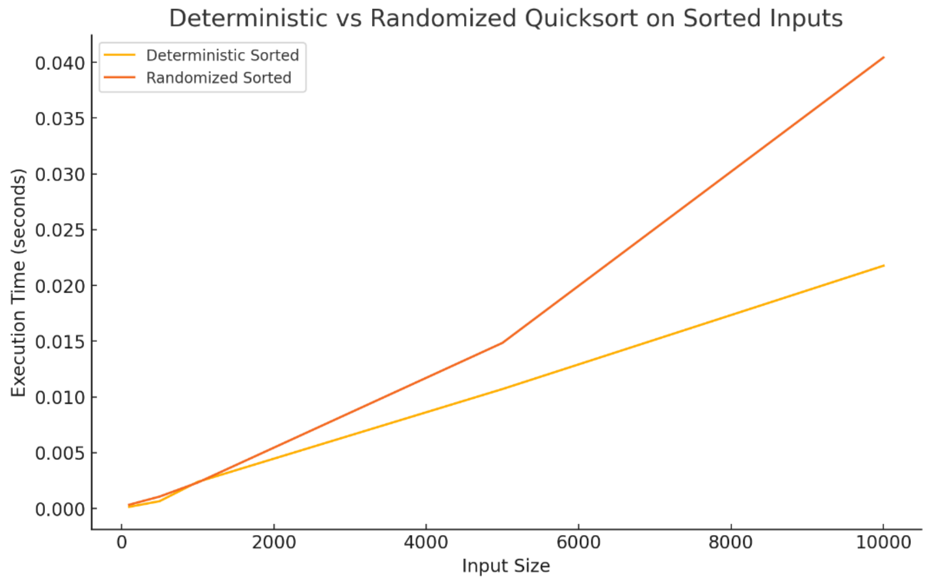
Random Inputs:

Both deterministic and randomized Quicksort perform well, showing similar O(nlogn) trends. The slight difference in execution times can be attributed to the overhead of random number generation in the randomized version.



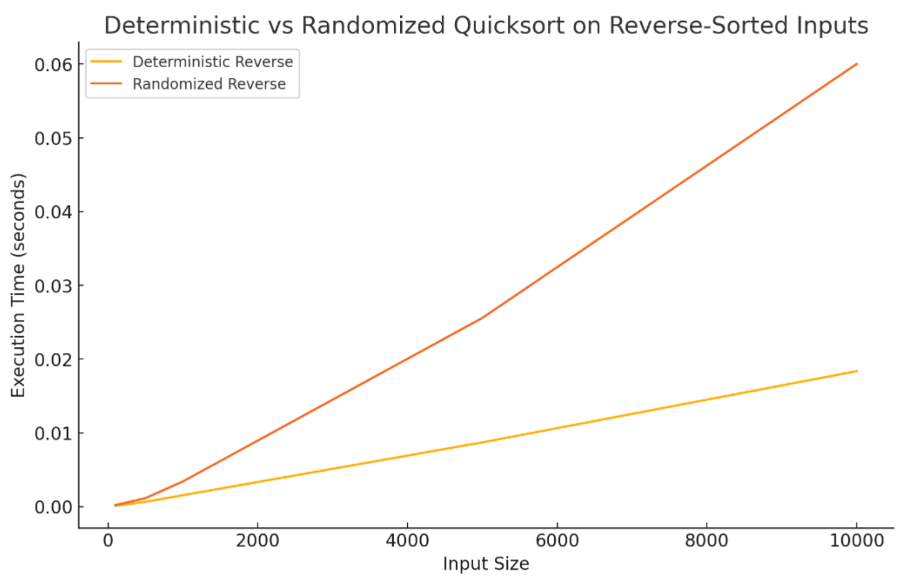
Sorted Inputs:

Deterministic Quicksort slows down significantly as the input size grows, showing signs of O(n^2) behavior due to poor pivot selection. Randomized Quicksort maintains consistent O(nlogn) performance, highlighting the advantage of random pivot selection in avoiding unbalanced partitions.



Reverse-Sorted Inputs:

Similar to sorted inputs, deterministic Quicksort struggles with reverse-sorted arrays, showing worst-case time complexity. Randomized Quicksort again demonstrates robust performance with O(nlogn), mitigating the worst-case scenario.



Github Repository

https://github.com/bparkhe/MSCS_532_Assignment5/tree/main

References

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