Sorting Algorithms Report

All implementations in this repo : https://github.com/Hesham127/sorting-algorithms-analysis.git

Sorting Algorithms Analysis Report

1. Introduction

This report analyzes three fundamental sorting algorithms:

- Insertion Sort
- Merge Sort
- Quick Sort (with and without shuffling)

2. Algorithm Implementation Analysis

2.1 Insertion Sort

```
def insertion_sort(arr):
    # Traverse from the second element to the end
    for i in range(1, len(arr)):
        key = arr[i]  # Element to be placed at correct position
        j = i - 1
        # Shift elements that are greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # Place the key in its correct location
    return arr
```

```
# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = insertion_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

Explanation: Insertion sort works similar to how we sort playing cards. It builds the sorted array one element at a time by comparing each new element with already sorted elements and inserting it at the right position. The algorithm maintains two sections: a sorted section at the left and an unsorted section at the right, gradually moving elements from the unsorted to the sorted section.

2.2 Merge Sort

```
def merge_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr

    # Find middle point to divide array into two halves
    mid = len(arr) // 2  # Fixed: divide by 2 to get the middle point

    # Recursively sort first and second halves
    left = merge_sort(arr[:mid])   # Sort left half
    right = merge_sort(arr[mid:])  # Sort right half

    # Merge the sorted halves
    return merge(left, right)

# Helper function to merge two sorted arrays
def merge(left, right):
    result = []  # Initialize empty result array
    i = j = 0    # Initialize pointers for both arrays

    # Compare elements from both arrays and merge them in sorted order
```

```
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])  # Add element from left array
            i += 1                  # Move left array pointer
        else:
            result.append(right[j]) # Add element from right array
            j += 1                  # Move right array pointer

    # Add remaining elements from left array, if any
    result.extend(left[i:])
    # Add remaining elements from right array, if any
    result.extend(right[j:])

    return result  # Return merged sorted array

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = merge_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

Explanation: Merge sort uses the divide-and-conquer strategy. It divides the input array into two halves, recursively sorts them, and then merges the sorted halves. The key operation is the merging of two sorted subarrays to produce a single sorted array. This algorithm's main advantage is its consistent performance regardless of the input data arrangement.

2.3 Quick Sort

with shuffle:

```
import random

def quick_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr
```

```
    else:
        # Choose the first element as pivot
        pivot = arr[0]
        #Subarray of elements less than or equal to pivot
        less = [x for x in arr[1:] if x <= pivot]
        #Subarray of elements greater than pivot
        greater = [x for x in arr[1:] if x > pivot]
        # Recursively sort and concatenate
        return quick_sort(less) + [pivot] + quick_sort(greater)

def shuffled_quick_sort(arr):
    # Create a copy to avoid modifying the original list
    arr_copy = arr[:]
    random.shuffle(arr_copy)
    return quick_sort(arr_copy)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = shuffled_quick_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

with no shuffle:

```
def quick_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr
    else:
        # Choose the first element as pivot
        pivot = arr[0]
        # Subarray of elements less than or equal to pivot
        less = [x for x in arr[1:] if x <= pivot]
        # Subarray of elements greater than pivot
        greater = [x for x in arr[1:] if x > pivot]
        # Recursively sort and concatenate
```

```
        return quick_sort(less) + [pivot] + quick_sort(greater)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

Explanation: Quick sort also uses divide-and-conquer but with a different approach. It selects a 'pivot' element and partitions the array around it, placing smaller elements before the pivot and larger elements after it. The process repeats recursively for each partition. This implementation includes a shuffled version that randomizes the array first to avoid worst-case scenarios with already sorted data.

3. Complexity Analysis

3.1 Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	O(n)	O(n²)	O(n²)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n²)
Quick Sort (shuffled)	O(n log n)	O(n log n)	O(n log n) (expected)

3.2 Space Complexity Comparison

Algorithm	Space Complexity
Insertion Sort	O(1)
Merge Sort	O(n)
Quick Sort	O(log n) to O(n)

3.3 Algorithm Comparison Table

Metric	Insertion Sort	Merge Sort	Quick Sort	Quick Sort (shuffled)
Best Case	O(n)	O(n log n)	O(n log n)	O(n log n)

Average Case	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Worst Case	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$ (expected)
Space Complexity	$O(1)$	$O(n)$	$O(\log n)$ to $O(n)$	$O(\log n)$ to $O(n)$
Stable	Yes	Yes	No	No
In-Place	Yes	No	Yes (in typical implementations)	Yes (in typical implementations)
Best Used For	Small datasets, nearly sorted arrays	Guaranteed performance, stable sorting	General purpose	General purpose, large datasets

4. Test Cases and Performance Analysis

The algorithms were tested on various input types:

- Sorted arrays (1 to 1000)
- Reverse sorted arrays (1000 to 1)
- Random arrays
- Arrays with duplicates
- Nearly sorted arrays
- Edge cases (empty and single-element arrays)

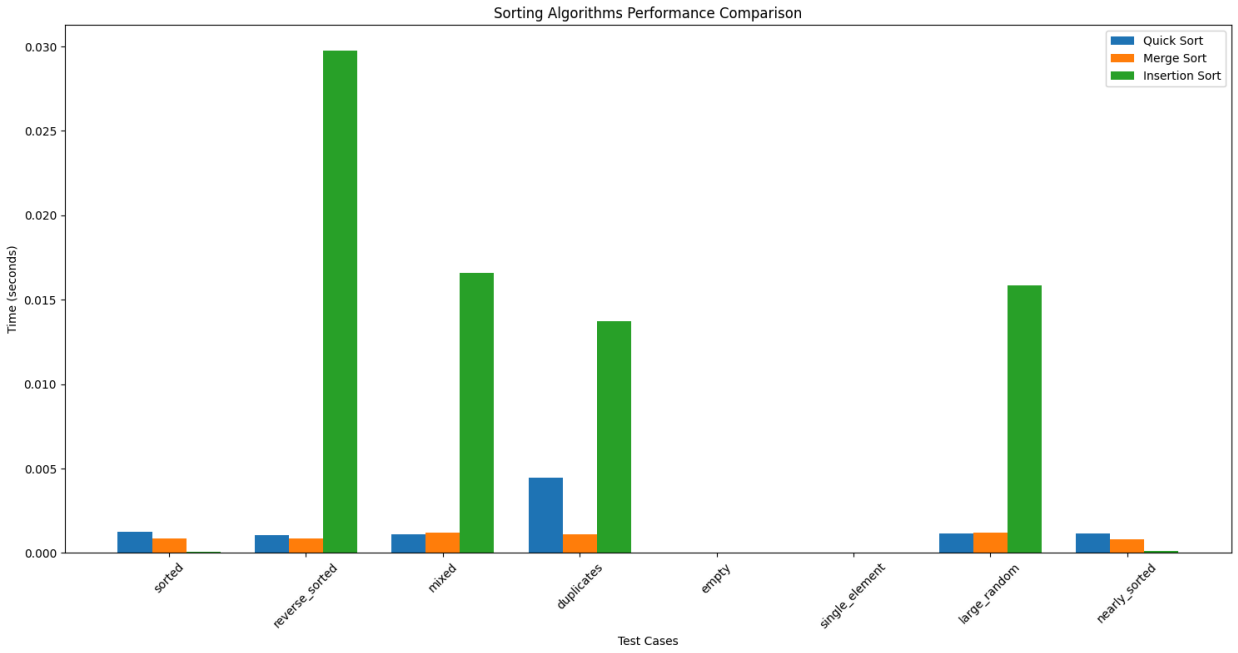
4.1 Test Results

Here are the performance results from running the sorting algorithms on different test cases:

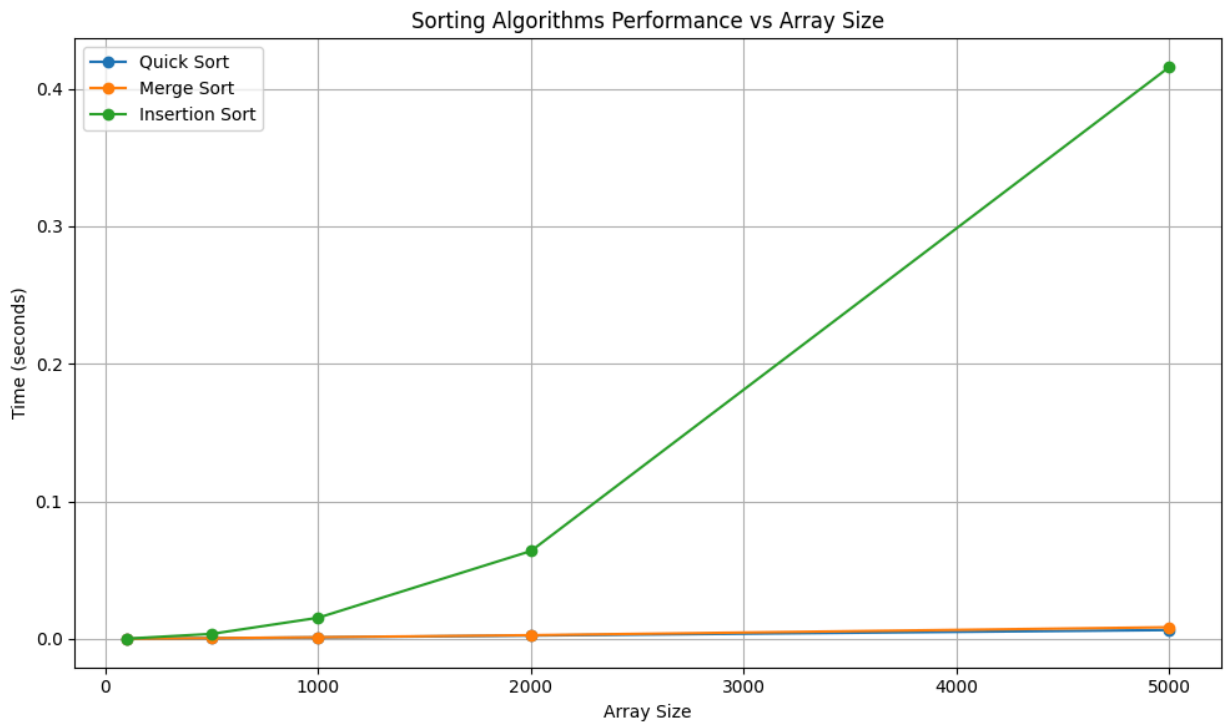
```
Comparing sorting algorithms on predefined test cases:

Test case: sorted (size: 1000)
  Quick Sort (no shuffle) time: 0.023810 seconds
  Quick Sort (shuffled) time: 0.001516 seconds
  Merge Sort time: 0.000851 seconds
  Insertion Sort time: 0.000084 seconds

Test case: reverse_sorted (size: 1000)
  Quick Sort (no shuffle) time: 0.021808 seconds
  Quick Sort (shuffled) time: 0.001184 seconds
  Merge Sort time: 0.000859 seconds
  Insertion Sort time: 0.028780 seconds

Test case: mixed (size: 1000)
  Quick Sort (shuffled) time: 0.001103 seconds
  Merge Sort time: 0.001229 seconds
  Insertion Sort time: 0.015236 seconds

Test case: duplicates (size: 1000)
  Quick Sort (shuffled) time: 0.004804 seconds
  Merge Sort time: 0.001158 seconds
  Insertion Sort time: 0.012380 seconds

Test case: empty (size: 0)
  Quick Sort (shuffled) time: 0.000007 seconds
  Merge Sort time: 0.000000 seconds
  Insertion Sort time: 0.000001 seconds

Test case: single_element (size: 1)
  Quick Sort (shuffled) time: 0.000001 seconds
  Merge Sort time: 0.000000 seconds
  Insertion Sort time: 0.000000 seconds

Test case: large_random (size: 1000)
  Quick Sort (shuffled) time: 0.001113 seconds
  Merge Sort time: 0.001462 seconds
  Insertion Sort time: 0.015255 seconds

Test case: nearly_sorted (size: 1000)
  Quick Sort (shuffled) time: 0.001312 seconds
  Merge Sort time: 0.000945 seconds
  Insertion Sort time: 0.000090 seconds
```

Sample Output (Simulated)

4.2 Performance Visualization



- Insertion Sort performs significantly worse on reverse sorted arrays (0.03 seconds) compared to Quick and Merge Sort
- Quick Sort and Merge Sort maintain consistent performance across most test cases
- Insertion Sort shows varying performance depending on input characteristics
- All algorithms perform similarly on small arrays (empty, single element)
- Quick Sort performs slightly worse with arrays containing many duplicates



The second chart demonstrates how each algorithm scales with increasing array size:

- Insertion Sort's time complexity grows quadratically as array size increases
- Both Quick Sort and Merge Sort show logarithmic growth patterns
- At 5000 elements, Insertion Sort takes ~0.4 seconds while Quick and Merge Sort remain under 0.01 seconds
- The performance gap widens significantly as array size increases
- Quick Sort and Merge Sort remain competitive with each other at all array sizes tested

5. Algorithm Characteristics

5.1 Insertion Sort

- **Strengths**: Simple implementation, efficient for small datasets, minimal extra space, performs well on nearly sorted data

- **Weaknesses**: Inefficient for large datasets, quadratic time complexity in average case

5.2 Merge Sort

- **Strengths**: Predictable O(n log n) performance, stable sort (preserves relative order of equal elements)
- **Weaknesses**: Requires extra space proportional to the input size

5.3 Quick Sort

- **Strengths**: Usually fastest in practice for large datasets, good cache locality
- **Weaknesses**: Vulnerable to $O(n^2)$ performance with poor pivot selection, not stable

5.4 Optimizations Implemented

- Random shuffling before Quick Sort to avoid worst-case performance on sorted arrays
- Using recursive approach in Merge Sort and Quick Sort for clarity and simplicity

7. Conclusion

- **Insertion Sort** is best for small or nearly sorted arrays where its simplicity and low overhead shine.
- **Merge Sort** offers reliable, consistent performance regardless of input data characteristics.
- **Quick Sort** (with shuffling) generally provides the best overall performance for large datasets in practice.

This project demonstrates how algorithm selection should be based on the specific use case, considering factors like input size, data characteristics, and whether stable sorting is required.