Comparison of Sorting Algorithms

Introduction

Sorting algorithms are fundamental to computer science and are used in various applications to organize and analyze data. This document compares six popular sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort. The comparison focuses on their time complexities, space complexities, and use cases.

How To Run

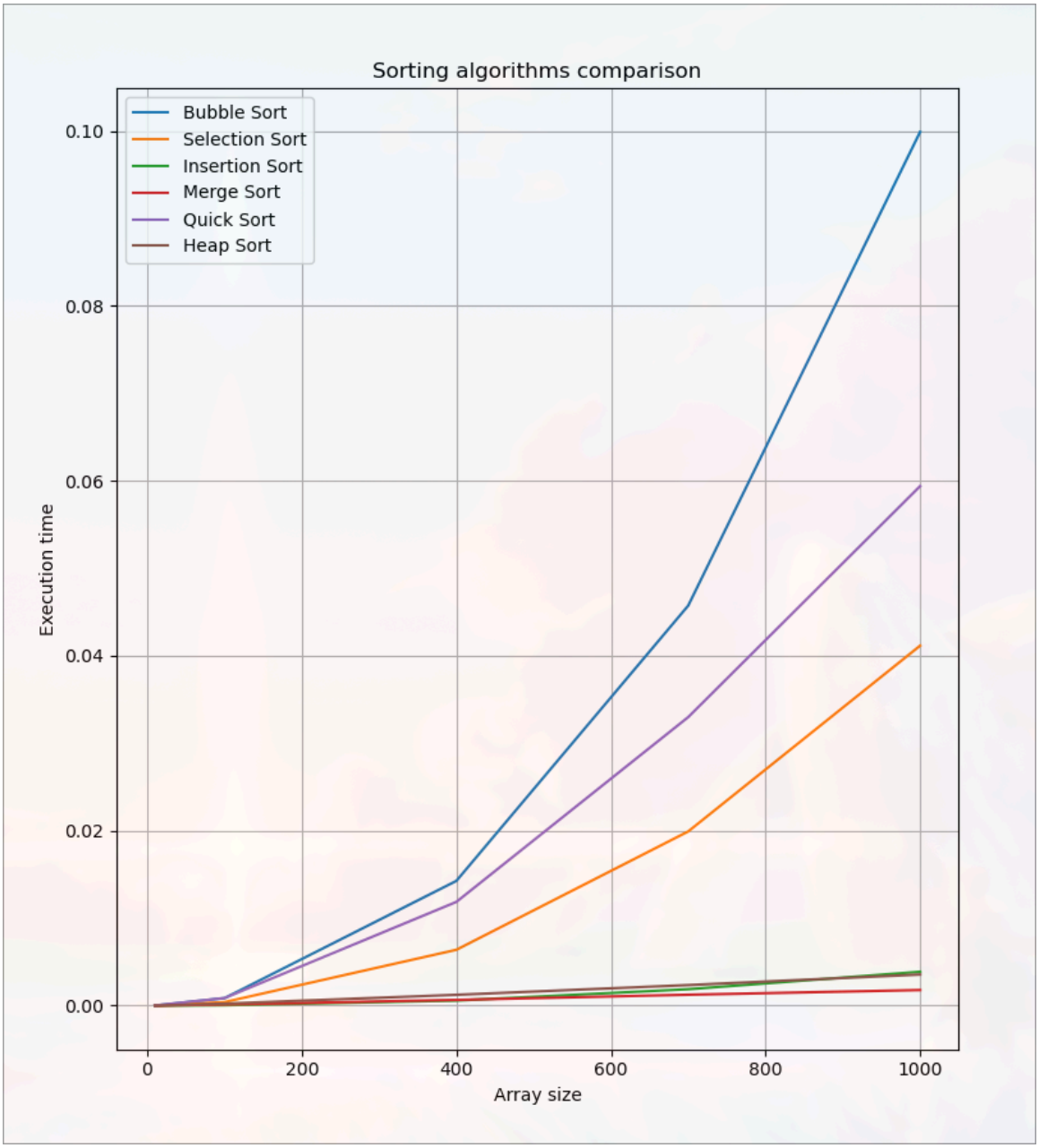
Prerequisites

- numpy
- matplotlib
- python

Command

```
python -m lab1.sorting
```

Output



This graph shows the execution time of different sorting algorithms for varying array sizes.

Bubble Sort

Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Time Complexity: O(n^2)

Space Complexity: O(1)

Use Case: Best suited for small datasets or educational purposes.

Selection Sort

Selection Sort divides the list into a sorted and unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and places it in the sorted region.

Time Complexity: O(n^2)

Space Complexity: O(1)

Use Case: Useful when memory writes are costly.

Insertion Sort

Insertion Sort builds the sorted list one element at a time by repeatedly inserting the next element into its correct position.

Time Complexity: O(n^2) (average), O(n) (best case)

Space Complexity: O(1)

Use Case: Efficient for small or nearly sorted datasets.

Merge Sort

Merge Sort is a divide-and-conquer algorithm that splits the list into smaller sublists, recursively sorts them, and merges them back together.

Time Complexity: O(n log n)

Space Complexity: O(n)

Use Case: Best for large datasets requiring stable sorting.

Quick Sort

Quick Sort is a divide-and-conquer algorithm that partitions the list around a pivot element, placing smaller elements to the left and larger ones to the right, and recursively sorts the partitions.

Time Complexity: O(n log n) (average), O(n^2) (worst case)

Space Complexity: O(log n) (average)

Use Case: Widely used due to its average-case efficiency.

Heap Sort

Heap Sort uses a binary heap data structure to repeatedly extract the maximum (or minimum) element and rebuild the heap until the list is sorted.

Time Complexity: O(n log n)

Space Complexity: O(1)

Use Case: Good for in-place sorting without recursion.

Conclusion

Each sorting algorithm has its strengths and weaknesses. The choice of algorithm depends on the size and nature of the dataset, as well as performance requirements. While simple algorithms like Bubble Sort and Selection Sort are easy to understand, advanced algorithms like Quick Sort and Merge Sort are more suitable for larger datasets. Following table summarizes the time complexity of the different algorithms:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)