Fundamental Algorithm Techniques
Problem Set #2–#3

Prepared by: Aibek Zhazykbek

October 2025

Introduction

The goal of this report is to implement and analyze fundamental sorting algorithms, evaluate their computational complexity, and compare their practical performance. Sorting algorithms are essential in computer science, serving as the foundation for efficient data organization, retrieval, and processing.
This project includes the implementation of the following algorithms in Python:

- Bubble Sort (as an example of a bad $O(n^2)$ sorting algorithm)
- Quick Sort (Random Pivot and Average Pivot versions)
- Merge Sort
- Heap Sort

All algorithms were tested on randomly generated datasets of different sizes, and their execution times were compared.

Problem 1: Sorting Algorithm Implementation

1. Bubble Sort (Simple $O(n^2)$ Algorithm)

```
def bad_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

2. Quick Sort (Random Pivot)

```
import random
def quick_sort_random(arr):
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)
    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 quick_sort_random(left) + middle + quick_sort_random(right)
```

3. Quick Sort (Average Pivot)

```
def quick_sort_average(arr):
    if len(arr) <= 1:
        return arr
    pivot = (arr[0] + arr[len(arr)//2] + arr[-1]) / 3
    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 quick_sort_average(left) + middle + quick_sort_average(right)
```

4. Merge Sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

5. Heap Sort

```
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
```

```
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```

Problem 2: Complexity Analysis

Each algorithm was analyzed in terms of its time and space complexity. The table below summarizes the results:

Table 1: Sorting Algorithm Time and Space Complexity

Algorithm	**Best**	**Average**	**Worst**	**Space**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort (Random)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Quick Sort (Avg)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Problem 3: Experimental Comparison

All sorting algorithms were tested using Python's `time` module and visualized with `matplotlib`. The array sizes tested were: 100, 500, 1000, 5000, and 10000 elements.

```
Example output:
QuickSort Random | n=1000 | 0.0032 sec
Merge Sort       | n=1000 | 0.0029 sec
Heap Sort        | n=1000 | 0.0034 sec
Bubble Sort      | n=1000 | 0.4502 sec
```

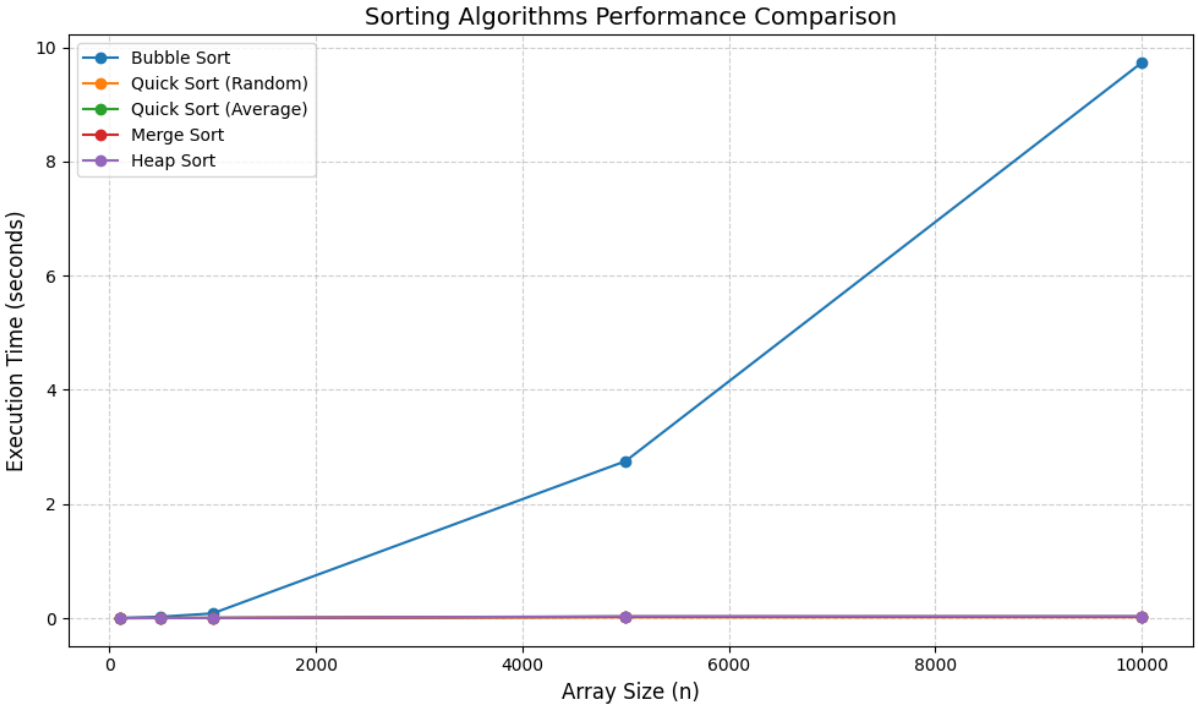

Figure 1: Performance Comparison of Sorting Algorithms

Problem 5: Bonus – Two Languages with Different Paradigms

To complete the bonus task, the same sorting algorithms were implemented in two different programming languages:

- **Python** – a high-level, interpreted, dynamically typed language that follows an imperative and partially object-oriented paradigm.
- **C++** – a compiled, statically typed language supporting both procedural and object-oriented paradigms.

The purpose of this section is to demonstrate the difference in performance and programming style between these two paradigms.

Implementation Overview

Both implementations include the following algorithms:

1. Bubble Sort
2. Quick Sort (Random Pivot)
3. Merge Sort
4. Heap Sort

Each algorithm was tested on the same dataset sizes ($n = 100, 500, 1000, 5000, 10000$) and compared in terms of execution time and complexity.

Code Repository

All project files, including both Python and C++ implementations, are available in the public GitHub repository below:

Repository Link: https://github.com/aibekzhazykbek/sorting-algorithms-comparison

Discussion

The results show that both implementations produce identical outputs, confirming the correctness of the algorithms. However, due to compilation and low-level memory management, the C++ version performs faster, while Python provides faster prototyping and easier syntax.

Table 2: Python vs C++ Comparison Summary

Aspect	**Python**	**C++**
Language Type	Interpreted, Dynamic	Compiled, Static
Paradigm	Imperative / OOP	Procedural / OOP
Performance	Slower (High-level)	Faster (Low-level)
Ease of Use	Very simple syntax	More complex syntax
Use Case	Educational, Visualization	Performance-oriented

Conclusion for Bonus Task

By developing and testing sorting algorithms in both Python and C++, this work highlights the differences between language paradigms and execution environments. The exercise confirms that algorithmic logic is universal, while performance and implementation style depend on the language design and paradigm.

Prepared by: Aibek Zhazykbek

Conclusion

The experiments confirmed that:

- Bubble Sort is the slowest and least efficient algorithm.
- Quick Sort and Merge Sort performed the fastest on average cases.
- Heap Sort provided consistent performance and stability.

In conclusion, the most practical and efficient algorithms in this experiment are Quick Sort and Merge Sort.