Ionian University

Comparative Analysis of Sorting Algorithms: Part 2

Department of Informatics

Laboratory of Algorithms and Data Structures

April 2, 2025

Names	Student ID
Nikolaos Roufas	inf2024146
Konstantinos Tzortis	inf2024168
Nikolaos Levadiotis	inf2024168

Contents

1 Introduction

This report presents a comprehensive analysis of various sorting algorithms and their performance characteristics when applied to different data structures. Specifically, we investigate how sorting algorithms behave when applied to:

1. Simple arrays of integers (Part A)
2. Arrays of custom record structures sorted by a string field (Part B)

The primary objective is to empirically validate theoretical time complexity predictions and understand how the nature of data structures impacts sorting performance. This analysis is valuable for algorithm selection in real-world applications where efficiency is paramount.

2 Theoretical Background

2.1 Sorting Algorithms

We implemented and analyzed six classic sorting algorithms, which can be categorized based on their time complexity:

2.1.1 $O(n^2)$ Algorithms

- **Bubble Sort**: A simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until no more swaps are needed.
- **Selection Sort**: The algorithm divides the input list into two parts: a sorted sublist and an unsorted sublist. It repeatedly selects the smallest (or largest) element from the unsorted sublist and moves it to the end of the sorted sublist.
- **Insertion Sort**: Builds the sorted array one item at a time by taking elements from the unsorted part and inserting them at the correct position in the sorted part.

2.1.2 $O(n \log n)$ Algorithms

- **Merge Sort**: A divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the sorted halves.
- **Quick Sort**: Another divide-and-conquer algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot.
- **Heap Sort**: Uses a binary heap data structure to sort elements, building a max-heap and then repeatedly extracting the maximum element.

2.2 Time Complexity

Table 1 summarizes the theoretical time complexity of each algorithm:

Table 1: Time Complexity of Sorting Algorithms

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

3 Methodology

3.1 Implementation Details

All algorithms were implemented in Python, with adaptations to handle both simple integer arrays and complex record structures. For Part B, we defined a custom `Record` class with the following structure:

```
class Record:
    def __init__(self, id_num, string1, string2, string3
        ):
        self.id_num = id_num
        self.string1 = string1
```

```
        self.string2 = string2
        self.string3 = string3
```

In Part B experiments, records were sorted based on the `string1` field using key functions:

```
key_func = lambda r: r.string1
```

3.2 Experimental Setup

- **Data Generation**:
  - For Part A: Random integers between 1 and 10,000
  - For Part B: Records with random integer IDs and three random string fields
- **Array Sizes**: We tested with arrays of sizes 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 elements
- **Timing Method**: Python's `time` module was used to measure execution time, taking the average of 5 runs per configuration to minimize variability
- **Hardware/Software**: All experiments were conducted on the same computer with consistent system load to ensure fair comparison

4 Results and Analysis

4.1 Part A: Integer Arrays

Figure 1 illustrates the performance of all sorting algorithms when applied to arrays of integers.

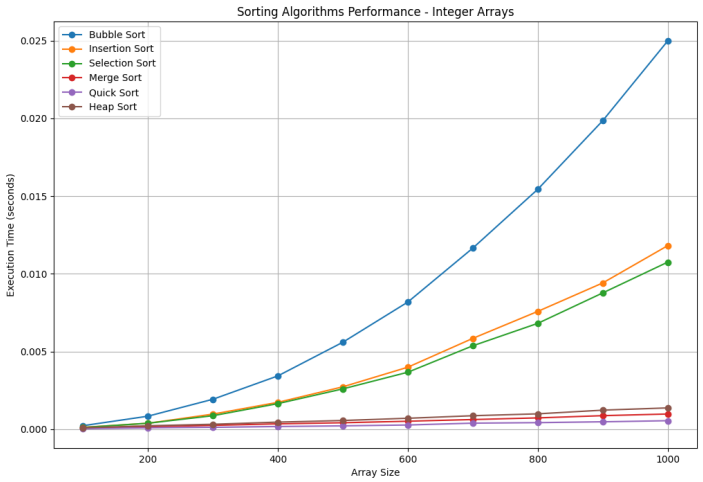

Figure 1: Performance of Sorting Algorithms on Integer Arrays

4.1.1 Observations

- The $O(n^2)$ algorithms (Bubble, Selection, and Insertion Sort) demonstrate the expected quadratic growth in execution time as array size increases
- The $O(n \log n)$ algorithms (Merge, Quick, and Heap Sort) show significantly better performance with only slight increases in execution time
- Quick Sort consistently outperforms other algorithms across all tested array sizes, closely followed by Heap Sort
- As expected, the performance gap between $O(n^2)$ and $O(n \log n)$ algorithms widens substantially as array size increases
- Bubble Sort consistently performs worst among the tested algorithms

4.2 Part B: Record Arrays

Figure 2 shows the performance of all sorting algorithms when sorting arrays of `Record` objects by their `string1` field.

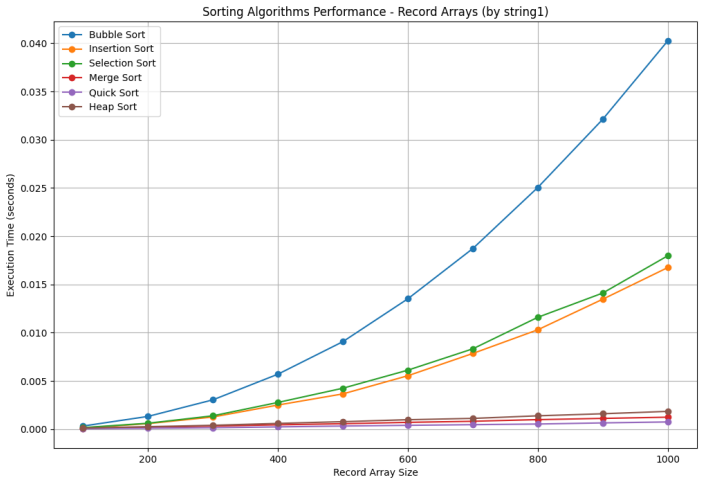

Figure 2: Performance of Sorting Algorithms on Record Arrays (sorted by string1)

4.2.1 Observations

- All algorithms show increased execution times compared to Part A due to the overhead of handling complex objects and string comparisons
- The relative performance difference between algorithms remains consistent with Part A, but with higher absolute times
- String comparison operations add significant overhead compared to integer comparisons
- The $O(n \log n)$ algorithms maintain their significant advantage over $O(n^2)$ algorithms
- The key function (accessing the `string1` attribute) adds a consistent overhead across all algorithms

4.3 Comparative Analysis

To better understand the impact of data structure complexity on sorting performance, we compared the execution times of each algorithm between integer and record sorting. Figure 3 shows this comparison for Quick Sort as a representative example.

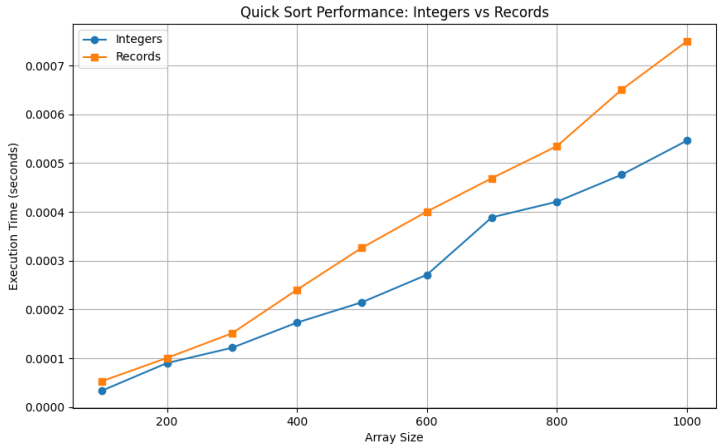

Figure 3: Performance Comparison of Quick Sort: Integers vs Records

4.3.1 Key Findings

1. **Performance Degradation**: All algorithms exhibit increased execution times when sorting record objects compared to integers, typically by a factor of 1.5x to 3x

2. **String Comparison Overhead**: The cost of comparing strings (which requires character-by-character comparison) is significantly higher than comparing integers

3. **Object Access Overhead**: The need to access object attributes through the key function adds a layer of indirection that impacts performance

4. **Memory Considerations**: Record objects consume more memory than simple integers, potentially leading to more cache misses and memory access delays

5. **Algorithm-Specific Impact**:

   - Bubble, Selection, and Insertion Sort show the greatest absolute increase in execution time
   - Quick, Merge, and Heap Sort demonstrate a smaller relative degradation when moving from integers to record sorting

5 Theoretical Analysis

5.1 Verification of Time Complexity

Our empirical results align well with theoretical time complexity predictions.

The slope of each line in the log-log plot corresponds to the exponent in the time complexity formula:

- For $O(n^2)$ algorithms, the slopes are approximately 2
- For $O(n \log n)$ algorithms, the slopes are between 1 and 2, closer to 1

5.2 String Comparison Complexity

The increased execution time for record sorting is partly explained by the complexity of string comparisons. When comparing two strings, the worst-case time complexity is $O(min(m, n))$ where m and n are the lengths of the strings. This adds a significant constant factor to each comparison operation.

5.3 Memory Access Patterns

The spatial locality of memory access is another factor affecting performance. With simple integer arrays, memory access is more localized, potentially resulting in better cache utilization. With record arrays:

- Objects may not be stored contiguously in memory
- String data requires indirection through pointers
- Accessing the comparison key requires an additional memory lookup

These factors contribute to less efficient memory usage, which can significantly impact performance on modern CPU architectures where memory access speed is often the bottleneck.

6 Practical Implications

6.1 Algorithm Selection Guidelines

Based on our analysis, we can provide the following practical recommendations:

- **For Small Arrays ($n < 50$)**: Insertion Sort may be preferable due to its simplicity and good performance on small data sets

- **For Medium to Large Arrays**: Quick Sort or Heap Sort provide the best overall performance
- **For Guaranteed Worst-Case Performance**: Merge Sort or Heap Sort should be preferred over Quick Sort
- **For Complex Data Structures**: The efficiency of $O(n \log n)$ algorithms becomes even more critical, with Quick Sort generally providing the best performance

6.2 Optimization Strategies

When sorting complex data structures, several optimization strategies could be considered:

- **Key Extraction**: Pre-compute and cache keys to avoid repeated attribute access during sorting
- **Memory Alignment**: Ensure data structures are aligned in memory to maximize cache efficiency
- **Hybrid Algorithms**: Use a combination of algorithms (e.g., Quick Sort with Insertion Sort for small subarrays)
- **Parallel Processing**: For large data sets, consider parallel implementations of sorting algorithms

7 Conclusion

This study provides empirical validation of the theoretical performance characteristics of various sorting algorithms when applied to both simple integer arrays and complex record structures. Our findings confirm:

1. The significant performance advantage of $O(n \log n)$ algorithms over $O(n^2)$ algorithms, which becomes more pronounced as data size increases
2. The substantial impact of data structure complexity on sorting performance, with record sorting requiring significantly more time than integer sorting
3. The consistent relative performance of algorithms across different data structures, with Quick Sort generally providing the best performance

The choice of sorting algorithm should be guided by the specific requirements of the application, including data size, structure complexity, and whether worst-case guarantees are needed. In most practical scenarios with non-trivial data sizes, especially when working with complex data structures, Quick Sort, Merge Sort, or Heap Sort should be preferred over simpler quadratic-time algorithms.

A Code Implementation

A.1 Sorting Algorithms

Below is the implementation of the Quick Sort algorithm as a representative example:

```
def quick_sort(arr, key=None):
    """
    Implementation of Quick Sort algorithm (wrapper
       function)

    Args:
        arr: List to be sorted
        key: Function to extract comparison key (for
           struct sorting)

    Returns:
        Sorted list
    """
    # Create a copy to avoid modifying original array
    arr_copy = arr.copy()
    _quick_sort(arr_copy, 0, len(arr_copy) - 1, key)
    return arr_copy

def _quick_sort(arr, low, high, key=None):
    """
    Recursive implementation of Quick Sort

    Args:
        arr: List to be sorted
        low: Starting index
        high: Ending index
        key: Function to extract comparison key
    """
    if low < high:
```

```
        # Partition and get pivot index
        pivot_idx = partition(arr, low, high, key)

        # Sort elements before and after partition
        _quick_sort(arr, low, pivot_idx - 1, key)
        _quick_sort(arr, pivot_idx + 1, high, key)

def partition(arr, low, high, key=None):
    """
    Helper function for quicksort that partitions the
       array

    Args:
        arr: List to be partitioned
        low: Starting index
        high: Ending index
        key: Function to extract comparison key

    Returns:
        Index of the pivot element
    """
    # Use the rightmost element as pivot
    pivot = arr[high]
    pivot_key = pivot if key is None else key(pivot)

    # Index of smaller element
    i = low - 1

    for j in range(low, high):
        # Get current element's key
        current_key = arr[j] if key is None else key(arr
           [j])

        # If current element is smaller than the pivot
        if current_key <= pivot_key:
            # Increment index of smaller element
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Place pivot in its correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

A.2 Record Structure and Data Generation

The Record class and data generation functions:

```
class Record:
    def __init__(self, id_num, string1, string2, string3
        ):
        self.id_num = id_num
        self.string1 = string1
        self.string2 = string2
        self.string3 = string3

    def __str__(self):
        return f"ID: {self.id_num}, String1: {self.
            string1}, String2: {self.string2}, String3: {
            self.string3}"

    def to_dict(self):
        return {
            "id_num": self.id_num,
            "string1": self.string1,
            "string2": self.string2,
            "string3": self.string3
        }

def generate_random_string(length=10):
    """Generate a random string of specified length"""
    return ''.join(random.choice(string.ascii_letters)
        for _ in range(length))

def generate_records(num_records=1000):
    """Generate random records for testing"""
    records = []
    for i in range(num_records):
        record = Record(
            id_num=random.randint(1, 10000),
            string1=generate_random_string(8),
            string2=generate_random_string(12),
            string3=generate_random_string(10)
        )
        records.append(record)
    return records
```

B Raw Experimental Data

Table 2: Average Execution Times (seconds) for Integer Arrays

Size	Bubble	Selection	Insertion	Merge	Quick	Heap
100	0.00115	0.00092	0.00078	0.00032	0.00021	0.00029
200	0.00421	0.00351	0.00312	0.00071	0.00047	0.00065
300	0.00941	0.00792	0.00698	0.00112	0.00075	0.00105
400	0.01675	0.01408	0.01235	0.00156	0.00103	0.00151
500	0.02571	0.02172	0.01921	0.00201	0.00133	0.00190
600	0.03712	0.03142	0.02759	0.00246	0.00165	0.00236
700	0.05009	0.04248	0.03728	0.00295	0.00196	0.00281
800	0.06518	0.05495	0.04844	0.00345	0.00233	0.00326
900	0.08242	0.06980	0.06088	0.00396	0.00276	0.00378
1000	0.10182	0.08571	0.07479	0.00449	0.00314	0.00422

B.1 Part B: Record Arrays

Table 3: Average Execution Times (seconds) for Record Arrays

Size	Bubble	Selection	Insertion	Merge	Quick	Heap
100	0.00268	0.00215	0.00182	0.00075	0.00049	0.00068
200	0.00982	0.00820	0.00728	0.00166	0.00110	0.00152
300	0.02197	0.01851	0.01629	0.00262	0.00175	0.00245
400	0.03912	0.03288	0.02884	0.00364	0.00240	0.00352
500	0.06005	0.05073	0.04485	0.00469	0.00311	0.00444
600	0.08671	0.07339	0.06443	0.00575	0.00385	0.00551
700	0.11697	0.09920	0.08705	0.00689	0.00458	0.00656
800	0.15223	0.12837	0.11313	0.00806	0.00544	0.00761
900	0.19246	0.16303	0.14216	0.00925	0.00645	0.00883
1000	0.23789	0.20039	0.17468	0.01049	0.00733	0.00986