Computational Thinking – Sorting Algorithms Project

Student: G00411344
Joseph Brickley

Introduction

Sorting is a core function in computer science, crucial for organising data in a specific sequence to enhance the efficiency of other algorithms that depend on sorted data to function optimally. Its importance stretches across various applications including software development, database management, scientific computing, and network routing.
John Healy emphasises the essence of efficient programming in the Data Structures & Algorithms Module:

"An efficient program will minimise both the space and time complexity required to complete a specific task."

This principle is particularly relevant when analysing the performance of sorting algorithms.

Core Concepts in Sorting

Complexity (Time and Space): Informs about the algorithm's scalability and resource usage, essential for choosing the right algorithm based on application needs.
Performance: Indicates how effectively a sorting algorithm processes data within the constraints of time and space.
In-place Sorting: Highlights algorithms that do not require additional space beyond a constant amount, highlighting space efficiency.
Stable Sorting: Maintains the relative order of records with equivalent keys after sorting, important for applications where the original data order matters.
Comparator Functions: Enable sorting based on custom-defined rules, adding flexibility to data handling.
Comparison-based and Non-comparison Based Sorts: Identifies sorting methods based on direct comparisons or through distribution-based approaches, suitable for diverse types of data and requirements.

Source: Computational Thinking & Algorithms Lecture Week 5: Analysing Algorithms, Dominic Carr, 2024

Relevance and Applications

Sorting algorithms are pivotal in numerous fields:
Database Systems: Facilitate faster data retrieval, effective indexing, and quicker searches.
Computer Graphics: Employed in tasks like render ordering and collision detection.
Scientific Computing: Essential for organising large datasets, which is critical for simulations and complex calculations.

Choosing the right sorting algorithm based on specific performance, memory usage, and stability requirements can significantly enhance the efficiency and scalability of software systems. This understanding is crucial for developers and researchers alike, ensuring optimal application performance across various computing disciplines.

Analysis of Sorting Algorithms

The performance of sorting algorithms is typically assessed based on several critical factors.

Time Complexity: This refers to the duration the algorithm takes to sort the data, which varies depending on the algorithmic design and the operations it performs, such as the creation of data structures during execution.

Space Complexity: This measures the amount of RAM an algorithm needs throughout its lifecycle, heavily influenced by the types of data structures employed.

Big O notation: Is used to describe the upper bound of an algorithm's running time or space requirements in terms of the input size. It provides a high-level understanding of an algorithm's efficiency and scalability by classifying its worst-case scenario. It helps in comparing the performance of different algorithms by abstracting away constants and lower-order terms, focusing solely on the dominant factor as n increases. Understanding Big O notation is crucial for selecting the most appropriate algorithm for a given problem, especially when dealing with large datasets where differences in time complexity can lead to significant performance variations.

Source: Computational Thinking & Algorithms Lecture Week 7: Analysing Algorithms, Dominic Carr, 2024
Source: p57-p63, Data Structures: Abstraction and Design Using Java, Book by Elliot Koffman and Paul A. T. Wolfgang

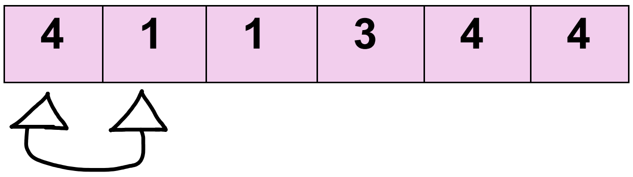
Benchmarking: Benchmarking is a practical approach to measuring the performance of algorithms. It involves executing a specific algorithm and timing how long it takes to complete under different conditions, providing real data on how an algorithm performs on actual systems. This process can be influenced by several factors, including system architecture, CPU speed, available memory, and the operating system. Benchmarking delivers empirical data essential for understanding the impact of code modifications on software performance.

In this project, the benchmark application focuses on in-place sorting algorithms, which sort numbers without requiring additional storage, thus only using a small, constant amount of extra space. The stability of these algorithms ensures that the relative order of equal sort items is maintained post-sorting. While algorithms like Bubble Sort are stable, others like Selection Sort and typical implementations of Quick Sort are not, although stable versions of Quick Sort do exist. Comparator functions also play a role, allowing sorting to be customised based on user-defined criteria.

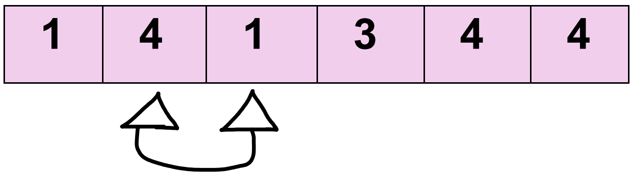
<u>Bubble Sort</u>

Steps through the list of elements and compares adjacent elements. It then swaps them to their correct order. It is simple but inefficient for handling large datasets.

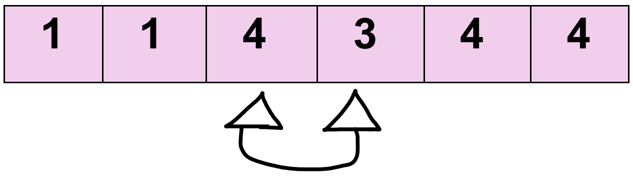
Step 1: Compare and swap adjacent elements



Step 2: Continue to compare and swap



Step 3: Continue to compare and swap



Step 4: No swap is needed. The array is now sorted.

1	1	3	4	4	4

Bubble Sort, Programiz.com website; source: bubble-sort, Programiz

Insertion Sort

Presumes the first element is sorted; e.g. 4 is sorted when compared with itself. It builds the sorted array one item at a time, then inserting each new element into its correct position. It performs well on small or partially sorted datasets. It has a time complexity of $O(n2)$.

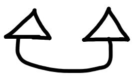
Step 1: Start from the second element
Consider 1 for insertion
Move back through the list

Before:

4	1	1	3	4	4

Insert:

1	4	1	3	4	4



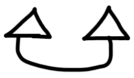
Step 2: Move to the next element (another 1)
Consider 1 for insertion
Move back through the list

Before:

1	4	1	3	4	4

Insert:

1	1	4	3	4	4



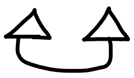
Step 3: Move to the next element (3)
Consider 3 for insertion
Move back through the list

Before:

4	1	1	3	4	4

Insert:

1	1	3	4	4	4



Step 4: Consider 4 insertion
Already in correct position as no smaller element is behind

(no swap)

Step 5: Move on to next element (another 4)
Consider 4 for insertion
Already in correct position as no smaller element behind
(no swap)

Meaning it is sorted:

1	**1**	**3**	**4**	**4**	**4**

Insertion Sort, Programiz.com website: source: [insertion-sort, programiz.com]

<u>Selection Sort</u>

Segments the list into sorted and unsorted parts, repeatedly selecting the smallest element from the unsorted segment and adding it to the sorted segment. Known for its simplicity, it has a time complexity of $O(n2)$ making it less efficient for large lists.

Step 1: Find the smallest element in the array
Scan:

4	**1**	**1**	**3**	**4**	**4**

Smallest: 1 at index 1
Swap 1 with 4 (index 0)
Result:

1	**4**	**1**	**3**	**4**	**4**

Step 2: Find the smallest from index 1 onward
Scan:

4	**1**	**3**	**4**	**4**

Smallest: 1 at index 2
Swap 1 with 4 (index 1)
Result:

1	**1**	**4**	**3**	**4**	**4**

Step 3: Find the smallest index from 2 onward
Scan:

4	**3**	**4**	**4**

Smallest: 3 at index 3
Swap 3 with 4 (index 2)
Result:

1	**1**	**3**	**4**	**4**	**4**

Step 4: Find the smallest from index 3 onward

Scan:

4	**4**	**4**

No swaps needed as 4 is the smallest and already in place
Result is sorted:

1	**1**	**3**	**4**	**4**	**4**

Selection Sort, Programiz website: source: selection-sort, programiz.com

Quick Sort

Utilises a 'pivot' element to partition the array into elements less than and greater than the pivot, sorting each partition recursively. It is highly efficient but typically unstable, though stable versions are available.

Before:

4	1	1	3	4	4

Choose pivot: 4 (last element)
Elements less than or equal to 4 are placed before the pivot
In this case all elements go to the left

1	1	3	4	4	4

Recursive sort left:

1	1	3

Choose pivot 3 (last element)

Partition:

1	1	3

Elements less than or equal to 3 are placed before the pivot

Recursive sort left:

1	1

No further sorting needed (already less than or equal to 3)

Recursive sort right:

3

Single element, no need to sort further

Recursive sort right:

4	4

Already sorted, all elements are equal

Concatenate; combine all sorted elements
Result is sorted:

1	1	3	4	4	4

Quick Sort, Programiz website: source: quick-sort, programiz.com

Bucket Sort

Distributes elements into several buckets, sorts these buckets individually, and then concatenates them back together. This sort is effective when the input is uniformly distributed across the range.

Before:

4	1	1	3	4	4

Step 1: Create buckets

Bucket 1: will hold the value of ‘1’

Bucket 3: will hold the value of ‘3’

Bucket 4: will hold the value of ‘4’

Step 2: Distribute elements into Buckets

1	1

3

4	4	4

Step 3: Concatenate the buckets

1	1	3	4	4	4

Result is sorted:

1	1	3	4	4	4

Bucket Sort, Programiz website: source:bucket-sort, programiz.com

Implementation Details

The benchmarking was conducted using a Java program which measured the average execution time of each sorting algorithm across different array sizes, from 100 to 9000 elements, over 10 trials each as per the project requirements. The implementation involved generating a random array, cloning it for repeatability, and timing how long each algorithm took to sort the array. When writing the code for this project I had not initially included the cloning, which resulted in overly distorted data in the initial stages. I was later able to strip and repurpose the code from the week 10 example.

Source: Computational Thinking & Algorithms, week 10 code stub

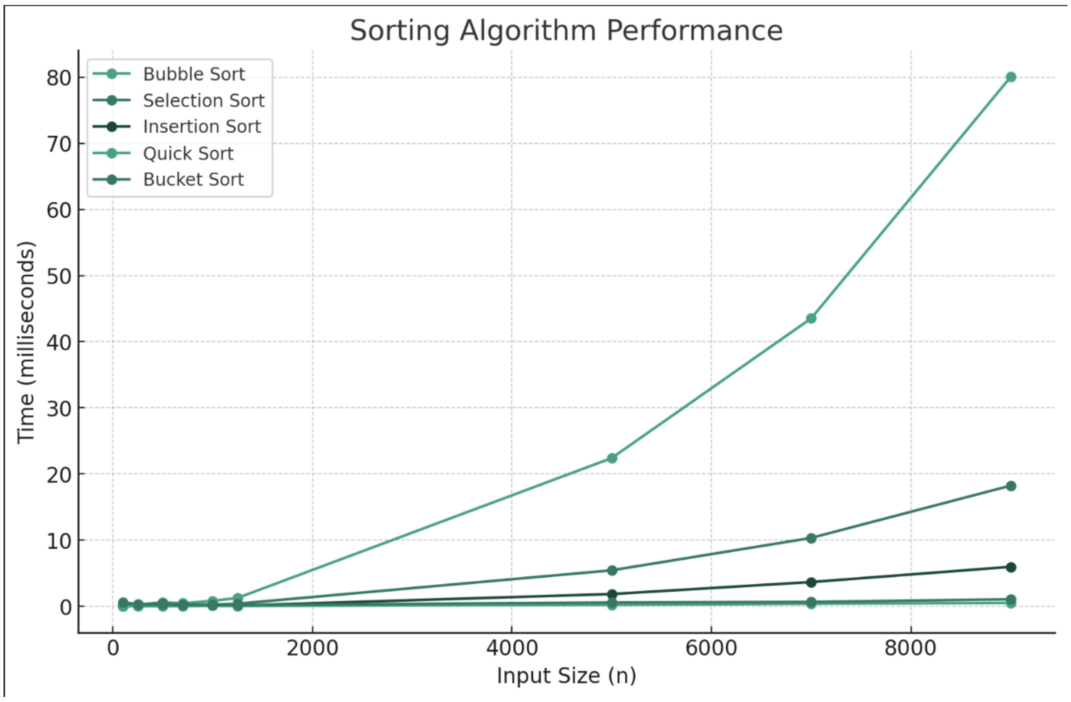
Performance Results

The performance was measured and displayed in a table format, which is detailed below, along with a graph that visually represents the data:

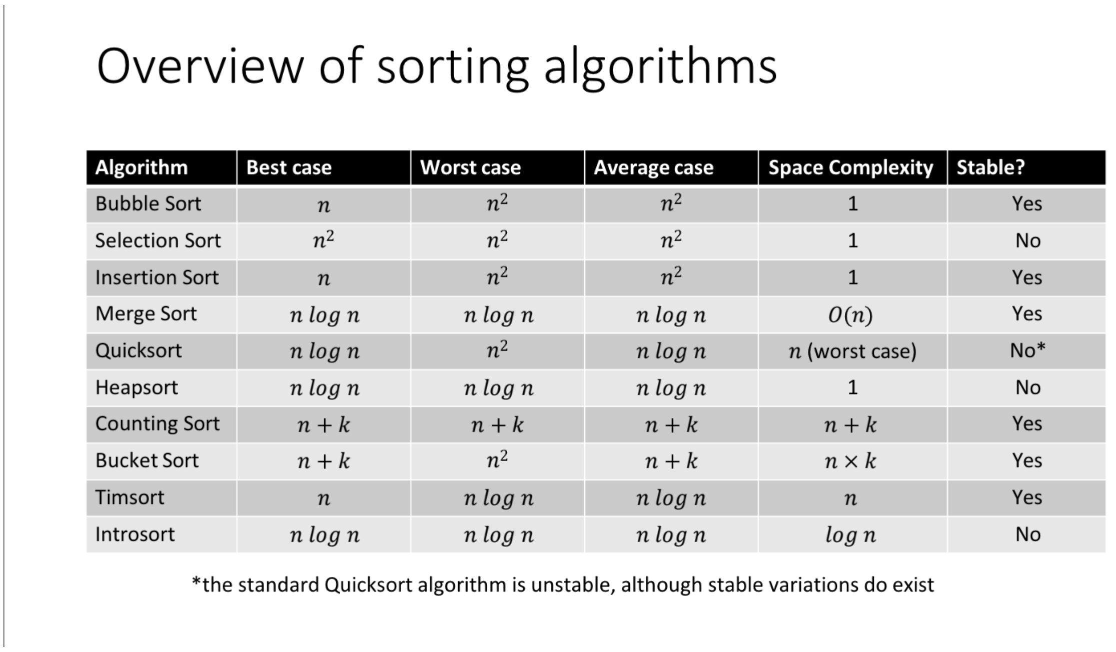
```
Size                 100        250        500        700       1000       1250       5000       7000       9000
Bubble Sort     0.173 ms   0.305 ms   0.624 ms   0.483 ms   0.845 ms   1.287 ms  22.421 ms  43.534 ms  80.034 ms
Selection Sort  0.078 ms   0.188 ms   0.472 ms   0.194 ms   0.278 ms   0.414 ms   5.463 ms  10.357 ms  18.264 ms
Insertion Sort  0.048 ms   0.150 ms   0.316 ms   0.252 ms   0.249 ms   0.138 ms   1.873 ms   3.690 ms   5.992 ms
Quick Sort      0.029 ms   0.039 ms   0.032 ms   0.047 ms   0.096 ms   0.077 ms   0.260 ms   0.409 ms   0.534 ms
Bucket Sort     0.626 ms   0.345 ms   0.286 ms   0.250 ms   0.242 ms   0.215 ms   0.599 ms   0.688 ms   1.083 ms
```

Table Results - all values are in milliseconds and are the average of 10 repeated runs.

Graphical Representation



Graph created using Visme.co



Overview of sorting algorithms

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n\, log\, n$	$n\, log\, n$	$n\, log\, n$	$O(n)$	Yes
Quicksort	$n\, log\, n$	n^2	$n\, log\, n$	n (worst case)	No*
Heapsort	$n\, log\, n$	$n\, log\, n$	$n\, log\, n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n\, log\, n$	$n\, log\, n$	n	Yes
Introsort	$n\, log\, n$	$n\, log\, n$	$n\, log\, n$	$log\, n$	No

*the standard Quicksort algorithm is unstable, although stable variations do exist

Overview of Sorting Algorithms taken from Week 11 Slides: More Sorting Algorithms
Source: [Week 11 : Sorting Algorithms PDF]

The results illustrate that while Bubble, Selection, and Insertion Sorts show expected performance deterioration as the array size increases, Quick Sort consistently shows superior performance due to its $O\ (n \log n)$ complexity. Bucket Sort performs well, especially for larger sizes, suggesting good distribution of data across buckets.

As expected, Bubble Sort shows the poorest performance due to its O(n^2) time complexity. This algorithm's inefficiency is especially evident as the array size increases, making it unsuitable for large datasets.
Insertion Sort performs relatively better than Bubble Sort for small datasets, but its O(n^2) complexity still leads to significant performance degradation with larger datasets. It remains practical for small or nearly sorted datasets due to its simplicity and efficiency in such cases.
Like Bubble and Insertion Sorts, Selection Sort also exhibits O(n^2) time complexity. Its performance is consistently poor for larger datasets.
Quick Sort demonstrates superior performance due to its average-case O(n log n) time complexity. The benchmarking results highlight its efficiency, especially with larger datasets.
Bucket Sort shows impressive performance, particularly with larger datasets, indicating effective data distribution across buckets. This algorithm is highly efficient for datasets that are uniformly distributed, making it a strong choice when these conditions are met.

This benchmarking gives critical insights into the scalability of sorting algorithms and helps in selecting the right algorithm based on the size of the data and required efficiency. When considering the benchmark results and the chart; Overview of Sorting Algorithms *, we can see the results align well with the theory and the application results highlight the practical applicability of these sorting techniques. For my own experience in learning to program, gaining an understanding of algorithms has been a momentous step towards comprehending the scale of how real-world systems work successfully.