Algorithms-DSA-Report

Data structures and Algorithms

Based by following course: PG4200 Datastructures and Algorithms

Author: Kamal Al Masoudi

Submitted 25th of April 2025

Table of Contents

1. Introduction

1.1 What Is Data and Why Organize It?

Data, Jonathan Furner argues, has always been understood as “things given”. Premises taken to be true so that other, less certain conclusions may be reached (Furner, 2016, p. 290). In classical geometry the givens were lines or angles. In modern science they are sensor readings, survey responses, log files, and the like. Yet Furner insists that such observations have little value until they are shaped into what he calls datasets. Datasets are purposeful aggregations that include not just the recorded facts but also the contextual information. It includes who collected them, when, by what method and what future analysts will need in order to judge their reliability (Furner, 2016, p. 302).

The same distinction appears in applied tutorials on data science. W3Schools begins its primer by defining data simply as “a collection of information,” but immediately divides that collection into unstructured and structured forms (W3Schools, n.d.-a). Unstructured data are portrayed as amorphous and resistant to analysis, whereas structured data is presented as orderly and readily interpretable.

To illustrate the transition from the first state to the second, the site offers the humble Python array: [80, 85, 90, 95, 100, 105, 110, 115, 120, 125].
This is the smallest possible example of imposing sequence, boundaries, and type consistency on an otherwise formless list of numbers. Even this miniature structure, the tutorial suggests, embodies the essence of data science. It turns scattered facts into something a computer can scan, slice, and compare when working with lots of data.

Viewed together, Furner’s theoretical emphasis on provenance and W3Schools focus on computational order reveal two sides of the same principle. Whether stored in an academic repository or in a beginner’s array variable, a dataset is valuable because it is both intelligible to people and tractable for machines.

The intellectual labour of specifying units, recording collection conditions, and documenting the purpose of the data guards against misinterpretation. While the technical labour of choosing arrays, tables, or databases enables efficient retrieval and processing. Thus, to speak of “organizing data” is to recognise a dual imperative, safeguarding meaning and enabling computation. Only when both aims are satisfied do “things given” become true resources for discovery and decision-making.

1.2 Data in the Context of Algorithms and Data Structures

To define the concepts of algorithms and data structures, we follow Robert Lafore's explanation: “Data structures are ways in which data is arranged in your computer’s memory (or stored on disk)” (Lafore, 2002, p. 30). In this context, data refers to the values or information that are stored and organized for efficient use. Lafore further defines algorithms as “The procedures a software program uses to manipulate the data in these structures” (Lafore, 2002, p. 30). The synergy between algorithms and data structures is crucial, as an algorithm’s performance is typically tied to how easily and efficiently it can retrieve and manipulate data.

1.3 The Importance of Data and Organization

Modern applications collect and utilize huge amounts of information, from user analytics in social media platforms to satellite data in climate modeling. Consider large-scale websites like Google.com. An article covering how much data is generated each day mentions that Google handles over 20 petabytes of data daily (Loy, 2024). One of the functions of Google’s algorithm is to rank websites based on user's searches (Webfx, n.d). If the data isn’t structured to support quick lookups and updates, the user experience is compromised and the recommended websites would either be generated slowly or be irrelevant to the user.

Good data organization also enables scalability. While a system that handles a few thousand entries might survive with a basic, unoptimized storage approach, scaling to millions or billions of entries requires a more rigorous approach. According to Matt Loy’s prediction for the upcoming years, the amount of volume will only increase (Loy, 2024). A once sufficient structure can turn into a critical bottleneck, triggering massive reengineering efforts or severely limited performance.

Using Google as an example, if their algorithms fail to scale effectively, they risk significant performance issues as data volumes expand. Consequently, addressing data organization from the outset isn't just a best practice for small scale prototypes but a necessity for real world systems that must grow over time. By storing data in well structured formats, developers build platforms that can grow and be modified with minimal disruptions.This flexibility helps extend the software's lifetime.

2. Data Structures

2.1 Why Use Data Structures?

The connection between data and data structures can be visualized as a supply chain. Data serves as the "raw materials" fueling the entire operation, while data structures provide the "warehouse" that stores these materials in an orderly manner. Algorithms, in this analogy, are the systems that retrieve and process the materials in ways that add value. The basic operations of data structures are such as searching, sorting, inserting, updating or deleting (Dahl, n.d.-a, p. 9). Without a well-managed warehouse, even the most capable systems waste precious time searching through cluttered shelves. Choosing the right data structure is therefore just as important as crafting an efficient algorithm. Failing to do so can undermine performance to such an extent that a normally good algorithm performs poorly due to suboptimal data storage.

2.2 Linear Data Structures

Arrays

The array is the most fundamental linear data structure. It consists of contiguous memory locations, each storing an element of the same type. (Dahl, n.d.-b, p. 11). The key advantage of arrays is constant-time random access. Accessing the i-th element is simply a matter of computing an address offset. This makes arrays ideal for scenarios requiring repeated index-based lookups or iterative processing.

However arrays can be restrictive. They must often be declared with a fixed size, making it difficult to add new elements once capacity is reached. Removing or inserting elements in the middle requires shifting, which can degrade performance for large arrays. Consequently, while arrays are an excellent default choice when the size and usage patterns of data are well understood, they can become inefficient if frequent insertions or deletions are necessary.

Linked Lists

A linked list is an alternative for storing linear sequences. Instead of placing elements in consecutive memory cells, it arranges them in nodes, where each node holds data and a pointer (or reference) to the next node (Dahl, n.d.-b, p. 15). This implementation is called a singular linked list. However, there are other ways to implement linked lists.

In some cases, a node might also contain a pointer to the previous element, forming a doubly linked list that facilitates backward traversal (Dahl, n.d.-b, p. 15). A doubly linked list is ideal for features like a music player playlist. Each song node holds next and prev pointers, so the app can move instantly to the previous or next track and delete or insert the current track in O(1) time. Without the backward pointer, a singly linked list would require an O(n) scan to find the predecessor before any “skip back” or deletion, slowing the user experience.

Further we have circular singly linked lists which also have nodes with pointers for the next element, but instead of having a null pointer at the end of the list, the last node has a pointer to the first node (head) (Geeksforgeeks, 2025). A circular singly linked list fits well in a multiplayer board game that cycles turns endlessly. Each player is a node whose next pointer leads to the next player, and the last player’s pointer loops back to the first. Advancing to the next turn is always a single pointer hop (O(1)), and the code never has to test for “end of list” before restarting the order. This keeps turn management simple and efficient even as players join or leave mid game.

The last list is the circular doubly linked list combining the features of both a doubly linked list and a circular list. Each node stores references to both its next and previous nodes, and the list's tail is linked back to the head while the head's previous pointer links to the tail (Geeksforgeeks, 2025). A circular doubly linked list is ideal for implementing an image carousel in a GUI. Users can swipe forward or backward through photos, and when they reach the last image, the next swipe instantly loops to the first (and vice versa) without any edge checks. Because each node holds both next and prev links and the tail connects back to the head, moving in either direction or inserting/removing images is constant time, giving a continuous browsing experience.

Linked lists enable dynamic allocation. Expanding or shrinking the list is straightforward, as nodes can be inserted or removed efficiently without shifting other data elements. They are used in applications requiring frequent insertion or deletion at known positions, especially in the middle of the sequence. However, linked lists have slower random access times compared to arrays. Accessing the i-th element requires traversing from the beginning of the list, resulting in linear (O(n)) time complexity for retrieval operations.

Stacks

A stack operates on a Last In, FirstOut (LIFO) principle, meaning the most recently added item is the next one to be removed (Dahl, n.d.-b, p. 23). Stack operations include "push" which places an item on top, and "pop" which removes the top item.

A simple way to understand stacks is by visualizing a pile of dishes. The most recently placed dish is always removed first, making it a Last In, First Out (LIFO) structure. Similarly, when a user performs actions such as typing or deleting, these actions are pushed onto a stack. Undoing an action pops the most recent activity from the stack, reversing it. This ensures actions are undone precisely in the reverse order they occurred.

Queues

A queue follows the FIFO principle (Dahl, n.d.-b, p. 25), resembling a real-life line at a store. In a linear queue, the first element inserted into the que (enqueue) is also the first element which will be removed (dequeue). A linear queue is useful in scenarios like customer support ticket systems, where fairness and order matter. Customers who submit tickets first are addressed first, following the FIFO principle. This ensures an orderly processing sequence, prevents newer requests from unfairly jumping ahead, and guarantees predictable wait times.

A circular queue, or ring buffer, is a fixed-size FIFO structure where the front and rear pointers wrap around the array. When a pointer reaches the end, it loops back to index 0, letting the buffer reuse slots without shifting data. This closed loop enables constant-time inserts and removals until the queue is full. (Dahl, n.d.-b, p. 27). A circular queue is ideal for buffering keyboard inputs in a computer system. Key presses are placed into a fixed size ring buffer as they occur, ensuring efficient storage. The circular nature allows older entries to be processed and cleared quickly, enabling continuous, real-time responsiveness without reallocating memory or shifting data.

In a priority queue, each item is assigned a priority, and when dequeued, the item with the highest priority is returned. If multiple items share the same priority, the one inserted earliest is removed first, preserving FIFO order among equals. There are two common approaches to enqueuing: in an ordered implementation, items are inserted at their correct position based on priority; in an unordered implementation, items are simply appended, and sorting is deferred until removal (Dahl, n.d.-b, p. 28). A priority queue is useful in hospital emergency rooms, where patients are treated based on the severity of their condition rather than arrival time. Critical cases like heart attacks are given higher priority and treated first, while less urgent cases wait their turn. This ensures that medical attention is allocated where it’s needed most.

A deque (double-ended queue) allows insertion and deletion from both the front and the rear, offering more flexibility than a standard FIFO queue. It is typically implemented using a doubly linked list, which enables constant time operations at

both ends, or a dynamic array with circular indexing to achieve amortized O(1) performance (Dahl, n.d.-b, p. 29). A deque is useful in managing a playlist in a music app where users can add songs to either the front or the back of the queue. For example, a user might queue a new favorite song to play next (front), while letting the rest of the playlist continue in order (rear). The ability to insert and remove songs from both ends gives users flexible control over how their music plays.

2.3 Non-Linear Data Structures

Trees

A tree is a powerful data structure that organizes elements in a hierarchical format, allowing each node to connect to multiple child nodes (Dahl, n.d.-c, p. 27). Unlike linear structures such as arrays or linked lists, trees can branch out in various directions, making them ideal for representing data with parent-child relationships. The top node is called the root, while nodes with no children are referred to as leaf nodes. Each node, except the root, has exactly one parent, and connections between nodes are known as edges. Trees are widely used in applications such as file systems, organizational models, databases, and network routing due to their efficiency in searching, sorting, and structuring data (W3Schools, n.d.-b).

Common types include binary trees, where each node has up to two children, and binary search trees (BSTs), which maintain sorted order for quick lookup. AVL trees are a special kind of BST that keeps the structure balanced for optimal performance. These variations make trees an essential tool in both theoretical computer science and practical software development (W3Schools, n.d.-b).

Graphs

A graph is a versatile data structure used to represent relationships between objects. Unlike linear structures such as arrays or linked lists, graphs are non-linear, meaning elements can be connected in complex ways. A graph consists of vertices (also called nodes) and edges, where each edge represents a connection between two vertices (Dahl, n.d.-c, p 34). This structure is especially useful in scenarios where

entities are interconnected in multiple directions, such as in social networks, transportation maps, and the internet (W3Schools, n.d.-c).

Graphs can be directed or undirected. In a directed graph, edges have a specific direction, indicating a one way relationship, whereas undirected graphs allow bidirectional connections. Edges can also have weights, representing quantities like distance, cost, or time, which is useful for algorithms that calculate shortest paths or optimize routes. Other properties include cyclicity, where cycles or loops can form within the structure, and connectivity, which determines whether all nodes are reachable from one another.

Graphs are typically represented using either an adjacency matrix or an adjacency list. An adjacency matrix is a two dimensional array where each cell indicates the presence or weight of an edge between two vertices (Dahl, n.d.-c, p 36). It is straightforward but memory-intensive, especially for sparse graphs. An adjacency list, on the other hand, stores only the existing connections for each vertex, making it more space-efficient for graphs with fewer edges.

Because of their flexibility, graphs are foundational in computer science and are applied in fields ranging from network routing and search engines to biological modeling and recommendation systems (W3Schools, n.d.-c).

3. Algorithms

3.1 What Is an Algorithm?

In computational terms, we could further define an algorithm as a definitive set of instructions for accomplishing a task within a finite number of steps (W3Schools, n.d.-d). An algorithm should be consistent which means it should always produce the same outcome given the same input. Researchers and professionals design algorithms to solve an array of problems. Some appear as trivial sorting or searching tasks. Others underpin large scale machine learning pipelines, cryptographic protocols, or complex optimization workflows.

A well structured algorithm is like a meticulously detailed recipe. Each step is clear, each resource is accounted for, and each action moves the process toward the desired conclusion. However, in practice, algorithms must not only be correct but also efficient. A mathematically valid solution is of limited practical use if it consumes excessive time or memory. This interplay between correctness and performance is central to algorithm design and analysis.

3.2 Algorithm Efficiency

Time and Space Complexity

The notion of **algorithmic complexity** provides a framework for describing how an algorithm's required resources scale with the input size. Two primary measures are time and space.

Time Complexity, describes how the execution time (or number of fundamental operations) grows with the input (Dahl, n.d.-d, p. 6). Common notations include:

- **O(1)**: Constant time, independent of input size.
- **O(n)**: Linear time grows in direct proportion to the input size.
- **O(log n)**: Logarithmic time, typical in divide-and-conquer or binary search contexts.
- **O(n²)**: Quadratic time, which often emerges from nested loops.
- **O(n log n)**: A common hallmark of efficient sorting or divide-and-conquer algorithms.

Space Complexity, indicates how much memory an algorithm requires beyond the input and output (Dahl, n.d.-d, p 16).
To calculate space complexity we use asymptotic notations:

- **Omega notation Ω:** Represents the best case
- **Theta notation Θ**: Represents the average case
- **Big-O notation O**: Represents the worst case

Why Complexity Matters

Computational resources are not infinite. If an algorithm's time complexity is too high, it may become prohibitively expensive to run as input sizes grow, consuming excessive CPU resources or memory. With the rise of big data applications, high complexity can mean the difference between finishing computations in an hour versus never finishing at all. Furthermore, understanding an algorithm's complexity aids in predicting its behavior and choosing optimizations. A theoretically optimal algorithm that uses advanced data structures might be necessary to keep a large enterprise application running efficiently under heavy loads.

3.3 Sorting Algorithms in Focus

Sorting is one of the most frequently encountered problems in computer science. Whether the goal is to arrange search results, ensure a list is in ascending numerical order, or group data by specific characteristics, sorting arises across a broad spectrum of fields. Sorting also has a close synergy with many other algorithms, since a sorted input can drastically reduce the complexity of subsequent operations such as searching or merging datasets. Four different algorithms illustrate different approaches to the sorting problem:

Bubble Sort: Repeatedly swap adjacent elements if they are out of order (Gupta, n.d.-a, p. 3). Bubble sort has a best-case time complexity of O(n) when the list is already sorted, but the normal and the worst cases are $O(n^2)$ (W3schools, n. d.-e). The space complexity is O(1). It is commonly used for educational purposes of small datasets.

Insertion Sort: Build a sorted list one element at a time by "inserting" each new item into its correct place (Gupta, n.d.-a, p. 6). Its best-case time complexity is O(n), while both average and worst cases are $O(n^2)$ (W3Schools, n.d.-f). Space complexity is O(1), making it efficient for small, partially sorted lists or real-time sorting scenarios.

Merge Sort: A divide-and-conquer approach that splits the list in half, recursively sorts each half, then merges the two sorted halves into a complete, sorted result (Gupta, n.d.-a, p. 9). Merge Sort consistently has a time complexity of O(n log n) for

best, normal, and worst cases (W3Schools, n.d.-g). Its space complexity is O(n) due to additional temporary lists needed during merging. It is ideal for large datasets requiring stable and predictable sorting performance.

Quick Sort: Another divide-and-conquer technique where a pivot is chosen, and the list is partitioned around it. Quick Sort is frequently used for in-memory sorting (Gupta, n.d.-a, p. 13). It typically has O(n log n) time complexity for best and average cases, but its worst-case scenario is $O(n^2)$, often when poor pivots lead to unbalanced partitions (W3Schools, n.d.-h). Quicksort's space complexity is generally O(log n), making it suitable for efficient memory sorting.

3.4 Computability and Complexity Theory

Computability theory explores the fundamental limitations of algorithms, determining precisely which problems can and cannot be solved by computational means. Basic tasks like arithmetic operations or straightforward sorting algorithms are easily computable, whereas certain other problems, notably the Halting Problem. The Halting Problem, which questions if a program will terminate given specific inputs are proven to be undecidable. Such undecidable problems lack universal algorithmic solutions, highlighting the intrinsic limitations within computation (Gupta, n.d.-b, p. 2).

Complexity theory further analyzes problems based on the resources, such as time and memory, needed for their resolution. It classifies solvable problems into several clearly defined categories:

Class P consists of problems solvable in polynomial time, indicating that solutions can be reached efficiently as input sizes grow. Examples include fundamental computations such as arithmetic calculations and basic sorting operations (Gupta, n.d.-d, p. 7).

Class NP encompasses problems whose solutions are challenging to discover directly, yet are easily verifiable once presented. Notable examples include logic

puzzles like Sudoku and optimization challenges like the Traveling Salesman Problem (Gupta, n.d.-d, p. 7).

Co-NP includes problems for which it is relatively simple to verify the incorrectness of proposed solutions, although finding or verifying correct solutions can be computationally intensive. These problems typically involve exhaustive checking for accuracy across various conditions (Gupta, n.d.-d, p. 8). NP-hard problems represent challenges at least as difficult as the most demanding NP problems. An efficient solution to any NP-hard problem would inherently provide efficient solutions to every NP-classified problem, underscoring their significance and complexity within computational theory.

NP-complete problems are particularly crucial as they lie at the intersection of NP and NP-hard categories. Efficiently solving any one NP-complete problem would mean that all problems within NP could be solved efficiently. This pivotal role makes NP-complete problems central to ongoing research in computational complexity theory (Gupta, n.d.-d, p. 9). A comprehensive understanding of these complexity classes is essential as it aids computer scientists in devising strategies like heuristic algorithms and approximation techniques. Such approaches help tackle computationally demanding problems that resist straightforward algorithmic solutions, allowing practical handling of theoretically challenging scenarios (Gupta, n.d.-d, p. 10).

4. Methodology

To evaluate and compare sorting algorithms, we followed a consistent methodology across all implementations. First, we extracted a list of unique latitude values from the worldcities.csv dataset, removing duplicates and cleaning the data. In the code we have a check that sees the amount of unique latitudes. Afterwards we sort the data in ascending order. To verify that the list has been ordered correctly, each algorithm result includes the first five and last five latitudes. For each algorithm, we tested performance on both the original list and a randomized version where needed, created using Collections.shuffle(). This allowed us to assess how input order affects runtime and behavior.

Each sorting algorithm was implemented in Java, ensuring consistent input and output handling. Instead of having a separate main class for all algorithms, a main function is included inside each algorithm with the check of the order. The data structure used is List<Double> since the algorithms rely on accessing elements by index and swapping or comparing them. We measured performance using three key metrics. Execution time (via System.nanoTime() in seconds), the number of internal operations, such as swaps or comparisons and memory usage. Our analysis compares the experimental results to theoretical time and space complexity, noting where real-world behavior aligned or differed. Finally, we discussed trade-offs between algorithms in terms of speed, input sensitivity, and memory usage.

5. Problem 1 - Bubble Sort

1A Implementing optimized and non-optimized bubble sort

Conceptual explanation

Bubble Sort is one of the most fundamental and straightforward sorting algorithms. It works by repeatedly scanning through an array, comparing adjacent elements and swapping them if they are out of order. Over multiple passes, the largest elements "bubbles" are placed to the end, leaving the smallest elements at the beginning. Despite its simplicity, Bubble Sort is inefficient for large datasets because it runs in $O(n^2)$ time in average and worst-case scenarios.

Describing the code

The provided code underneath implements a non-optimized version of the Bubble Sort algorithm. Designed specifically to sort a list of Double values and count the number of element swaps performed during the process. This implementation is often used as an educational example due to its simplicity and clear demonstration of basic sorting principles. The method bubbleSortNonOptimized accepts List<Double> as input and returns an integer representing the total number of swaps made while sorting the list. Internally, the function first initializes a swaps counter to zero, which is used to record the number of times adjacent elements are exchanged. It also stores

the size of the list in a local variable n to avoid redundant sized computations in the nested loops.

```java
private static int bubbleSortNonOptimized(List<Double> list) {
    int swaps = 0;
    int n = list.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (list.get(j) > list.get(j + 1)) {
                Collections.swap(list, j, j + 1);
                swaps++;
            }
        }
    }
    return swaps;
}
```

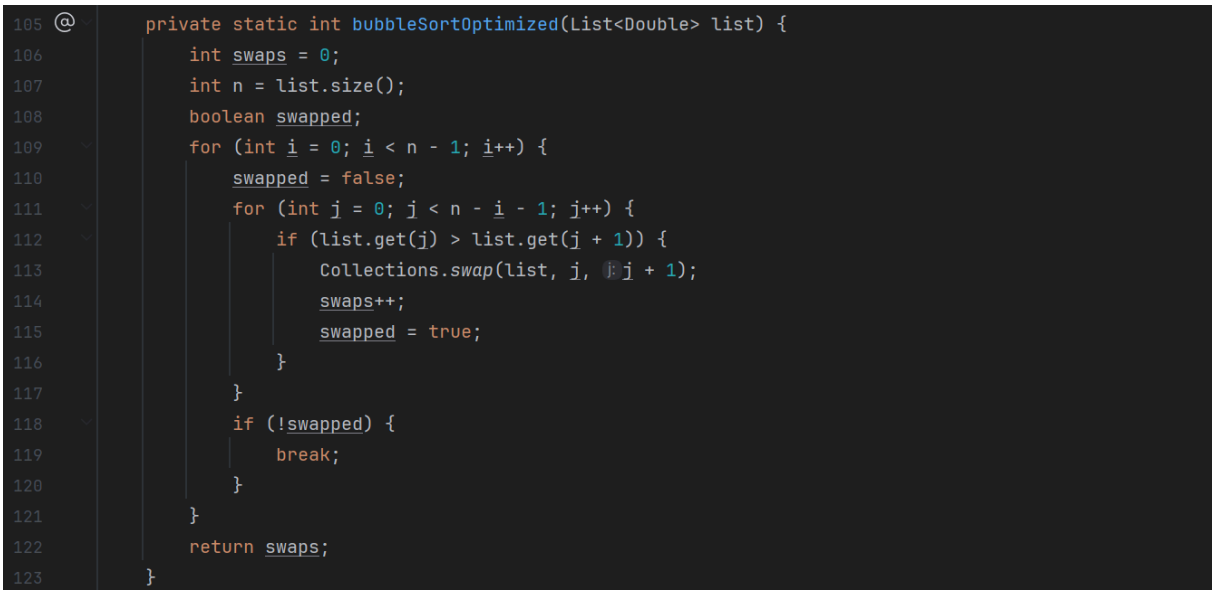
The core of the algorithm consists of nested loop structure. The outer loop iterates from the first element to the second-to-last. And for each iteration of the outer loop, the inner loop iterates through the unsorted portion of the list. During each inner loop iteration, the algorithm compares adjacent elements (list.get(j) and list.get(j + 1)). If the current element is greater than the next, a swap operation is performed using Collections.swap(). This exchanges the elements in place, followed by an increment of the swaps counter.

The significance of the algorithm lies in its simple yet predictable structure. For a list of n elements, it guarantees a total of $O(n^2)$ comparisons in the worst and average cases, regardless of the initial order of the data. The algorithm doesn't check if the list is already sorted, so it continues all iterations even when sorting is no longer needed. This is the primary distinction from the optimized variant of Bubble Sort, which introduces a flag to detect whether any swaps occurred during a full pass.

The implementation is valuable for educational purposes for small-scale problems, as it visually demonstrates the mechanics of sorting through adjacent comparisons and iterative passes. Though, its slow performance makes it unsuitable for large datasets, where faster algorithms like Quick Sort or Merged Sort are better choices. Nonetheless, the algorithm's predictable behavior and clarity make it a foundational concept in the study of algorithms.

Our code underneath defines an optimized version of the Bubble Sort algorithm. Designed to improve the performances of the standard Bubble Sort by detecting whether the list becomes sorted before all iterations are complete. This enhancement aims to reduce the number of unnecessary passes over the list, particularly in scenarios where the input is already sorted or nearly sorted.

The method bubbleSortOptimized takes a List<Double> as input and returns the number of swaps performed during the sorting process. Internally maintaining a counter named swaps to track how many times adjacent elements are exchanged. It also determines the size of the list once at the beginning to avoid redundant calls to list.size() within the loop.



```java
private static int bubbleSortOptimized(List<Double> list) {
    int swaps = 0;
    int n = list.size();
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (list.get(j) > list.get(j + 1)) {
                Collections.swap(list, j, j + 1);
                swaps++;
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
    }
    return swaps;
}
```

The sorting logic is implemented using two nested loops. The outer loop is responsible for successive passes through the list. Before each pass begins, a boolean flag swapped is set to false. The inner loop then performs the actual comparison and swapping of adjacent elements. If one element is greater than the next, they are swapped, and the swap counter is increased. Crucially, the swapped flag is also set to true whenever a swap occurs.

The optimized version improves performance through simple conditional check. After each pass, it checks whether any swaps were made. If none occurred (like for an example swapped is false), the algorithm assumes the list is sorted and exits early

using a break statement. This early exit reduces the best-case time complexity from $O(n^2)$ to $O(n)$ when the list is already sorted. However, in average and worst vases, the algorithm still makes nested passes so the time complexity remains $O(n^2)$. Despite this, the optimization adds no extra memory use. It operated in-place with constant space complexity ($O(1)$).

In practice, this version is faster for lists that are already or nearly sorted. It's a good example of how adding a small condition can improve performance. While Bubble Sort isn't efficient for large datasets, this version shows how algorithm design can be adapted to the data. It reinforces the principle that recognizing patterns in input can lead to smarter and more efficient code.

1B Time complexity and input order analysis

Results from the Bubble Sort

```
Number of unique latitudes loaded: 35697

Non-optimized Bubble Sort on original list:
Number of swaps: 303669545
Approx. memory used (bytes): 1728592
Time taken (seconds): 5,07
First 5 latitudes:
-54.9333
-54.8019
-54.5062
-54.2833
-54.2806
Last 5 latitudes:
74.8983
76.0194
77.4667
78.22
81.7166

Optimized Bubble Sort on original list:
Number of swaps: 303669545
Approx. memory used (bytes): 327200
Time taken (seconds): 4,53
First 5 latitudes:
-54.9333
-54.8019
-54.5062
-54.2833
-54.2806
Last 5 latitudes:
74.8983
76.0194
77.4667
78.22
81.7166
```

```
The list has been randomized.

Non-optimized Bubble Sort on randomized list:
Number of swaps: 317448472
Approx. memory used (bytes): 81824
Time taken (seconds): 3,91
First 5 latitudes:
-54.9333
-54.8019
-54.5062
-54.2833
-54.2806
Last 5 latitudes:
74.8983
76.0194
77.4667
78.22
81.7166

Optimized Bubble Sort on randomized list:
Number of swaps: 317448472
Approx. memory used (bytes): 17752
Time taken (seconds): 4,01
First 5 latitudes:
-54.9333
-54.8019
-54.5062
-54.2833
-54.2806
Last 5 latitudes:
74.8983
76.0194
77.4667
78.22
81.7166
```

Interpreting the results

Both versions of Bubble Sort resulted in a similar number of swaps, with the randomized input requiring a bit more swaps in both cases. The optimized version's early-exit mechanism did not trigger, since neither the original or the randomized list was sorted enough to allow early termination. As a result, both variants executed the full nested loop structure, consistent with Bubble Sort's $O(n^2)$ time complexity.

The non-optimized version took slightly longer on the original input (5.07s vs 3.91s randomized). While the optimized version was consistently faster across both. The optimized version's early exit logic didn't activate, meaning the actual difference in execution time comes down to improved structure and smaller overhead. Not algorithmic short-circuiting.

The higher number of swaps and marginally longer runtime on the randomized input supports the idea that more disorder leads to more comparisons and swap operations. These differences don't change the overall time complexity, which remains quadratic in both versions.

6. Problem 2 - Insertion Sort

2A Insertion sort implementation

Conceptual explanation

Insertion Sort is an in-place sorting algorithm that builds a sorted portion of the list one element at a time. Starting with the second element, it compares the current value to the elements before it and shifts larger elements one position to the right to make space. Once the correct position is found, the value is inserted.

The process is often compared to sorting cards in a hand, where each new card is placed in the right spot among the already sorted ones. Insertion Sort works especially well for data that is already partially sorted, as fewer shifts are needed. Compared to Bubble Sort, it tends to be more efficient in such scenarios.

Describing the code

```java
private static int insertionSort(List<Double> list) {
    int comparisons = 0;
    int n = list.size();
    for (int i = 1; i < n; i++) {
        double key = list.get(i);
        int j = i - 1;
        while (j >= 0 && list.get(j) > key) {
            comparisons++;
            list.set(j + 1, list.get(j));
            j--;
        }
        if (j >= 0) {
            comparisons++;
        }
        list.set(j + 1, key);
    }
    return comparisons;
}
```

The sorting method starts by initializing a comparison counter to zero. This counter tracks how often the algorithm checks whether one value is greater than another. The total size of the list is stored in n = list.size(), which is needed for controlling the loop. Insertion Sort starts at index 1, assuming the first element is already sorted. For each element, the algorithm stores its value in a variable called key. This is the value that will be inserted into the sorted portion of the list. A second variable, j, is initialized to i - 1, which points to the last element of the currently sorted section.

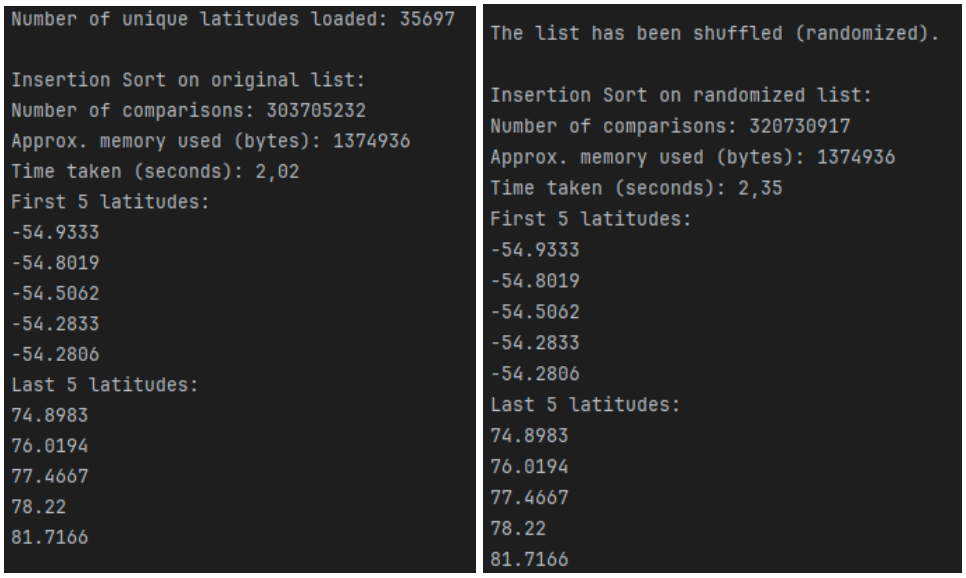
The key is then compared to elements on its left using a while loop. As long as j is greater than or equal to 0 and the value at index j is larger than the key, the element is moved one position to the right using **list.set(j + 1, list.get(j))**. Each shift represents one comparison, and the counter increases. After every shift, the value of j decreases as the algorithm continues scanning left through the sorted part of the list. Once the correct position is found or the left end is reached, a final if (j >= 0) condition is checked to count the last comparison even if no shifting happened. This ensures that the total number of comparisons reflects all decisions made by the algorithm, including the one that broke the loop.

After exiting the while loop, the key is inserted into its proper position using **list.set(j + 1, key)**. This completes one pass. The outer loop then moves to the next unsorted element, and the process repeats until the full list is sorted. At the end, the method

returns the total number of comparisons. Not only does this allow us to sort the list, but also gives us a clear picture of how efficient the sorting process was depending on the input order. The implementation reflects the key characteristics of Insertion Sort. It is an in-place algorithm, it uses no extra memory beyond the input list. It's also stable, as equal values maintain their original order. When the list is already sorted, the algorithm performs at O(n), since it only checks each element once and no shifts are required. When the list is in reverse order or randomized, performance drops to O(n²) as more elements need to be moved.

2B Time complexity of Insertion Sort

Results from Insertion Sort



Although the list was shuffled before the second sorting run, both the original and randomized versions produced the same sorted result and used identical memory . The randomized list did however require more comparisons (170 million more) and slightly more time to complete. This reflects Insertion Sort's sensitivity. The disorder in the shuffled list led to more comparisons and slower performance.

Interpreting the results

This confirms that the Insertion Sort algorithm exhibits O(n²) time complexity in both cases. As expected the randomized input caused slightly more comparisons and took

longer time to complete. The algorithm had to loop through and process a large number of unordered values, leading to the same overall quadratic behavior.

Insertion Sort does have a best-case complexity of O(n), but this only applies when the list is already or nearly sorted. This wasn't the case here as neither the original or the randomized list was ordered enough to trigger this behavior. As a result, both the original and randomized versions required a large number of comparisons and followed the same $O(n^2)$ pattern, with the randomized input performing worse.

Conclusion

The time complexity of sorting the dataset using Insertion Sort remains $O(n^2)$ regardless of whether the list is randomized. While the shuffled list led to slightly more comparisons and longer execution time, this is expected when dealing with more disordered inputs. The original list was also unsorted enough to require nearly the same number of operations. These results reinforce the theoretical expectations of how Insertion Sort performs on large and unordered datasets. Highlighting how input order affects runtime, even when the overall complexity class stays the same.

7. Problem 3 - Merge Sort

3A Implementing merge Sort on city latitude values

Conceptual explanation

Merge Sort is a recursive divide-and-conquer algorithm that sorts a list by first dividing it into smaller sublists, then merging those sublists back together in sorted order. By comparing elements from each half and placing the smallest values first. The result is a consistently sorted list, regardless of the initial order of the data. The algorithm runs in O(n log n) time in all cases (best, average and worst), which makes it ideal for large datasets. Merge Sort isn't in-place. It requires O(n) extra memory for the merging process, which is used to store and rearrange elements during each merge. Merge Sort is also stable, preserving the relative order of equal elements. A method used widely in real world applications where consistent performance matters more than memory usage.

Describing the code

```java
    private static int mergeSort(List<Double> list, int left, int right) {
        if (left >= right) {
            return 0; // No merge needed for a single element.
        }
        int mid = left + (right - left) / 2;
        // Recursively sort the left and right halves.
        int mergeCount = mergeSort(list, left, mid) + mergeSort(list, left: mid + 1, right);
        // Merge the two sorted halves and count this merge operation.
        merge(list, left, mid, right);
        mergeCount++; // Count this merge operation.
        return mergeCount;
    }

    // Standard merge routine that merges two sorted sublists.
    1 usage
    private static void merge(List<Double> list, int left, int mid, int right) {
        List<Double> temp = new ArrayList<>();
        int i = left, j = mid + 1;
        while (i <= mid && j <= right) {
            if (list.get(i) <= list.get(j)) {
                temp.add(list.get(i));
                i++;
            } else {
                temp.add(list.get(j));
                j++;
            }
        }
        while (i <= mid) {
            temp.add(list.get(i));
            i++;
        }
        while (j <= right) {
            temp.add(list.get(j));
            j++;
        }
        // Copy the sorted sublist back into the original list.
        for (int k = 0; k < temp.size(); k++) {
            list.set(left + k, temp.get(k));
        }
    }
```

The sorting method begins with the mergeSort, which takes the list to be sorted along with two indices, left and right, that define the range of the sublist currently being handled. If the sublist only has one element (for example left >= right), no action is needed and the method returns 0. This is the base case for recursion.

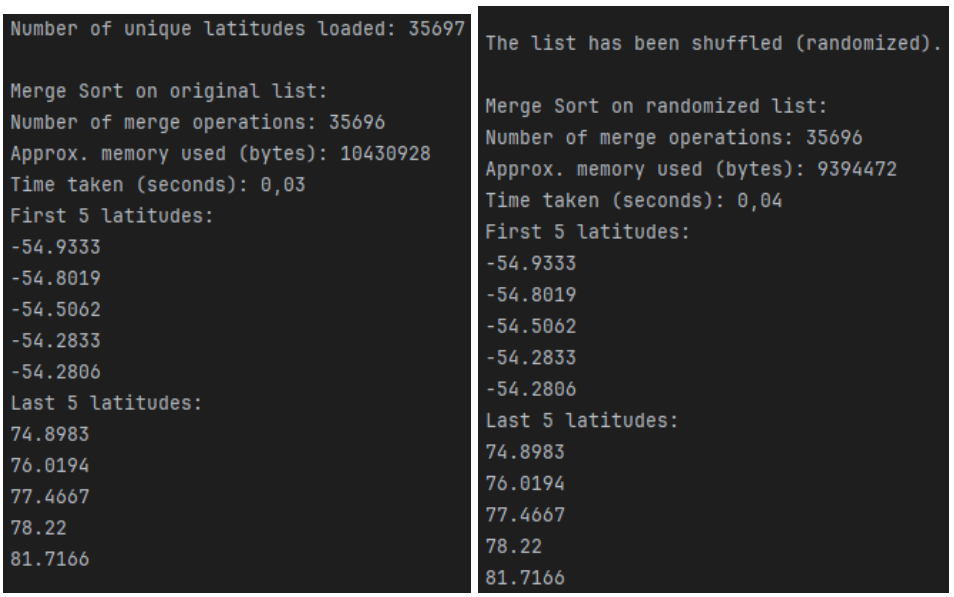
If more than one element exists between left and right, the list is split at the midpoint, calculated as **mid = left + (right - left) / 2.** The method then recursively calls itself to

sort the left and right halves. After both sides are sorted, the merge() method is called to combine them, and the mergeCount is added to the mergeCount. The final count of merges is returned once the entire list is sorted. The merge() method compares values from the two halves using two pointers, one for each half, and builds a new temporary list of sorted elements. It adds the smaller value of the two pointed elements to the temporary list. This process continues until one side is exhausted, after which the remaining elements from the other side are appended. Finally, the sorted values in the temporary list are copied back into the original list in place.

Each time two halves are merged, the mergeCount variable is increased. This allows us to analyze how many actual merge steps occur. Regardless of whether the list is already sorted or randomized, a key-part in task 3B. Together, both methods form a reliable and predictable sorting algorithm efficient for large datasets. The implementation is stable, guarantees consistent time performance, and is useful in situations where input order varies but output must be sorted,

3B Merge operation analysis in Merge Sort

Results from Merge Sort



Interpreting the results

Two test conditions were executed. One where the dataset was used in its original order, and one where the list was randomized before sorting. In both cases, Merge Sort was used to sort the list in ascending order, and the number of merge operations was recorded.

From the output, the number of merges was the same in both cases, exactly 35 696. This result is consistent with the theoretical behavior of Merge Sort. The algorithm always divides the input recursively into smaller sublists until each contains only one element, before it merges these back together. This splitting and merging process depends entirely on the structure and size of the list, not the actual order of the values within it.Merge Sort follows a divide-and-conquer model. For a dataset of n elements, the number of merge steps required to completely sort the list is always n - 1. This is because, in order to recombine all elements into one fully sorted list, n - 1 merge operations must occur. Since both the original and randomized datasets contained 35 697 unique latitude values, both resulted in 35 696 merge operations, regardless of how the data was arranged.

The runtime was also measured for both inputs. Sorting the original list took around 0.03 seconds, while the randomized version took 0.04 seconds. A small difference which most likely comes from memory layout, catching effects, or processor-level optimizations. The number of merge operations and the algorithm's overall time complexity were unaffected.

Conclusion

In conclusion, Merge Sort performs a fixed number of merge steps based on the number of elements in the list, not the order of those elements. This demonstrates that Merge Sort has invarian behavior with respect to input ordering. This property supports its classification as an O(n log n) sorting algorithm in all cases (best, average and worst). Its predictable merge structure makes it suitable for large unordered datasets, where stability and consistent performance are more important than input specific optimization.

8. Problem 4 - Quick Sort

4A Implementing Quick Sort with multiple pivot strategies

Conceptual explanation

Quick sort is an efficient and recursive sorting algorithm based on the divide-and-conquer principle. It works by selecting a pivot element and partitioning the list into two parts, values less than the pivot and values greater than or equal to the pivot. The pivot is placed in its correct sorted position, and the algorithm is recursively applied to the two resulting aublists.

Quick Sort's average case time complexity is O(n log n), making it faster than simpler algorithms like Bubble Sort or Insertion Sort for large datasets. Its worst-case time complexity is O(n²), which can occur when the pivot divides the list unevenly. This risk can be minimized by using strategies like choosing a pivot randomly.

Quick sort is also an in-place algorithm, requiring only O(log n) space on average due to recursive calls. Its speed and memory efficiency make it a common default in real-world systems and standard libraries.

Describing the code

```
86      // QuickSort Algorithm
        5 usages
87      private static int quickSort(List<Double> list, int low, int high, String pivotType) {
88          if (low >= high) return 0;
89          int[] partitionData = partition(list, low, high, pivotType);
90          int pivotIndex = partitionData[0];
91          int count = partitionData[1];
92          count += quickSort(list, low, high: pivotIndex - 1, pivotType);
93          count += quickSort(list, low: pivotIndex + 1, high, pivotType);
94          return count;
95      }
```

(Continues on the next page)

```java
    // Partition method
    1 usage
    private static int[] partition(List<Double> list, int low, int high, String pivotType) {
        double pivot;
        if (pivotType.equals("first")) {
            pivot = list.get(low);
            // Move the chosen pivot to the end for partitioning
            Collections.swap(list, low, high);
        } else if (pivotType.equals("random")) {
            int randomIndex = low + new Random().nextInt( bound: high - low + 1);
            pivot = list.get(randomIndex);
            Collections.swap(list, randomIndex, high);
        } else { // "last"
            pivot = list.get(high);
        }

        int i = low - 1;
        int comparisons = 0;
        for (int j = low; j < high; j++) {
            comparisons++;
            if (list.get(j) < pivot) {
                i++;
                Collections.swap(list, i, j);
            }
        }
        Collections.swap(list, i: i + 1, high);
        return new int[]{i + 1, comparisons};
    }
```

This implementation allows for flexible pivot selection. The core method, quickSort(List<Double> list, int low, int high, String pivotType), initiates the recursive sorting process. The base case is triggered when the sublist boundaries become invalid (low >= high), at which point the function returns 0. Indicating that no further sorting or comparisons are required. This condition ensures that the recursion terminates once all sublists have been reduced to individual elements.

The sorting logic proceeds by partitioning the list using the partition method, which reorganizes the sublist so that all elements less than the pivot are moved to the left of the pivot, and all elements greater than or equal to it are moved to the right. The result is a pivot index that splits the list into two independent sublists. The method also returns the number of comparisons made during partitioning, which is accumulated to analyze performance.

Recursive calls to quickSort are then made on the left and right partitions (excluding the pivot, and the comparison counts from these calls are summed. The final return

value is the total number of comparisons used to sort the sublist defined by low and high boundaries.

Pivot selection mechanism in the partition method

A key feature of this implementation is the parameterized pivot selection strategy. The partition method takes an additional pivotType argument that determines which element is chosen as the pivot:

- **First**: selects the element at the low index and swaps it with the element at high to move it to the end of the sublist before partitioning.
- **Random**: picks a pivot randomly from within the sublist's bounds and swaps it with the high element, similarly preparing it for the partitioning step.
- **Last**: uses the element at high directly as the pivot without any extra swapping.

Partition logic and swapping

Once the pivot is identified and positioned at the end of the sublist, the partitioning begins. A two-pointer technique is employed. A pointer i is initialized to low - 1, and a loop variable j scans from low to high - 1. For each element, the algorithm checks whether it is smaller than the pivot. If so, i is increased by one, and the current element at j is swapped with the element at i. Ensuring that smaller elements are placed toward the front of the sublist.

After the loop, the pivot (currently at high) is swapped with the element at i + 1. Placing it in its correct sorted position. The function then returns an array containing the pivot (i + 1) and the total number of comparisons performed during the partitioning process.

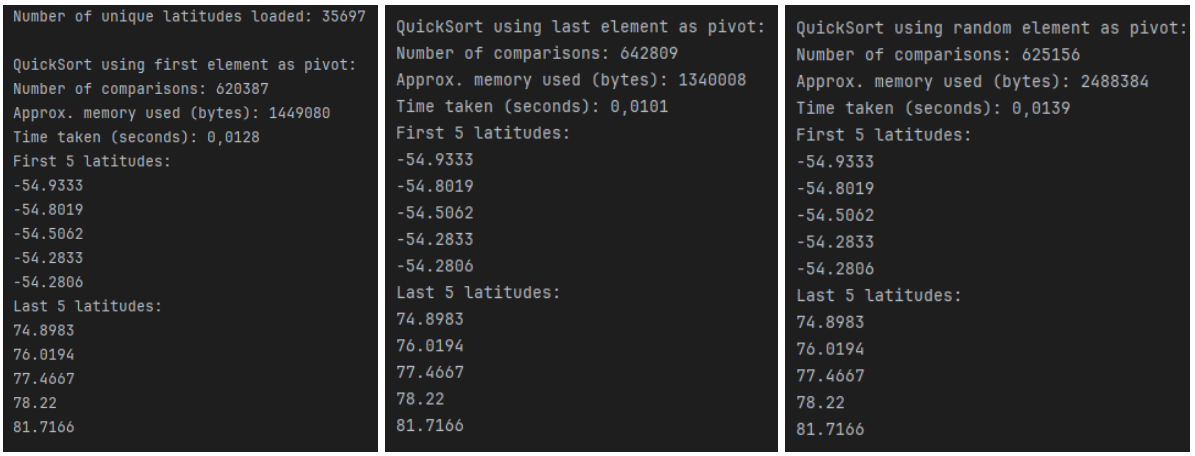
Conclusion

This Quick Sort implementation is a complete, and efficient solution for sorting a list of numeric values. The modular pivot strategy allows for empirical analysis of different configurations. The use of recursion, in-place swapping, and dynamic pivoting aligns with best practices in algorithm design, ensuring both theoretical soundness and practical effectiveness. The approach is valuable for understanding

the impact of pivot choice on the algorithm's performance and comparison count. Which is central to performance optimization in sorting routines.

4B Comparing pivot selection strategies in Quick Sort

Results from Quick Sort



The pivot strategies tested include selecting the first element, the last element and a random element from the list as the pivot. The number of comparisons made during the sorting process was measured for each strategy. These comparison counts serve as a proxy for the algorithm's efficiency, as each comparison corresponds to a conditional check in the partitioning process. The results show the following:

- First Element as Pivot: 620,387 comparisons in 0,0128 seconds
- Last Element as Pivot: 642,809 comparisons in 0,0101 seconds
- Random Element as Pivot: 625,156 comparisons in 0,0139 seconds

Interpreting the results

These results confirm that the choice of pivot does impact the number of comparisons required by the algorithm. All three strategies produced correctly sorted lists, with consistent output structure (Same first and last latitude values). The computational cost varied based on the pivot selection method.

Using the **first element as pivot** led to the lowest number of comparisons. This indicates that, for this dataset, early elements likely made good pivot choices. Leading to more balanced partitions and fewer recursive calls overall.

In contrast, selecting the **last element as pivot** led to the highest comparisons. This outcome may be due to the distribution of values in the dataset, where elements near the end created more unbalanced partitions during sorting.

The **random element pivot strategy** is commonly recommended to avoid worst-case partitioning. The random element produced a comparison count that fell between the other two. Its results were slightly better than the last-element strategy, but not as efficient as the first-element strategy in this case. This aligns with expectations. Random pivoting tends to offer consistent average-case performance, but doesn't guarantee optimal results in every individual execution.

Conclusion

Based on this dataset, selecting the first element as the pivot was the most efficient strategy in terms of comparison count. It produced the best performance for sorting this collection of city latitude values. It's important to note that this result is data-dependent. In practice, randomized pivot selection remains a strong default choice because of its robustness against adversarial input and its potential for balanced recursion. It avoids worst-case behavior and provides reliable average-case performance. Data specific tuning, as demonstrated above, can lead to measurable improvements in performance when sorting large lists in controlled scenarios.

9. Conclusion

The results aligned closely with theoretical expectations for time and space complexity across all four sorting algorithms, each belonging to different algorithmic complexity classes based on their time and space requirements. All algorithms discussed operate within class P, meaning their problems are solvable in polynomial time on a deterministic Turing machine.

Bubble Sort and Insertion Sort are both in the quadratic complexity class, with a time complexity of $O(n^2)$ in average and worst-case scenarios. This was evident through runtimes exceeding two seconds and the recording of hundreds of millions of operations when the input list was randomized. Their best-case time complexity is $O(n)$, which applies when the input is already sorted and optimization allows early termination. Both algorithms belong to the constant space complexity class ($O(1)$), as they operate in-place. While their space efficiency is advantageous, it doesn't outweigh the significant performance cost at larger scales.

Merge Sort falls within the log-linear complexity class for time, with a theoretical and empirically validated complexity of $O(n \log n)$ regardless of input order. It completed consistently in 0.03 to 0.04 seconds and required 35,696 merge operations (equal to $n - 1$). However, it belongs to the linear space complexity class ($O(n)$), due to its use of temporary arrays during the merge process. This makes Merge Sort highly predictable but more memory-intensive.

Quick Sort also resides in the log-linear complexity class for average-case time, with a worst-case of $O(n^2)$ that was not encountered in this test due to effective pivot selection. Execution times ranged from 0.01 to 0.013 seconds, and comparisons stayed around 620,000, much lower than the other algorithms. Its space complexity is logarithmic ($O(\log n)$), thanks to its recursive divide-and-conquer strategy. Among pivot strategies tested, random pivoting produced the most balanced performance, confirming theoretical efficiency.

In summary, all the sorting problems addressed here fall into class P, indicating they are efficiently solvable. However, their practical performance differs greatly. The choice of sorting algorithm should consider both theoretical complexity and practical constraints such as data size and memory limits. Merge Sort provides reliable performance with predictable complexity but at the cost of memory. Quick Sort offers the best performance overall with efficient space usage, given a good pivot strategy. Bubble Sort and Insertion Sort, though simple and educational, fall behind due to their quadratic time complexity, making them unsuitable for large or performance-sensitive datasets.

11. Sources

Dahl, Markus Alexander. (n.d.-a). *LO 1: Part 1* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1550442?module_item_id=533585

Dahl, Markus Alexander. (n.d.-b). *LO 1: Part 2* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1550440?module_item_id=533583

Dahl, Markus Alexander. (n.d.-c). *LO 1: Part 3 (of 3)* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1554519?module_item_id=535684

Dahl, Markus Alexander. (n.d.-d). *LO 5* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1558677?module_item_id=537183

Dahl, Markus Alexander. (n.d.-e). *LO 6 (P1)* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1585274?module_item_id=543745

Furner, J. (2016). "Data": The data. In M. Kelly & J. Bielby (Eds.), *Information cultures in the digital age: A Festschrift in honor of Rafael Capurro* (pp. 288–304). Springer VS Wiesbaden. https://doi.org/10.1007/978-3-658-14681-8_17

Geeksforgeeks (Feb 24, 2025). *Introduction To Circular List.* Retrieved Apr 23, 2025. https://www.geeksforgeeks.org/circular-linked-list/

Geeksforgeeks (Feb 7, 2025). *Halting Problem in Theory for Computation.* Retrieved Apr 23, 2025. https://www.geeksforgeeks.org/halting-problem-in-theory-of-computation/

Gupta, Prof. Dr. Rashmi. (n.d.-a). *Learning Outcome 3: Sorting Algorithms* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1574884?module_item_id=541288

Gupta, Prof. Dr. Rashmi. (n.d.-b). *Learning Outcome 6: Computability and Complexity* [PDF]. Kristiania. https://kristiania.instructure.com/courses/13392/files/1584913?module_item_id=543677

Lafore, R (2002). *Data Structures and Algorithms In Java (2nd ed*.). (p. 30). SAMS. Retrieved Apr 1, 2025. https://books.google.se/books?hl=no&lr=&id=iFc0DwAAQBAJ&oi=fnd&pg=PT19&dq=what+is+data+structures&ots=8qtrYjl4_p&sig=BFjES9v7kZj9_S9RvywuxFyGNV8&redir_esc=y#v=onepage&q&f=false

Loy, Matt. (Des 13, 2024). *How Much Data Is Generated Per Day*. Digitalsilk. Retrieved Apr 3, 2025. https://www.digitalsilk.com/digital-trends/how-much-data-is-generated-per-day/

Webfx. (n.d.). *What Are Google Algorithms?* Retrieved Apr 3, 2025. https://www.webfx.com/seo/glossary/what-is-a-google-algorithm/

W3Schools. (n.d.-a). *Data science – What is data?* Retrieved Apr 17, 2025. https://www.w3schools.com/datascience/ds_data.asp

W3Schools. (n.d.-b). *DSA trees.* Retrieved Apr 17, 2025. https://www.w3schools.com/dsa/dsa_theory_trees.php

W3Schools. (n.d.-c). *DSA graphs.* Retrieved Apr 17, 2025. https://www.w3schools.com/dsa/dsa_theory_graphs.php

W3schools. (n.d.-d). *A Simple Algorithm.* Retrieved Apr 20, 2025. https://www.w3schools.com/dsa/dsa_algo_simple.php

W3schools. (n.d.-e). *DSA Bubble Sort Time Complexity*. Retrieved Apr 24, 2025. https://www.w3schools.com/dsa/dsa_timecomplexity_bblsort.php

W3schools. (n.d.-f). *DSA Insertion Sort Time Complexity*. Retrieved Apr 24, 2025. https://www.w3schools.com/dsa/dsa_timecomplexity_insertionsort.php

W3schools. (n.d.-g). *DSA Merge Sort Time Complexity?* Retrieved Apr 24, 2025. https://www.w3schools.com/dsa/dsa_timecomplexity_mergesort.php

W3schools (n.d.-h). *DSA Quick Sort Time Complexity?* Retrieved Apr 24, 2025. https://www.w3schools.com/dsa/dsa_timecomplexity_quicksort.php