UNIVERSITI TEKNOLOGI PETRONAS

SEMESTER MAY 2025

TEB1113

ALGORITHM AND DATA STRUCTURE

FINAL PROJECT : ALGORITHM COMPARISON

PROJECT TITLE : BOOKNEST

No.	Students Name	ID No.	Program
1	WAN AMISYA SYAFIRA BINTI WAN AZMAN	22009194	BACHELOR IN COMPUTER SCIENCE
2	AIMI SYAZWINA BINTI MOHD ZAKI	22011161	BACHELOR IN COMPUTER SCIENCE
3	AHMAD MUQRI BIN JOHARI	22010897	BACHELOR IN COMPUTER SCIENCE
4	ABDUL IZZANY BIN HELMY	22011630	BACHELOR IN COMPUTER SCIENCE

Table of Contents

1.0 INTRODUCTION

Sorting is a fundamental operation in computer science, forming the backbone of many software applications that require data organization and quick retrieval. Among the simplest and most widely taught sorting techniques are Bubble Sort and Selection Sort. Although both algorithms share a similar time complexity, they differ significantly in how they process and manipulate data. Bubble Sort works by repeatedly swapping adjacent elements to move the largest unsorted value toward the end of the list, whereas Selection Sort searches for the smallest remaining element and places it in its correct position in each iteration. These two approaches represent distinct algorithmic strategies and serve as excellent candidates for direct comparison.

This comparison becomes especially meaningful when applied to a real-world context. In this project, both algorithms are implemented in a mobile application called BookNest—an interactive platform that allows users to manage and organize their personal book collections. Books are presented as visual cards and can be sorted alphabetically by title. By embedding both sorting techniques within the app, users can experience the differences in performance and behaviour firsthand through visual feedback and user interaction.

The study is not merely theoretical; it emphasizes the practical implications of algorithm selection in user-facing applications. For small to medium datasets, which are typical in personal library apps, understanding the nuances between simple sorting methods can influence the overall responsiveness and usability of the app. Moreover, integrating algorithm visualization into BookNest enhances user engagement while also serving as an educational tool for those unfamiliar with sorting algorithms.

Through this project, the comparison of Bubble Sort and Selection Sort provides valuable insight into algorithm efficiency, user experience, and the broader role of algorithm design in the development of interactive applications.

2.0 ALGORITHM EXPLANATION AND CODE

2.1 HOW IT WORKS

Bubble Sort

- Bubble Sort is a simple sorting algorithm that works by comparing two items next to each other and swapping them if they are in the wrong order. This process is repeated over and over until the entire list is sorted. After each full pass, the largest unsorted item "bubbles up" to its correct position at the end of the list.

  An optimized version of Bubble Sort includes a check to see if any swaps were made in a pass. If no swaps are needed, the algorithm stops early, which saves time when the list is already sorted or nearly sorted. While Bubble Sort is not efficient for large datasets due to its $O(n^2)$ time complexity, it is still useful for small lists, especially when the data is mostly in order. It is also a good teaching tool for understanding how sorting works step-by-step.

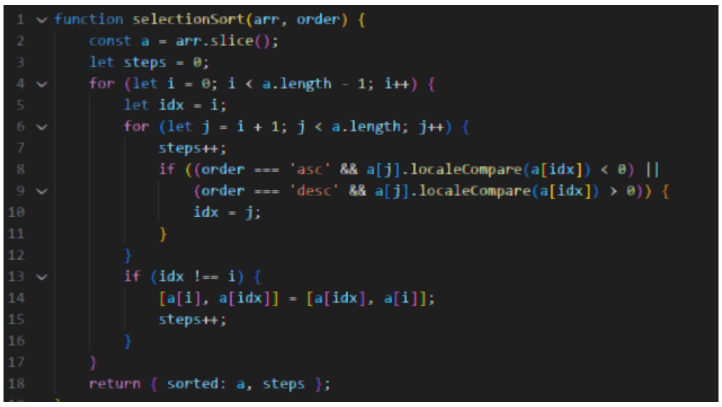
Selection Sort

- Selection Sort is a sorting algorithm that splits the list into two parts: a sorted section and an unsorted section. In each pass, it looks through the unsorted part of the list to find the smallest item, then swaps it with the first unsorted item. This way, the sorted section grows one element at a time.

  Unlike Bubble Sort, Selection Sort always goes through the entire unsorted list in every pass, even if the list is already mostly sorted. However, it only makes one swap per pass, which makes it more efficient when reducing the number of data movements is important. While Selection Sort still has $O(n^2)$ time complexity, its predictable behavior and low number of swaps make it useful in cases where memory writes should be minimized or where performance consistency is needed.

2.2 STEP BY STEP EXAMPLE

1. Selection Sort

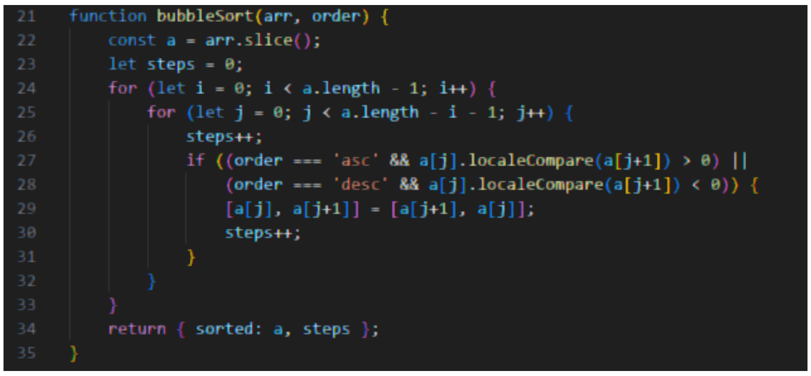


```
function selectionSort(arr, order) {
    const a = arr.slice();
    let steps = 0;
    for (let i = 0; i < a.length - 1; i++) {
        let idx = i;
        for (let j = i + 1; j < a.length; j++) {
            steps++;
            if ((order === 'asc' && a[j].localeCompare(a[idx]) < 0) ||
                (order === 'desc' && a[j].localeCompare(a[idx]) > 0)) {
                idx = j;
            }
        }
        if (idx !== i) {
            [a[i], a[idx]] = [a[idx], a[i]];
            steps++;
        }
    }
    return { sorted: a, steps };
```

Function :

- Takes an array of book titles and a sorting order (ascending or descending).
- Sorts the array using the Selection Sort algorithm.
- Tracks and returns the number of steps (comparisons and swaps).

2. Bubble Sort

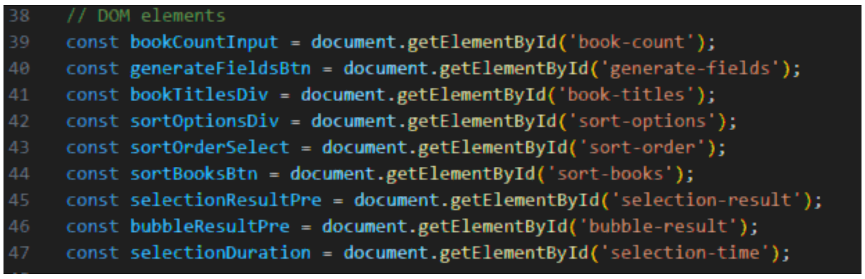


```
function bubbleSort(arr, order) {
    const a = arr.slice();
    let steps = 0;
    for (let i = 0; i < a.length - 1; i++) {
        for (let j = 0; j < a.length - i - 1; j++) {
            steps++;
            if ((order === 'asc' && a[j].localeCompare(a[j+1]) > 0) ||
                (order === 'desc' && a[j].localeCompare(a[j+1]) < 0)) {
                [a[j], a[j+1]] = [a[j+1], a[j]];
                steps++;
            }
        }
    }
    return { sorted: a, steps };
}
```

Function :

- Also takes an array and sort order.
- Sorts using the Bubble Sort method.
- Returns the sorted array and step count.

3. DOM Elements and Interaction

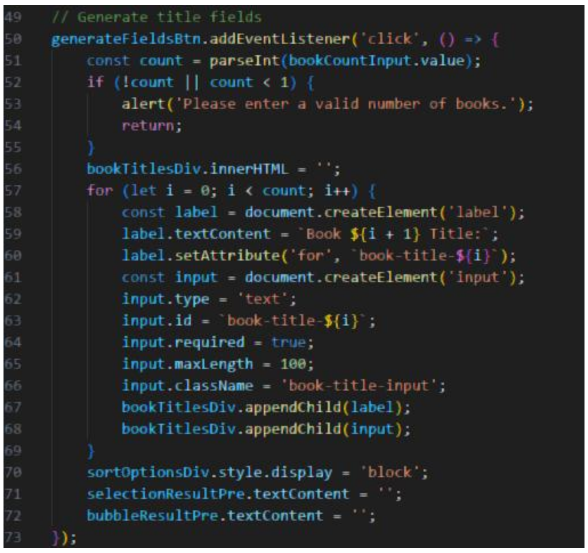


```javascript
// DOM elements
const bookCountInput = document.getElementById('book-count');
const generateFieldsBtn = document.getElementById('generate-fields');
const bookTitlesDiv = document.getElementById('book-titles');
const sortOptionsDiv = document.getElementById('sort-options');
const sortOrderSelect = document.getElementById('sort-order');
const sortBooksBtn = document.getElementById('sort-books');
const selectionResultPre = document.getElementById('selection-result');
const bubbleResultPre = document.getElementById('bubble-result');
const selectionDuration = document.getElementById('selection-time');
```

Explanation :

- Connects JavaScript to HTML using getElementById().
- Allows us to read from and write to specific elements on the webpage.

4. Generating Input Fields Dynamically

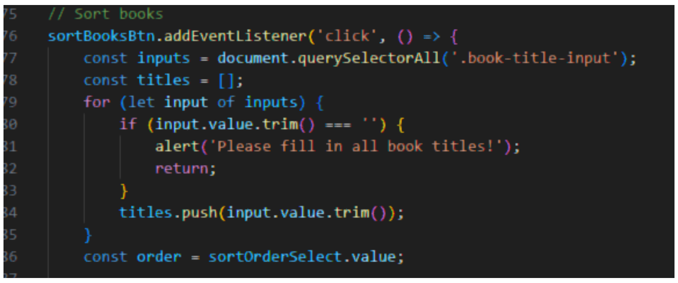


```javascript
// Generate title fields
generateFieldsBtn.addEventListener('click', () => {
    const count = parseInt(bookCountInput.value);
    if (!count || count < 1) {
        alert('Please enter a valid number of books.');
        return;
    }
    bookTitlesDiv.innerHTML = '';
    for (let i = 0; i < count; i++) {
        const label = document.createElement('label');
        label.textContent = `Book ${i + 1} Title:`;
        label.setAttribute('for', `book-title-${i}`);
        const input = document.createElement('input');
        input.type = 'text';
        input.id = `book-title-${i}`;
        input.required = true;
        input.maxLength = 100;
        input.className = 'book-title-input';
        bookTitlesDiv.appendChild(label);
        bookTitlesDiv.appendChild(input);
    }
    sortOptionsDiv.style.display = 'block';
    selectionResultPre.textContent = '';
    bubbleResultPre.textContent = '';
});
```

Function :

- Triggered when user clicks the "Generate Fields" button.
- Reads the number of book titles the user wants to input.
- Clears old inputs (if any) using innerHTML = ''.
- Creates a loop to dynamically add input fields for book titles.
- Appends labels and inputs to the bookTitlesDiv.
- Displays the sorting options (sortOptionsDiv).
- Clears any previous results.

5. Handling the Sort Operation

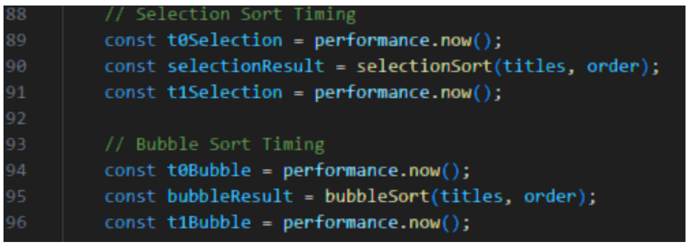


```
// Sort books
sortBooksBtn.addEventListener('click', () => {
    const inputs = document.querySelectorAll('.book-title-input');
    const titles = [];
    for (let input of inputs) {
        if (input.value.trim() === '') {
            alert('Please fill in all book titles!');
            return;
        }
        titles.push(input.value.trim());
    }
    const order = sortOrderSelect.value;
```

Function :

- Stores the book titles in an array.
- Gets the **selected order** (ascending or descending) from the dropdown.

6. Running and Timing Both Sort Algorithms



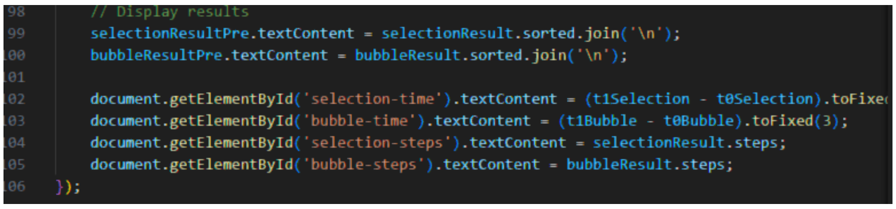
```
    // Selection Sort Timing
    const t0Selection = performance.now();
    const selectionResult = selectionSort(titles, order);
    const t1Selection = performance.now();

    // Bubble Sort Timing
    const t0Bubble = performance.now();
    const bubbleResult = bubbleSort(titles, order);
    const t1Bubble = performance.now();
```

Function :

- Measures how long each algorithm takes to run.
- Uses performance.now() to get timestamps before and after sorting.
- Calls both selectionSort() and bubbleSort() with the same titles and order.

7. Displaying Results



```
    // Display results
    selectionResultPre.textContent = selectionResult.sorted.join('\n');
    bubbleResultPre.textContent = bubbleResult.sorted.join('\n');

    document.getElementById('selection-time').textContent = (t1Selection - t0Selection).toFixe
    document.getElementById('bubble-time').textContent = (t1Bubble - t0Bubble).toFixed(3);
    document.getElementById('selection-steps').textContent = selectionResult.steps;
    document.getElementById('bubble-steps').textContent = bubbleResult.steps;
});
```

Displays:

- Time taken for Selection Sort and Bubble Sort (in milliseconds, 3 decimal places).
- Number of steps (comparisons and swaps) performed by each algorithm.

2.3 TIME AND SPACE COMPLEXITIES

Bubble Sort

Best Case: O(n)

- The best-case scenario occurs when the input list is already completely sorted. In this case, Bubble Sort only needs a single pass through the list to verify that no swaps are necessary. If implemented with an optimization (such as a boolean flag that tracks whether any swaps occurred), the algorithm will detect the sorted state and terminate early. This allows Bubble Sort to achieve linear time performance in the best case, making it particularly efficient for nearly sorted or pre-sorted data.

Average Case: $O(n^2)$

- In the average case, when elements are arranged in a random order, Bubble Sort performs a large number of comparisons and swaps across multiple passes. For each pass, it compares adjacent elements and moves the largest remaining value toward the end of the list. This results in a total number of operations that grow quadratically with the input size, making Bubble Sort inefficient for moderate to large datasets under normal conditions.

Worst Case: $O(n^2)$

- The worst-case occurs when the list is sorted in reverse order. In this case, every possible pair of elements must be compared and swapped, as the smallest elements are far from their correct positions. Bubble Sort performs the maximum number of comparisons and swaps, leading to a time complexity of $O(n^2)$. This makes it unsuitable for sorting large datasets with poor initial ordering.

Space Complexity: O(1)

- Bubble Sort is an in-place sorting algorithm, which means it does not require additional data structures to perform the sorting. Only a constant amount of extra memory is used, typically one or two temporary variables to assist with element swapping. This makes it memory-efficient and easy to implement, especially in environments with limited memory.

Selection Sort

Best Case: $O(n^2)$

- Selection Sort performs the same number of comparisons regardless of how sorted the input is. Even if the list is already in perfect order, the algorithm still scans the entire unsorted portion of the list to find the minimum element on every pass. While it may perform fewer swaps in this case, it still completes $n(n-1)/2$ comparisons, resulting in a best-case time complexity of $O(n^2)$.

Average Case: $O(n^2)$

- With randomly ordered data, Selection Sort performs consistently. On each pass, it finds the smallest element in the unsorted section and moves it to its correct position. The total number of comparisons remains the same as the best and worst cases, regardless of input distribution. Although it makes fewer swaps than Bubble Sort (only one per iteration), the overall runtime still grows quadratically, making it inefficient for large datasets.

Worst Case: $O(n^2)$

- Even in the worst-case scenario such as when the list is sorted in reverse order, Selection Sort behaves the same as it does in all other cases. It still performs the full number of comparisons and exactly one swap per pass. This makes it predictable but also highlights its lack of adaptability to different input arrangements. The time complexity remains $O(n^2)$.

Space Complexity: $O(1)$

- Selection Sort is an in-place algorithm, meaning it sorts the data without using extra space for another array or data structure. It only needs a constant amount of memory to hold variables like the current index, minimum index, and a temporary value for swapping. Its space complexity of $O(1)$ makes it memory-efficient and suitable for environments with limited storage or system resources.

2.4 PROS AND CONS OF THE ALGORITHM

Criteria	**Bubble Sort**	**Selection Sort**
Pros	Very easy to understand and implement Can be optimized to stop early if the list is already sorted Stable sorting algorithm (maintains relative order of equal elements) Useful for teaching basic concepts of comparison and swapping	Simple and easy to implement Performs minimal number of swaps (at most $n-1$), beneficial when swap operations are expensive Predictable behavior regardless of input order
Cons	Inefficient for large datasets due to high number of comparisons and swaps Performance depends heavily on input order Requires many unnecessary passes in worst case	Time complexity remains $O(n^2)$ in all cases, including already sorted data Not stable (can change the order of equal elements) Inefficient for large datasets

3.0 COMPARISON TABLE

Criteria	Bubble Sort	Selection Sort
Working Principle	Repeatedly compares and swaps adjacent elements until the entire list is sorted	Repeatedly selects the smallest unsorted element and places it in its correct position
Time Complexity	- Best Case: $O(n)$ - Average Case: $O(n^2)$ - Worst Case: $O(n^2)$	- Best Case: $O(n^2)$ - Average Case: $O(n^2)$ - Worst Case: $O(n^2)$
Space Complexity	$O(1)$ In-place sorting using a constant amount of memory	$O(1)$ Also sorts in-place with minimal memory usage
Number of Steps	- Comparisons: Up to $n(n-1)/2$ - Swaps: Can be up to $n(n-1)/2$ in worst case - Optimized versions reduce swaps and stop early if no swaps occur in a pass.	- Comparisons: Always $n(n-1)/2$ regardless of initial order - Swaps: Exactly $n-1$ swaps in all cases, making it more efficient for minimizing write operations.
Best Use Cases	- Ideal for small or nearly sorted datasets, where few swaps are needed. - Great for teaching and visualizing how sorting works step-by-step. - Useful in situations where simplicity matters more than performance.	- Preferred when the number of swaps must be minimized (e.g., limited memory write cycles). - Works well in embedded systems or scenarios where data movement is costly. - More consistent behavior regardless of input order.

4.0 USE CASE COMPARISON

- **Scenario 1: E-commerce Order Processing**

An e-commerce company processes customer orders hourly to apply tiered discounts (e.g., silver, gold, platinum). These orders are sorted by total purchase amount. Because orders are updated regularly throughout the day, the list retrieved every hour is often nearly sorted already.

Bubble Sort:

Bubble Sort works by comparing adjacent elements and swapping them if they're out of order. In this scenario, because the list is almost sorted, very few swaps are needed. When using an optimized version, the algorithm checks if any swaps occurred during a full pass. If not, it stops early, avoiding unnecessary passes. This makes Bubble Sort especially efficient in best-case or nearly sorted cases, reducing the time complexity from $O(n^2)$ to $O(n)$.

Its simplicity and ability to adapt to the list's condition make it ideal for repeated hourly operations where the data doesn't change much between cycles.

Selection Sort:

Selection Sort, on the other hand, always scans the entire unsorted portion of the list to find the minimum value, even if the list is already in order. It does not benefit from pre-sortedness and always performs the same number of comparisons which leading to $O(n^2)$ time complexity in all cases. This results in wasted processing time, especially when working with data that is nearly sorted..

Suitable Algorithm:

In this scenario, *Bubble Sort* is clearly the better choice. Its ability to take advantage of nearly sorted data and terminate early makes it significantly more efficient than Selection Sort, which performs a full round of operations regardless of the data's state. For a system that runs hourly and deals with minor updates between each cycle, Bubble Sort offers better performance and responsiveness.

➢ Scenario 2 : Classroom Attendance App

A teacher uses a classroom app to keep track of 25–30 student names for attendance. The student list is usually in alphabetical order but may have a few new names added or small edits made each day.

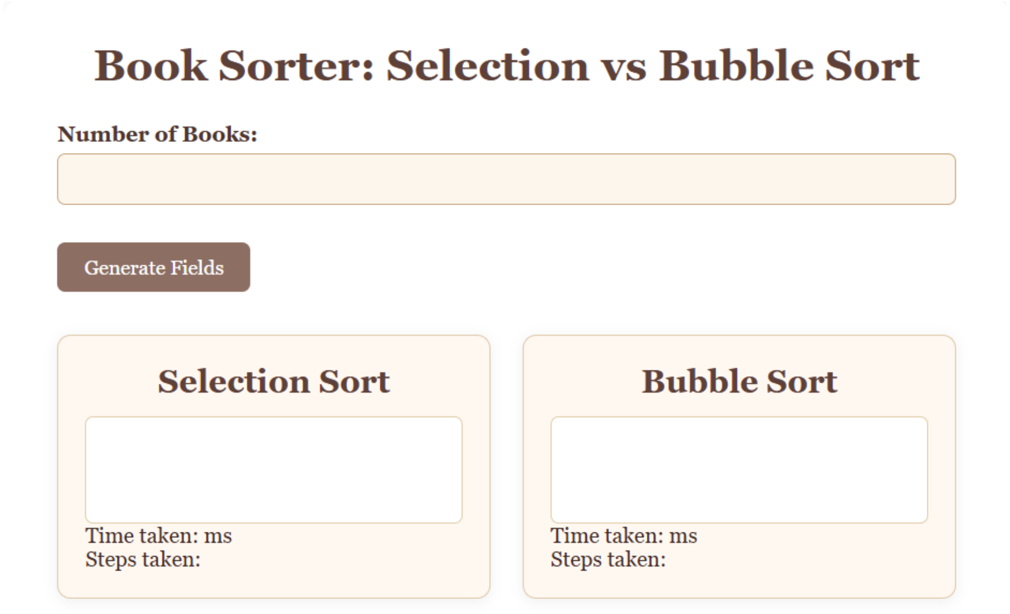
Bubble Sort:

Bubble Sort compares adjacent pairs of names and swaps them if they are out of order alphabetically. For a list that is nearly sorted like in this classroom setting, only a few swaps are needed. An optimized version of Bubble Sort can detect if no swaps occur during a full pass and will terminate early, often completing the sort in just one or two passes.
This makes Bubble Sort particularly suitable for small datasets that are frequently updated but largely ordered, as it saves time and resources while still maintaining accuracy.

Selection Sort:
Selection Sort works by scanning the entire remaining list in each pass to find the alphabetically smallest name, then swapping it with the current position. Even if the list is nearly in order, Selection Sort will still perform the same number of comparisons about $n(n-1)/2$ regardless of the actual disorder. This leads to unnecessary processing for only minor updates, which is inefficient for such a small and nearly sorted dataset.

Suitable Algorithm:
In this classroom setting, *Bubble Sort* is again the better choice. It is simpler to implement, handles small data efficiently, and performs exceptionally well on nearly sorted lists, exactly what this app deals with daily.

5.0 VISUAL AID



Book Sorter: Selection vs Bubble Sort

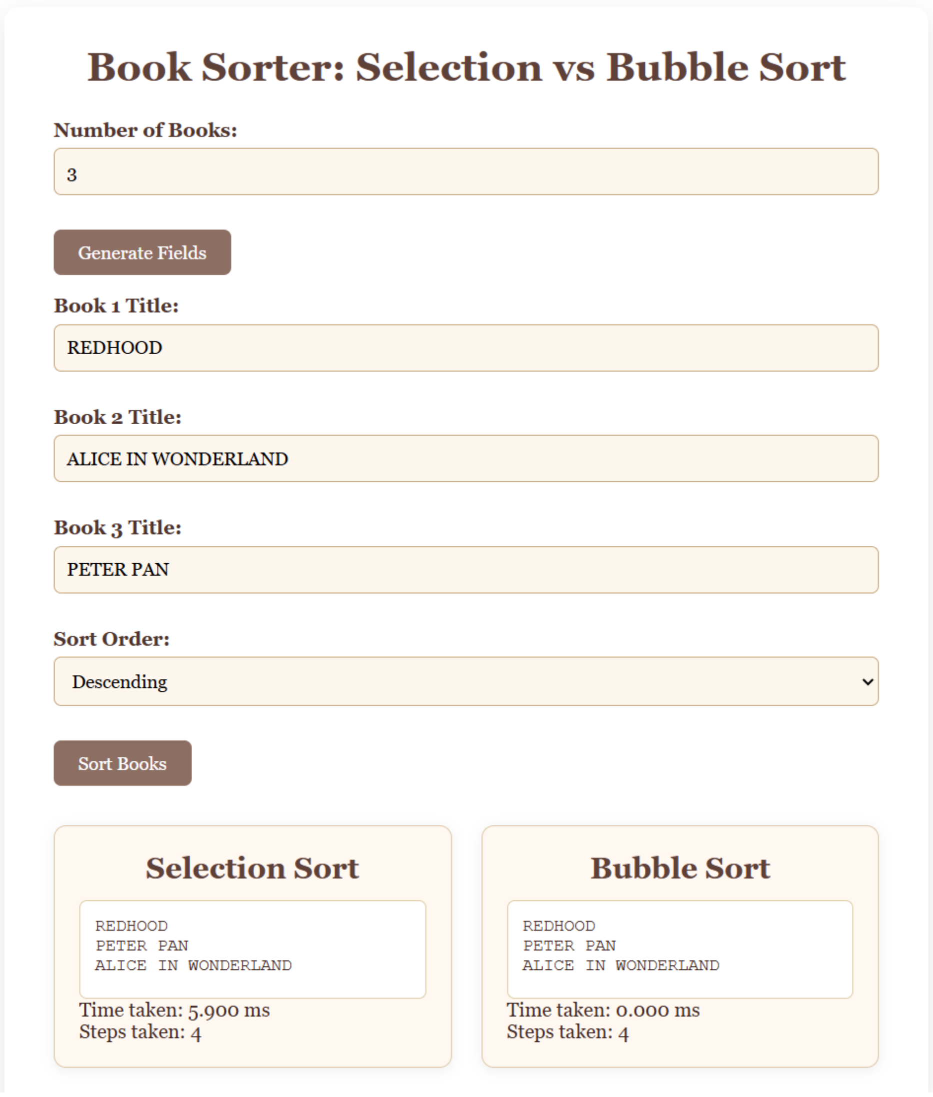
Number of Books:

Selection Sort

Time taken: ms
Steps taken:

Bubble Sort

Time taken: ms
Steps taken:



Book Sorter: Selection vs Bubble Sort

Number of Books: 3

Book 1 Title: REDHOOD

Book 2 Title: ALICE IN WONDERLAND

Book 3 Title: PETER PAN

Sort Order: Descending

Selection Sort

```
REDHOOD
PETER PAN
ALICE IN WONDERLAND
```

Time taken: 5.900 ms
Steps taken: 4

Bubble Sort

```
REDHOOD
PETER PAN
ALICE IN WONDERLAND
```

Time taken: 0.000 ms
Steps taken: 4

6.0 CONCLUSION

Based on our project, we can see that the JavaScript-based application effectively demonstrates how to use Selection Sort and Bubble Sort algorithms to organize a list of book titles chosen by users. Users can specify how many book titles they want to sort, enter them into generated fields, and choose to sort the titles in either ascending or descending order. The results appear in real-time, along with important performance metrics like execution time in milliseconds and the number of steps, which includes comparisons and swaps, taken by each algorithm.

Including both Selection Sort and Bubble Sort is important. Selection Sort is simple and efficient regarding swaps, making it suitable for small datasets and cases where minimizing write operations is essential. Bubble Sort may be less efficient because of the higher number of swaps, but it is easier to understand and visualize, which helps in learning how sorting works. Both algorithms compare elements but take different approaches. Selection Sort finds the minimum or maximum value during each pass, while Bubble Sort pushes the largest or smallest values to the end through adjacent swaps.

By letting users observe the detailed behavior and performance of each algorithm, the application serves as both a functional tool and an educational experience. It shows how the choice of algorithm can affect efficiency and behavior, especially with string-based data. Overall, this project illustrates the practical application of sorting algorithms, user interaction through dynamic DOM manipulation, and the ability to measure and assess algorithm performance in a real-world context.