Analysis of Insertion and Selection Sort Algorithms and Their Respective Implementations

Introduction

This document provides an enhanced analysis of the Insertion Sort and Selection Sort algorithms, focusing on their implementations, error handling, performance, and potential optimizations.

Sorting Algorithm Overview
"Sorting is a very classic problem of reordering items (that can be compared, e.g., integers, floating-point numbers, strings, etc) of an array (or a list) in a certain order (increasing, non-decreasing (increasing or flat), decreasing, non-increasing (decreasing or flat), lexicographical, etc)." (VisualAlgo)

Insertion Sort: Psuedocode

```
mark first element as sorted

for each unsorted element X

  'extract' the element X

  for j = lastSortedIndex down to 0

    if current element j > X

      move sorted element to the right by 1

    break loop and insert X here
```
(VisualAlgo)

Insertion Sort: Implementation and Enhancements

```java
/**
     * Sorts the given array using the Insertion Sort algorithm.
     *
     * @param array. An integer array to be sorted
```

```
     * @throws IllegalArgumentException if the array is null
     *
     *                                   The Insertion Sort algorithm sorts an
array
     *                                   by building a sorted subarray one
element at
     *                                   a time.
     *                                   It iterates through the array,
selecting
     *                                   each element (key) and inserting it
into the
     *                                   correct
     *                                   position in the sorted subarray. The
method
     *                                   prints the key selected and the state
of the
     *                                   array
     *                                   after each insertion for debugging
purposes.
     */
public static void insertionSort(int[] anArray) {
        if (anArray == null) {
            throw new IllegalArgumentException("Array cannot be null"); //
throws an exception if the array is null
        }
        int n = anArray.length; // sets n to the length of the array

        for (int i = 1; i < n; i++) { // sets i as 1, sets the condition for i
to stop when it reaches n, which is the
                                      // total length of the array, and sets
the increment of i by 1
            int key = anArray[i]; // sets key as the value of the element at
the current index of i
            int j = i - 1; // sets j as i-1, which is the element one index
behind i

            // This code segment selects the current element to be inserted
into the sorted
            // subarray. The variable `key` holds the value to be inserted,
while `j`
            // represents the index of the last element in the sorted
subarray.
```

```
            System.out.println("Insertion Sort - Key selected: " + key);

            while (j >= 0 && anArray[j] > key) {// A while loop is to run that
will continue to run as long as the index
                                              // of j is greater than or equal
to zero and the value of the element at
                                              // the current index of j is
greater than the value of the key. i.e if
                                              // it's in the incorrect natural
order.
                anArray[j + 1] = anArray[j]; // sets the value of the element
at the current index of j to the value of the
                                         // element at the current index of
j+1. This effectively shifts the element at
                                         // the current index of j to the right
by one index
                // This loop shifts elements encountered that greater than
`key` to the right
                // until the condition of the while loop is no longer met.
                // This means that the key value

                j = j - 1; // decrements j by 1, this serves to ensure that
the while loop will continue
                           // from the previous index of j until the condition
of the while loop is no
                           // longer met to reach the beginning of the array.
                // The while continues until an element less than or equal to
`key` is found or the beginning of the array is reached.

                System.out.println("Insertion Sort - Array after shifting: " +
Arrays.toString(anArray));
            }
            anArray[j + 1] = key;
            System.out.println("Insertion Sort - Array after inserting key: " +
Arrays.toString(anArray));
        }
    }
```
Time Complexity and Space Complexity Analysis
- **Time Complexity**:

  - **Best Case**: O(n) - This occurs when the array is already sorted. In this scenario, Insertion Sort only needs to iterate through the array once to confirm it's already sorted, resulting in a linear time complexity.
  - **Worst Case**: O(n^2) - This occurs when the array is sorted in reverse order. In this scenario, Insertion Sort needs to perform the maximum number of comparisons and swaps, leading to a quadratic time complexity.
  - **Average Case**: O(n^2) - The average case time complexity of Insertion Sort is also O(n^2), which is the same as the worst-case scenario. This is because the algorithm's performance is heavily influenced by the initial order of the array, and the average case is often close to the worst-case scenario.

- **Space Complexity**: O(1) - Insertion Sort is an in-place sorting algorithm meaning it does not require any additional space proportional to the input size. Insertion sort alters the array by shifting elements within the array itself, without creating a new array or using significant extra memory.

Diagram for Insertion Sort
`![alt text](InsertionSortLRDiagram.png)`

Output Log with No Logging

Output Log

```
Original Array: [5, 2, 8, 1, 3]
Pass 1: [2, 5, 8, 1, 3] // 2 is inserted into its correct position
Pass 2: [2, 5, 8, 1, 3] // 8 is already in the correct position
Pass 3: [1, 2, 5, 8, 3] // 1 is inserted at the beginning
Pass 4: [1, 2, 3, 5, 8] // 3 is inserted into its correct position
Sorted Array: [1, 2, 3, 5, 8]
```
Output Log of the Insertion Sort Algorithm with Enhanced Logging and Comments
```java
int [] anArray = [5, 2, 8, 1, 3]

insertionSort(anArray)

Output Log:
Original Array for Insertion Sort: [5, 2, 8, 1, 3] // j is set to five and key is set to two
Insertion Sort - Key selected: 2
```

```
Insertion Sort - Array after shifting: [5, 5, 8, 1, 3] // while loop encounters the condition array[j] > key, so the value of the element at the current index of j is set to the value of the element at the current index of j+1. This effectively shifts the element at the current index of j to the right by one index to the current index value of the key.
// Two is shifted to the left and five is shifted to the right.
Insertion Sort - Array after inserting key: [2, 5, 8, 1, 3] // the while loop ends and the value of the key is set to the value of the element at the current index of j+1. This completes the first incteration of the outer loop and i is incremented by one.
Insertion Sort - Key selected: 8 // The next key is selected and the inner loop condition is checked.
Insertion Sort - Array after inserting key: [2, 5, 8, 1, 3] // The inner loop condition is not met, since key is great than the value of the element at the current index of j, and the inner loop ends.
Insertion Sort - Key selected: 1 // The next key is selected and the inner loop condition is checked.
Insertion Sort - Array after shifting: [2, 5, 8, 8, 3] // Since the value of the key is less than the value of the element at the current index of j, the value of the element at the current index of j is set to the value of the element at the current index of j+1. This effectively shifts the element at the current index of j to the right by one index to the current index value of the key.
Insertion Sort - Array after shifting: [2, 5, 5, 8, 3] // The key value is one. This value is less than all other elements in the array so it continues to shift to the beginnig of the array.
Insertion Sort - Array after shifting: [2, 2, 5, 8, 3] // All elements of array[j]  are greater than key so j is decremented by one each iteration of the inner loop and the key value is checked against the new array[j] value each time.
Insertion Sort - Array after inserting key: [1, 2, 5, 8, 3] // J becomes -1 when the key reaches the beginning of the array and the while loop ends.
Insertion Sort - Key selected: 3 // The next key index is selected by the outer loop
Insertion Sort - Array after shifting: [1, 2, 5, 8, 8] // The inner loop array[j] is greater than key so the value of the element at the current index of j, which is one index before because j is decremented by one, set to the value of the element at the current index of j+1. This shifts the elements to the left and compares each element to the key value until the inner loop conditional is no longer met.
Insertion Sort - Array after shifting: [1, 2, 5, 5, 8] // Key value continues to shift to the left until the inner loop conditional is no longer met.
```

Insertion Sort - Array after inserting key: [1, 2, 3, 5, 8] // Key value is greater than the value of the element at the current index of j so the inner loop ends and the key value is set to the value of the element at the current index of j+1.
Sorted Array using Insertion Sort: [1, 2, 3, 5, 8]

```

### Error Handling and Edge Cases

- **Null Handling**: The method throws an `IllegalArgumentException` if the input array is null, ensuring that the algorithm does not attempt to process a null type.


## Selection Sort: Psuedocode

```
repeat (numOfElements - 1) times

  set the first unsorted element as the minimum

  for each of the unsorted elements

    if element < currentMinimum

      set element as new minimum

  swap minimum with first unsorted position
```
(VisualAlgo)

## Selection Sort: Implementation

```java
/**
     * Sorts the given array using the selection sort algorithm.
     *
     * @param array the array to be sorted

     * @throws IllegalArgumentException if the array is null
     *

```
     *                                    The selection sort algorithm sorts an
array
     *                                    by repeatedly finding the minimum
element
     *                                    (considering ascending order) from the
     *                                    unsorted part and putting it at the
     *                                    beginning.
     *                                    It uses two nested loops to iterate
through
     *                                    the array and swap the elements
according to
     *                                    whether or not the values of the two
     *                                    elements are greater or less than each
     *                                    other.
     *                                    If they are in the incorrect natural
order,
     *                                    which is not in ascending order, the
     *                                    elements at the current index of the
nested
     *                                    loop and the minimum index of the
outer loop
     *                                    are swapped.
     *
     *                                    1. The subarray which is already
sorted.
     *                                    2. The remaining subarray which is
unsorted.
     *
     *                                    In every iteration of the selection
sort,
     *                                    the minimum element (considering
ascending
     *                                    order)
     *                                    from the unsorted subarray is picked
and
     *                                    moved to the sorted subarray.
     *
     *                                    The method prints the starting index,
the
     *                                    current minimum index, and the array
at each
     *                                    step
```

```
     *                                   of the sorting process for debugging
     *                                   purposes.
     */
    public static void selectionSort(int[] array) {
        if (array == null) {
            throw new IllegalArgumentException("Array cannot be null"); //
throws an exception if the array is null
        }
        int n = array.length; // sets n to the length of the array
        for (int i = 0; i < n - 1; i++) { // sets i as zero, sets the condition
for i to stop when it reaches n-1, which
                                          // is the last index of the zero
indexed array, and sets the increment of i by
                                          // 1
            int minIndex = i;// sets i as minimum index value of the array,
            boolean swapped = false; // boolean to check if a swap has
occurred
            System.out.println("Selection Sort - Starting index: " + i); //
prints the starting index value of the
                                                                          //
array, which is always zero in the first
                                                                          //
logging since the loop is only in the first
                                                                          //
iteration

            for (int j = i + 1; j < n; j++) { // sets j as i+1, which is the
element one index ahead of i, sets the
                                              // condition for j to stop when
it reaches the value of n, which is the
                                              // total length of the array.
This is set to this value because j is an
                                              // index value ahead of i, so it
will reach the last index of the array
                                              // ahead of i,
                                              // and sets the increment of j by
1
                if (array[j] < array[minIndex]) { // condition to check if the
value of the element at the current index
                                                  // of j is less than the
value of the element at the minimum index.
```

```
                                                        // This will be true if the
value of the element at the current index
                                                        // of j is less than the
value of the element at the minimum index
                    minIndex = j; // sets the index of j as the new minimum
index since the value of array[j]
                                  // ahead of the minimum index array[i] is
less than the value at the minimum
                                  // index
                    swapped = true; // set swapped to true since a new minimum
was found
                    // Since the element ahead of the minimum index is less
than the element at the
                    // minimum index, the index values of the two elements are
swapped to be in
                    // ascending order, and the swapped boolean is set to
true.
                }
                System.out.println(
                        "Selection Sort - Current j: " + j + ", Current
minIndex: " + minIndex + ", Current array: "
                                + Arrays.toString(array));
            }

            if (swapped) { // only triggers if a swap has occurred
                int temp = array[minIndex]; // sets the value of the element at
the minimum index to a temporary
                                            // variable
                array[minIndex] = array[i]; // sets the value of the element at
the minimum index to the value of the
                                            // element at the current index of
i since the origional value of the
                                            // element at the minimum index was
stored in the temporary variable
                array[i] = temp; // sets the value of the element at array[i]
as the temperatory storage
                                 // variable, which is the value that was
swapped and pulled from the j index.
                System.out.println("Selection Sort - Swapped elements at
indices " + i + " and " + minIndex + ": "
                        + Arrays.toString(array)); // prints the swapped
elements at the indices of i and minIndex
```

````
            }
        }
    }
}
```
### Time and Space Complexity
- **Time Complexity**:
  - Best Case: O(n^2) - When the input is already sorted in ascending order, the algorithm still needs to iterate through the entire array to find the minimum element in the unsorted portion.
  - Worst Case: O(n^2) - When the input is sorted in descending order, the algorithm needs to iterate through the entire array for each element in the unsorted portion.
  - Average Case: O(n^2) - In most cases, the algorithm needs to iterate through the entire array for each element in the unsorted portion, leading to a quadratic time complexity.

- **Space Complexity**: O(1) - Selection Sort is an in-place sorting algorithm, meaning it doesn't require any extra space other than the input array.


 ### Diagram for Selection Sort
![alt text](InsertionSortLRDiagram.png)


### Output Log

```
Original Array: [5, 2, 8, 1, 3]
Pass 1: [1, 2, 8, 5, 3] // 1 is swapped with 5
Pass 2: [1, 2, 8, 5, 3] // 2 is already in the correct position
Pass 3: [1, 2, 3, 5, 8] // 3 is swapped with 8
Pass 4: [1, 2, 3, 5, 8] // 5 is already in the correct position
Sorted Array: [1, 2, 3, 5, 8]
```

#### Output Log of the Selection Sort Algorithm with Enhanced Logging and Comments


```java
int [] anArray = [5, 2, 8, 1, 3]
````

```
Output Log:
selectionSort(anArray)
Original Array for Selection Sort: [5, 2, 8, 1, 3]
Selection Sort - Starting index: 0 // i begins at 0
Selection Sort - Current j: 1, Current minIndex: 1, Current array: [5, 2, 8, 1, 3] // j begins at 1, minIndex is incremented to 1
Selection Sort - Current j: 2, Current minIndex: 1, Current array: [5, 2, 8, 1, 3] // inner loop continues until the end of the array (condition j < n)
Selection Sort - Current j: 3, Current minIndex: 3, Current array: [5, 2, 8, 1, 3] // element value one is less than the current minIndex value so minIndex is set to the j index. Boolean swapped is set to true.
Selection Sort - Current j: 4, Current minIndex: 3, Current array: [5, 2, 8, 1, 3] // j continues to the end of the array.
// This highlights why the time complexity of selection sort is O(n^2) as it has to iterate through the array entirely in the inner loop as each element [i] of the outer loop is compared to the all of the other elements of the array using the inner loop during each iteration of the outer loop.
Selection Sort - Swapped elements at indices 0 and 3: [1, 2, 8, 5, 3]
Selection Sort - Starting index: 1
Selection Sort - Current j: 2, Current minIndex: 1, Current array: [1, 2, 8, 5, 3]
Selection Sort - Current j: 3, Current minIndex: 1, Current array: [1, 2, 8, 5, 3]
Selection Sort - Current j: 4, Current minIndex: 1, Current array: [1, 2, 8, 5, 3] // None to swap so j continues to the end of the array.
Selection Sort - Starting index: 2
Selection Sort - Current j: 3, Current minIndex: 3, Current array: [1, 2, 8, 5, 3] // None to swap so j continues to the end of the array.
Selection Sort - Current j: 4, Current minIndex: 4, Current array: [1, 2, 8, 5, 3]
Selection Sort - Swapped elements at indices 2 and 4: [1, 2, 3, 5, 8]// Swap
Selection Sort - Starting index: 3
Selection Sort - Current j: 4, Current minIndex: 3, Current array: [1, 2, 3, 5, 8] // Swap
Sorted Array using Selection Sort: [1, 2, 3, 5, 8]
```

Error Handling and Edge Cases

````markdown
- **Null Handling**: Similar to Insertion Sort, this method throws an
`IllegalArgumentException` for null inputs.

### Performance Optimizations


- **Minimizing Swaps**: The algorithm only performs a swap ***if*** a new
minimum is found, reducing unnecessary operations.

## Design Patterns and Architectural Benefits

Both sorting algorithms can be viewed through the lens of design patterns:

- **Strategy Pattern**: The choice between Insertion Sort and Selection Sort
can be encapsulated using the Strategy Pattern, allowing for dynamic selection
of the sorting algorithm based on context (e.g., array size, order).
- **Decorator Pattern**: Enhancements such as logging or error handling can be
implemented using decorators, allowing for flexible and reusable code.

### Relevance in Modern Contexts

- **Microservices**: Sorting algorithms can be used in microservices for data
processing tasks, where efficient sorting is crucial for performance.
- **Reactive Systems**: In reactive programming, sorting can be applied to
streams of data, requiring efficient algorithms that can handle real-time data
flows.
- **Cloud-Native Applications**: Sorting algorithms are essential in
cloud-native applications for data management and processing, where scalability
and performance are critical.

## Testing Strategies and Error Management

- **Unit Testing**: Implement comprehensive unit tests using JUnit to validate
the correctness of the sorting algorithms, including edge cases such as null
inputs and empty arrays.
- **Error Management**: Use custom exceptions to provide more context in error
scenarios, enhancing the robustness of the code.

```
Works Cited
VisualAlgo. "Sorting." VisualAlgo, 2024,
https://visualgo.net/en/sorting?slide=1.
````
````

````
```
````
````