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

```
/**
 * Performs an insertion sort on the provided array.
 *
 * This method sorts the array in ascending order using the insertion sor
 * It iterates through each element, inserting it into the correct positi
 * already sorted portion of the array.
 *
 * @param anArray the array to be sorted; must not be null or empty
 * @throws IllegalArgumentException if the input array is null or empty
 */
public static void insertionSort(int[] anArray) {
    if (anArray == null || anArray.length == 0) {
        throw new IllegalArgumentException("Input array must not be null o
    }

    int n = anArray.length;

    // Outer loop: Iterates from the second element to the last, consider
    for (int i = 1; i < n; i++) {
        int key = anArray[i];
        int j = i - 1;

        // This code segment selects the current element to be inserted in
        System.out.println("Insertion Sort - Key selected: " + key);

        // Shift elements greater than 'key' one position to the right
        while (j >= 0 && anArray[j] > key) {
            anArray[j + 1] = anArray[j];
            // Move 'j' one position to the left
            j = j - 1;

            System.out.println("Insertion Sort - Array after shifting: " +
```

```
        }

        // Continue shifting until the correct insertion point for 'key' i
        anArray[j + 1] = key;
        System.out.println("Insertion Sort - Array after inserting key: "
    }
}
```

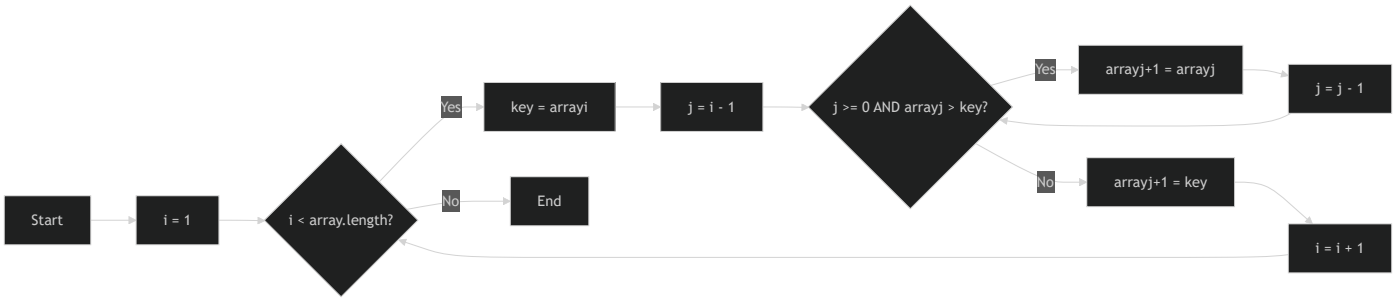
Time Complexity and Space Complexity Analysis of Insertion Sort

- **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, leading to approximately half the comparisons and shifts compared to the worst case, but still resulting in a quadratic time complexity overall as this is a nested loop.
- **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.

Stability of Insertion Sort

Insertion Sort is a stable sort. This means that if two elements are equal, their relative order is preserved in the sorted output.

Diagram for Insertion Sort



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

```
int [] anArray = {5, 2, 8, 1, 3};

insertionSort(anArray);

Output Log:
Original Array for Insertion Sort: [5, 2, 8, 1, 3] // Initial array
before sorting
Insertion Sort - Key selected: 2 // The first key to be inserted
Insertion Sort - Array after shifting: [5, 5, 8, 1, 3] // Element 5
is shifted right to make space for key 2
// Key 2 is now in the correct position, shifting 5 to the right.
Insertion Sort - Array after inserting key: [2, 5, 8, 1, 3] // Key 2
is inserted at its correct position
```

```
Insertion Sort - Key selected: 8 // The next key to be inserted
Insertion Sort - Array after inserting key: [2, 5, 8, 1, 3] // Key 8 is already in the correct position; no shifts needed

Insertion Sort - Key selected: 1 // The next key to be inserted
Insertion Sort - Array after shifting: [2, 5, 8, 8, 3] // Element 8 is shifted right to make space for key 1
Insertion Sort - Array after shifting: [2, 5, 5, 8, 3] // Element 5 is shifted right
Insertion Sort - Array after shifting: [2, 2, 5, 8, 3] // Element 2 is shifted right
Insertion Sort - Array after inserting key: [1, 2, 5, 8, 3] // Key 1 is inserted at the beginning

Insertion Sort - Key selected: 3 // The next key to be inserted
Insertion Sort - Array after shifting: [1, 2, 5, 8, 8] // Element 8 is shifted right to make space for key 3
Insertion Sort - Array after shifting: [1, 2, 5, 5, 8] // Element 5 is shifted right
Insertion Sort - Array after inserting key: [1, 2, 3, 5, 8] // Key 3 is inserted at its correct position

Sorted Array using Insertion Sort: [1, 2, 3, 5, 8] // Final sorted array
```

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.
- **Empty Array Handling**: The method throws an `IllegalArgumentException` if the input array is empty, ensuring that the algorithm does not attempt to process an empty array.

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

```
/**
 * Performs a selection sort on the provided array.
 *
 * This method sorts the array in ascending order using the selection sor
 * It repeatedly selects the minimum element from the unsorted portion an
 * the first unsorted element.
 *
 * @param array the array to be sorted; must not be null or empty
 * @throws IllegalArgumentException if the input array is null or empty
 */
public static void selectionSort(int[] array) {
    if (array == null || array.length == 0) {
        throw new IllegalArgumentException("Input array must not be null 
    }

    int n = array.length;

    // Outer loop: Iterates from the first element to the second-to-last 
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        boolean swapped = false;

        System.out.println("Selection Sort - Starting index: " + i);
```

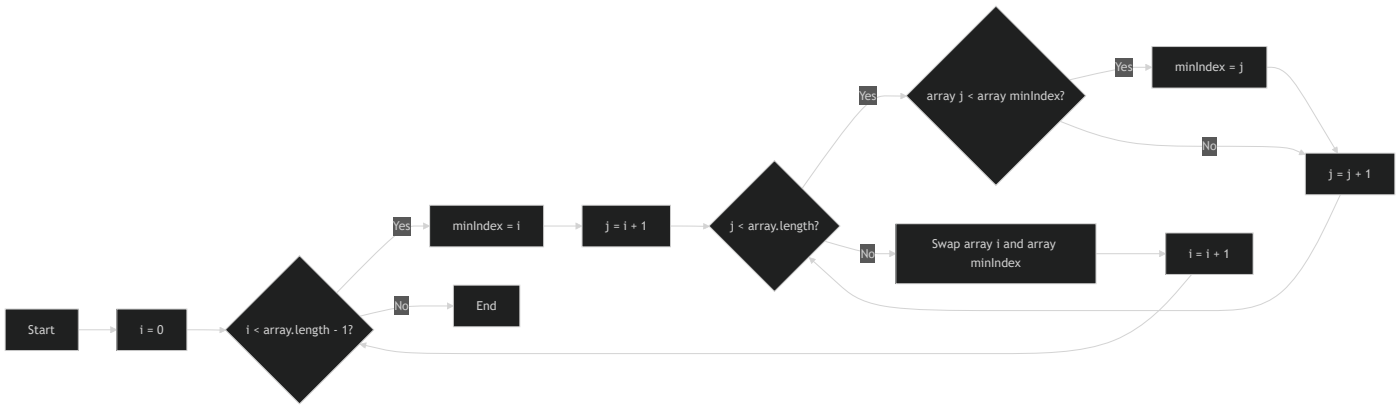
```
            // Inner loop: Iterates from the element after 'i' to the last el
            for (int j = i + 1; j < n; j++) {
                // Update minIndex if a smaller element is found
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                    swapped = true;
                }

                System.out.println(
                        "Selection Sort - Current j: " + j + ", Current minIn
                                + Arrays.toString(array));
            }

            // If a smaller element was found, swap it with the element at th
            if (swapped) {
                // Perform the swap using a temporary variable
                int temp = array[minIndex];
                array[minIndex] = array[i];
                array[i] = temp;

                System.out.println("Selection Sort - Swapped elements at indi
                        + Arrays.toString(array));
            }
        }
    }
```

Time and Space Complexity of Selection Sort

- **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 all 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 as described earlier in the analysis of the insertion sort algorithm.

Diagram for Selection Sort



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

```
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; // Outer loop begins with i = 0
Selection Sort - Current j: 1, Current minIndex: 1, Current array:
[5, 2, 8, 1, 3]; // Inner loop starts with j = 1; minIndex is set to
1
Selection Sort - Current j: 2, Current minIndex: 1, Current array:
[5, 2, 8, 1, 3]; // Inner loop continues; no change in minIndex
Selection Sort - Current j: 3, Current minIndex: 3, Current array:
[5, 2, 8, 1, 3]; // Found a smaller element (1); update minIndex to 3
Selection Sort - Current j: 4, Current minIndex: 3, Current array:
[5, 2, 8, 1, 3]; // Inner loop completes; minIndex remains 3
// This highlights the O(n^2) time complexity of selection sort, as
each element in the outer loop compares against all others in the
inner loop.
Selection Sort - Swapped elements at indices 0 and 3: [1, 2, 8, 5,
3]; // Swap the smallest found element value (1) with the first
unsorted element value (5)
Selection Sort - Starting index: 1; // Move to the next index
Selection Sort - Current j: 2, Current minIndex: 1, Current array:
[1, 2, 8, 5, 3]; // Inner loop starts; minIndex remains 1
Selection Sort - Current j: 3, Current minIndex: 1, Current array:
[1, 2, 8, 5, 3]; // No smaller element found; continue
Selection Sort - Current j: 4, Current minIndex: 1, Current array:
[1, 2, 8, 5, 3]; // No smaller element found; inner loop ends
Selection Sort - Starting index: 2; // Move to the next index
Selection Sort - Current j: 3, Current minIndex: 3, Current array:
[1, 2, 8, 5, 3]; // Inner loop starts; minIndex is set to 3
Selection Sort - Current j: 4, Current minIndex: 4, Current array:
[1, 2, 8, 5, 3]; // Found a smaller element value (3); update
minIndex to 4
Selection Sort - Swapped elements at indices 2 and 4: [1, 2, 3, 5,
8]; // Swap the smallest found element value (3) with the first
unsorted element value (8)
Selection Sort - Starting index: 3; // Move to the next index
Selection Sort - Current j: 4, Current minIndex: 3, Current array:
[1, 2, 3, 5, 8]; // Inner loop starts; no smaller element found
Sorted Array using Selection Sort: [1, 2, 3, 5, 8]; // Final sorted
array
```

Error Handling

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

Stability of Selection Sort

Selection Sort is not a stable sort. This means that if two elements are equal, their relative order is not guaranteed to be preserved in the sorted output.

Key Differences in Time Complexity

The time complexity of the algorithms highlights their efficiency differences.

- **Insertion Sort**: Constructs the sorted portion from left to right by inserting each element into its correct position within the already sorted subarray. This makes Insertion Sort more efficient for small or partially sorted datasets compared to Selection Sort.
- **Selection Sort**: Constructs the sorted portion from left to right by repeatedly selecting the minimum element from the unsorted portion and swapping it with the first unsorted element.

Time Complexity Analysis Comparison

- **Insertion Sort**:
  - **Best Case**: O(n) - When the input is already sorted, the algorithm only needs to iterate through the array once, confirming it's sorted.
  - **Worst/Average Case**: $O(n^2)$ - When the array is in reverse order or randomly ordered
  - **Reason**: It builds the sorted portion incrementally, inserting elements as needed.
- **Selection Sort**:
  - **Best Case**: $O(n^2)$ - Even if the input is sorted, the algorithm must still iterate through the entire array for each element in the unsorted portion.
  - **Reason**: It repeatedly selects the minimum element from the unsorted portion, requiring full traversal for each selection.

Insertion Sort and Selection Sort have different performance characteristics:

- Insertion Sort performs well on small datasets or partially sorted arrays. It's adaptive, meaning it can take advantage of existing order in the input.
- Selection Sort always performs the same number of comparisons, regardless of the initial order of the array. This makes it less efficient for partially sorted arrays.

Scenarios where Insertion Sort outperforms Selection Sort:

- Small datasets (typically < 10-20 elements)
- Nearly sorted arrays
- When writes are significantly more expensive than reads

Scenarios where Selection Sort might be preferred:

- When memory writes are very expensive compared to memory reads
- When simplicity of implementation is a priority

Works Cited

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