International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 14 Issue 3, March-2025

Brief Study about the variation of Complexities in Searching and Sorting Algorithms

Rohit Yadav[1], Mr. Manoj Kumar[2]

[1]M.Tech, Computer Science, Shri Venkateshwara University, Gajraula
[2]Assistant Professor, Computer Science Department, Shri Venkateshwara University, Gajraula

ABSTRACT

This paper presents a detailed comparative analysis of fundamental searching and sorting algorithms, focusing on their time and space complexities. The study examines linear search and binary search, evaluating their efficiency based on input size and data distribution. Additionally, it analyzes popular sorting techniques, including insertion sort, selection sort, bubble sort, merge sort, and quick sort, comparing their best, worst, and average-case performances. The analysis incorporates both theoretical complexity bounds and practical benchmarking, considering key operations such as comparisons, swaps, and memory usage. Experimental results validate theoretical predictions, offering insights into the optimal choice of algorithms based on different computational scenarios. This research aims to serve as a reference for developers and students in selecting appropriate algorithms for efficient data processing.

Keywords: Searching Algorithm Efficiency, Linear vs. Binary Search, Comparison of Sorting Techniques, Divide & Conquer Sorting, Sorting Performance Analysis, Complexity Variation Table, Execution Time Chart, Theoretical vs. Empirical Evaluation, Algorithm Benchmarking

Breif Introduction

Searching and sorting are fundamental operations in computer science, playing a crucial role in data organization and retrieval. Searching algorithms help locate an element in a dataset, while sorting algorithms arrange data in a specific order to improve efficiency in various applications. Among the searching techniques, linear search and binary search are widely used, while sorting is dominated by algorithms like insertion sort, selection sort, bubble sort, merge sort, and quick sort.

Each of these algorithms follows a unique approach and exhibits different performance characteristics based on input size, data distribution, and computational constraints. This paper provides an in-depth analysis of these algorithms, highlighting their efficiency in terms of time and space complexity

SEARCHING ALGORITHMS

1.1 Linear Search

Linear search is a simple searching algorithm that sequentially checks each element of a list until a match is found or the list is exhausted. It is applicable to both sorted and unsorted data, making it a straightforward but inefficient approach for large datasets. Linear search follows a brute-force technique, making it ideal for small or unordered datasets where setup time for more advanced searches is unnecessary.

1.2 Binary Search

Binary search is a highly efficient searching technique that operates on sorted datasets. It follows the divide-and-conquer approach, repeatedly dividing the search space in half until the desired element is found or the search space is empty. Binary search significantly reduces the number of comparisons, making it optimal for large datasets. However, it requires the dataset to be pre-sorted, which may add preprocessing overhead.

SORTING ALGORITHMS

2.1 Insertion Sort

Insertion sort is a simple and intuitive sorting technique that builds the sorted array one element at a time. It works by picking an element and inserting it into its correct position among the previously sorted elements. This algorithm is efficient for small datasets or nearly sorted data due to its adaptive nature. However, its performance degrades significantly for larger datasets, making it impractical for general-purpose sorting.

2.2 Selection Sort
Selection sort is a straightforward sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and places it in its correct position. This algorithm is easy to understand and implement but is inefficient for large datasets due to its $O(n^2)$ time complexity in all cases. Unlike insertion sort, selection sort does not adapt to partially sorted data, making it less efficient for practical applications.

2.3 Bubble Sort
Bubble sort is one of the simplest sorting algorithms, where adjacent elements are repeatedly compared and swapped if they are in the wrong order. The process continues until the array is fully sorted. While easy to implement, bubble sort is highly inefficient for large datasets, requiring multiple passes over the array. Despite its poor performance, it is useful for educational purposes and scenarios where minimal memory usage is required.

2.4 Merge Sort
Merge sort follows the divide-and-conquer paradigm, recursively dividing the array into two halves, sorting each half, and merging them back into a sorted sequence. This algorithm is highly efficient, with a consistent $O(n \log n)$ time complexity, making it suitable for large datasets. Unlike quick sort, merge sort is stable, ensuring that equal elements maintain their relative order. However, its additional space requirement makes it less efficient for in-place sorting.

2.5 Quick Sort
Quick sort is a divide-and-conquer sorting algorithm that selects a pivot element, partitions the array around the pivot, and recursively sorts the subarrays. It is known for its fast average-case performance of $O(n \log n)$ and is widely used in real-world applications. However, in its worst case (when the pivot is poorly chosen), quick sort can degrade to $O(n^2)$ complexity. Various optimizations, such as choosing a median pivot or using hybrid approaches, improve its efficiency..

Searching

1: Linear Search

Introduction:
Linear search is one of the simplest searching algorithms, based on a brute-force approach. It sequentially traverses each element of the list until the target element is found or the entire list is exhausted. The simplicity of linear search makes it easy to implement and applicable to both sorted and unsorted data structures.

Linear search follows a straightforward methodology: it starts at the first element of the array and compares it with the target value. If a match is found, the search terminates; otherwise, it continues until the last element. This approach ensures correctness but is inefficient for large datasets due to its $O(n)$ worst-case time complexity, where every element must be checked.

Despite its inefficiency in larger datasets, linear search remains important in the history of searching algorithms, serving as the foundation for more optimized searching techniques. It is particularly useful when the dataset is small, unstructured, or dynamically changing, where more complex search algorithms are not feasible.

Approach
The approach is to analyze the efficiency of linear search by studying its behavior in different scenarios, such as best-case, worst-case, and average-case complexities. Unlike recursive algorithms that use a divide-and-conquer paradigm, linear search follows a sequential approach, making it simple yet less efficient for large datasets.

Linear search works by iterating through each element of the list one by one until the desired element is found. The steps involved in the algorithm are:

1. Start from the first element of the array or list.
2. Compare the current element with the target element.
3. If a match is found, return the index (or position) of the element.
4. If no match is found, move to the next element.
5. Repeat the process until the end of the list is reached.

The best-case occurs when the element is found at the beginning (O(1)), while the worst-case occurs when the element is at the end or not present at all (O(n)). Unlike binary search, linear search does not require the data to be sorted, making it useful for dynamic and unordered datasets. However, its inefficiency in large datasets highlights the need for more optimized searching techniques.

ANALYSIS OF LINEAR SEARCH

Introduction
Linear Search is the simplest searching algorithm used to find an element in an array. It follows a sequential approach, where each element is compared one by one with the target element until a match is found or the entire list is traversed. It is also known as a sequential search and works efficiently for small datasets.

Working of Linear Search

1 Start from the first element of the array.
2. Compare the current element with the target element.
3. If a match is found, return the index of the element.
4. If not, move to the next element.
5. Repeat steps 2–4 until the end of the array is reached.
6. If the target element is not found, return -1.

Algorithm (Pseudo Code)

```
LinearSearch(A, n, key)
1. for i = 0 to n-1
2.     if A[i] == key
3.         return i  // Element found at index i
4. return -1  // Element not found
```

TIME COMPLEXITY ANALYSIS

Best Case (Ω(1))

1. The best-case scenario occurs when the target element is found at the first position.
2. The algorithm only needs one comparison in this case.

Worst Case (O(n))
The worst-case occurs when:

1. The element is found at the last position.
2. The element is not present in the array.
3. In this case, the algorithm performs n comparisons.

Average Case (Θ(n))
On average, the element may be found anywhere in the array.
The expected number of comparisons is n/2, leading to an average-case complexity of O(n).

Space Complexity Analysis
Linear Search has a space complexity of O(1) since it requires only a constant amount of extra space for variable.

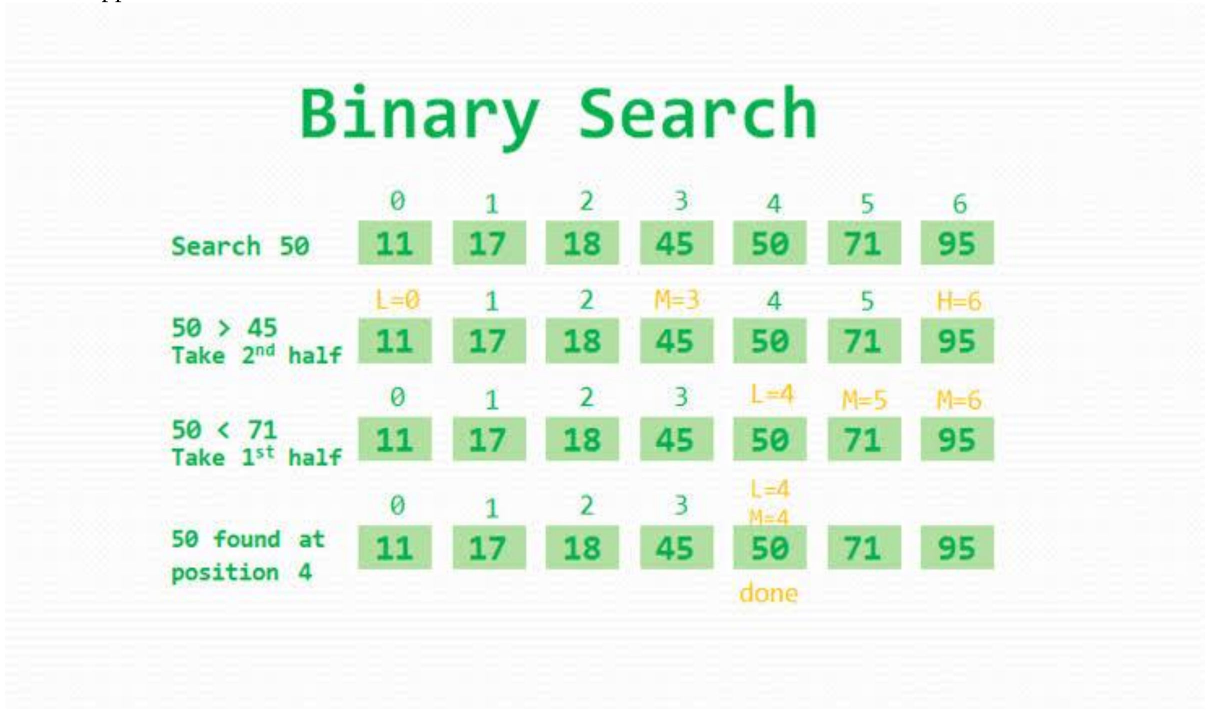
Advantages of Linear Search
1. Simple to implement – No complex logic is required.
2. Works on unsorted data – Unlike binary search, it doesn't require sorted input.
3. Useful for small datasets – Performs well for small arrays or lists.

Disadvantages of Linear Search
1. Inefficient for large datasets – As the array size grows, the time taken increases linearly.
2. High number of comparisons – Even if the element is not present, it still has to check all elements.

2: Binary Search

Introduction:
Binary Search is an efficient algorithm used for finding an element in a sorted array. It follows the divide-and-conquer approach, repeatedly dividing the search space into halves until the target element is found or determined to be absent.

Binary Search can be implemented in two ways:
1. Iterative Approach
2. Recursive Approach



Binary Search is an efficient algorithm used to find an element in a sorted array by repeatedly dividing the search range in half. In this example, we need to find 50 in the sorted array [11, 17, 18, 45, 50, 71, 95]. We start with L = 0 and H = 6, calculating the middle index M = 3, where the value is 45. Since 50 > 45, we discard the left half and update L = 4. Recalculating, M = 5, where the value is 71. Since 50 < 71, we discard the right half and update H = 4. Now, M = 4, where the value matches 50, confirming its position at index 4. This demonstrates how Binary Search efficiently finds an element in O(log n) time complexity.

1. Iterative Approach of Binary Search

Step-by-Step Explanation:
1. Initialize Pointers:
  low = 0 (starting index)
  high = n - 1 (last index of the array)
2. Find the Middle Element:

Compute the mid index using:
mid = \frac{low + high}{2}

3. Comparison:
If arr[mid] == target, return mid (element found).
If arr[mid] > target, search in the left half (high = mid - 1).
If arr[mid] < target, search in the right half (low = mid + 1).

4. Repeat the Process:
Continue until low > high (indicating that the element is not in the array).

Example of Iterative Binary Search:

Given Array: (Sorted)
arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

Target Element: 23
Step-by-Step Execution:

Iterative Binary Search - Pseudocode:

```
BinarySearchIterative(arr, n, target)
1. low = 0, high = n-1
2. while low ≤ high:
3.     mid = (low + high) / 2
4.     if arr[mid] == target:
5.         return mid  // Element found
6.     else if arr[mid] > target:
7.         high = mid - 1  // Search in left half
8.     else:
9.         low = mid + 1  // Search in right half
10. return -1  // Element not found
```

Time Complexity of Iterative Approach:

Iterative Binary Search
In the iterative approach, binary search uses a loop to repeatedly narrow down the search range. The key steps involve calculating the middle element and then adjusting the search range based on whether the target value is greater or less than the middle element. This process continues until the target value is found or the search range is exhausted.

Time Complexity:

Best Case: O(1) - This occurs when the target element is the middle element of the array.

Worst Case: O(log n) - This happens when the target element is either not present in the array or is at one of the ends. In each step, the search space is halved, leading to a logarithmic number of steps.

Average Case: O(log n) - On average, the target element will be found in logarithmic time relative to the size of the array.

Space Complexity:
O(1) → Only a few variables (low, high, mid) are used, so space usage is constant.

2. Recursive Approach of Binary Search

Step-by-Step Explanation:

1. Base Case:
If low > high, return -1 (element not found)

2. Compute Middle Index:
mid = (low + high) / 2.

3. Comparison:
If arr[mid] == target, return mid (found!).
If arr[mid] > target, recursively search in the left half.
If arr[mid] < target, recursively search in the right half.

Example of Recursive Binary Search:
Given Array: (Sorted)

arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

Target Element: 23

Recursive Calls Execution:

1. First Call: low=0, high=9, mid=4 → arr[mid] = 16 → Search right.
2. Second Call: low=5, high=9, mid=7 → arr[mid] = 56 → Search left.
3. Third Call: low=5, high=6, mid=5 → arr[mid] = 23 → Found! Return 5.

Recursive Binary Search - Pseudocode:

```
BinarySearchRecursive(arr, low, high, target)

1. if low > high:
2.     return -1  // Element not found
3. mid = (low + high) / 2
4. if arr[mid] == target:
5.     return mid  // Element found
6. else if arr[mid] > target:
7.     return BinarySearchRecursive(arr, low, mid-1, target)  // Search in left half
8. else:
9.     return BinarySearchRecursive(arr, mid+1, high, target)  // Search in right half
```

Time Complexity of Recursive Approach:

In the recursive approach, binary search is implemented by calling the function itself with updated parameters that narrow down the search range. The function checks if the middle element is the target; if not, it recursively searches either the left or the right half of the array.

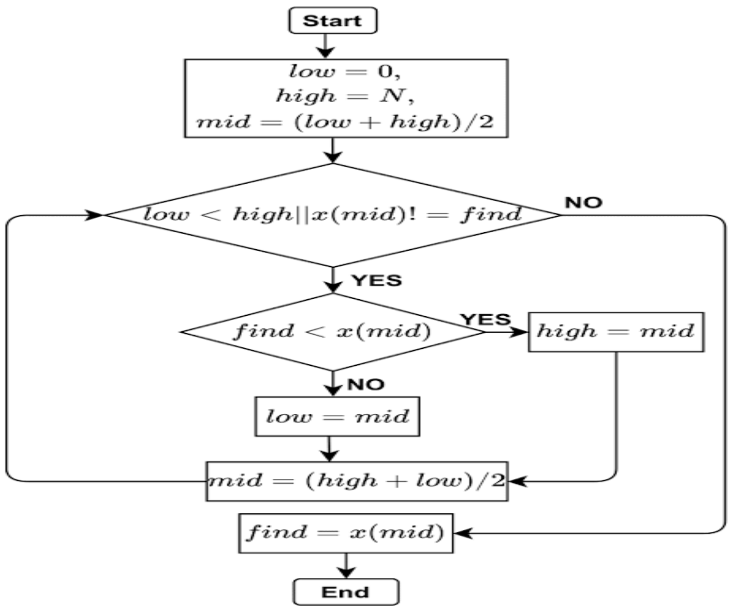
Best Case: O(1) - Similar to the iterative approach, this occurs when the target is the middle element.

Worst Case: O(log n) - The search space is halved with each recursive call, leading to a logarithmic number of calls.

Average Case: O(log n) - The average performance remains logarithmic as each recursive call effectively halves the search space

Space Complexity:
O(log n) → Due to recursive function calls stored in the call stack

Flow Chart :



Conclusion of Binary Search
Binary search is a highly efficient algorithm for searching in a **sorted** array or list. It operates by repeatedly dividing the search interval in half, which allows it to achieve a **time complexity of O(log n)** in the worst and average cases. This makes it significantly faster than linear search for large datasets. However, binary search requires the input data to be sorted beforehand, which can add an additional O(n log n) preprocessing cost if the data is unsorted. The **space complexity** of binary search is **O(1)** for the iterative implementation and **O(log n)** for the recursive implementation due to the call stack. Binary search is ideal for scenarios where the data is static or changes infrequently, and where fast search operations are critical.

SORTING ALGORITHMS

1: Insertion Sort

Introduction
Insertion sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It works by iteratively taking one element from the unsorted portion of the array and inserting it into its correct position in the sorted portion. This process is similar to how one might sort a hand of playing cards: picking one card at a time and placing it in the right order among the already sorted cards. Insertion sort is efficient for small datasets or nearly sorted datasets, where it performs well due to its adaptive nature. Although it has a **time complexity of O(n^2)** in the worst case, it is often preferred for its simplicity, low overhead, and in-place sorting capability (requiring only O(1) additional space). Insertion sort has been used for decades and remains a fundamental algorithm in computer science, particularly for educational purposes and in scenarios where data is already partially sorted.

Approach for Insertion Sort
The approach to understanding insertion sort involves analyzing its iterative and recursive implementations, as well as its time and space complexities. Insertion sort is a straightforward sorting algorithm that builds the final sorted array one element at a time. It works by dividing the array into two parts: a **sorted portion** and an **unsorted portion**. The algorithm iteratively picks an element from the unsorted portion and inserts it into its correct position in the sorted portion. This process continues until the entire array is sorted.

While insertion sort is typically implemented iteratively, it can also be implemented recursively. The recursive approach follows the same logic but uses function calls to handle the insertion process. Both implementations share the same core idea but differ in their structure and space complexity due to the use of recursion.

STEPS OF INSERTION SORT

1. Divide: The array is conceptually divided into a sorted portion (initially containing the first element) and an unsorted portion (containing the remaining elements).

2. Conquer: Iteratively or recursively, pick an element from the unsorted portion and insert it into the correct position in the sorted portion.

3. Combine: As each element is inserted into the sorted portion, the sorted portion grows, and the unsorted portion shrinks until the entire array is sorted.

Iterative Insertion Sort:

- The iterative approach uses nested loops. The outer loop iterates over the unsorted portion, while the inner loop shifts elements in the sorted portion to make space for the new element.

Time Complexity :
- Best Case: O(n) (when the array is already sorted).
- Worst Case: O(n^2) (when the array is sorted in reverse order).
- Average Case: O(n^2).

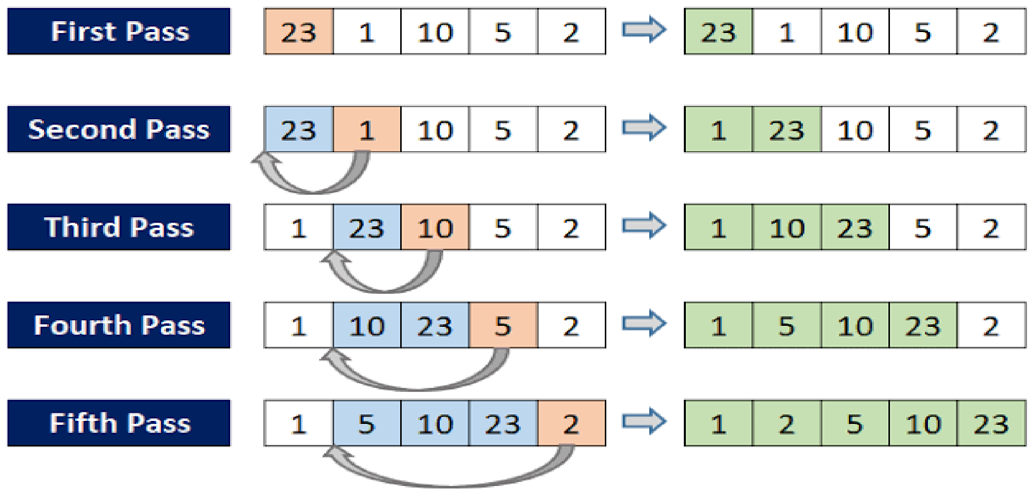
Space Complexity : O(1) (in-place sorting).

Recursive Insertion Sort:
- The recursive approach replaces the outer loop with recursive function calls. The base case handles the scenario when the array is fully sorted.

Time Complexity:
- Best Case: O(n).
- Worst Case: O(n^2).
- Average Case: O(n^2).

Space Complexity: O(n) due to the recursion call stack.

Analysis of Inserting Sort Algorithm



illustrates the insertion sort algorithm applied to an array step by step. The array starts as [23, 1, 10, 5, 2], and the sorting process is broken down into five passes.

In the first pass, the first element (23) is considered sorted. In the second pass, the algorithm compares 1 with 23 and swaps them, resulting in [1, 23, 10, 5, 2]. In the third pass, 10 is compared with 23, and it is placed correctly before 23, producing [1, 10, 23, 5, 2]. The fourth pass places 5 in the correct position by shifting 23 and 10 forward, yielding [1, 5, 10, 23, 2]. Finally, in the fifth pass, 2 is inserted at the correct position by shifting all larger elements forward, resulting in the sorted array [1, 2, 5, 10, 23].

This visual representation clearly demonstrates how insertion sort iteratively places each element in its correct position by shifting larger elements to the right, making it an efficient algorithm for small or nearly sorted datasets.

Insertion Sort is another classic sorting algorithm that works by building a sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as Merge Sort, Quick Sort, or Heap Sort. However, Insertion Sort has several advantages:

1. Simple Implementation: It is easy to understand and implement.
2. Efficient for Small Data Sets : It is efficient for small data sets or data sets that are already mostly sorted.
3. Adaptive : If the input array is already partially sorted, Insertion Sort can be very efficient.
4. In-Place Sorting : It requires only a constant amount of additional memory space.

How Insertion Sort Works

Insertion Sort works by iterating through the array and inserting each element into its correct position within the sorted portion of the array. Here's a step-by-step breakdown:

1. Start with the second element : Assume the first element is already sorted.
2. Compare with the previous element: For each subsequent element, compare it with the elements in the sorted portion of the array.
3. Shift elements : If the current element is smaller than the previous element, shift the previous element to the right.
4. Insert the element: Insert the current element into its correct position in the sorted portion.
5. Repeat : Continue this process until the entire array is sorted.

Pseudo Code for Insertion Sort

```
Insertion-Sort(A)
  for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

Explanation of the Pseudo Code

A : The array to be sorted.
j : The index of the current element to be inserted into the sorted portion.
key : The value of the current element to be inserted.
i : The index used to traverse the sorted portion of the array.

COMPLEXITY ANALYSIS

1. Time Complexity Analysis

Best Case (Already Sorted) -
If the array is already sorted, each element only needs to be compared once.
The inner loop runs only once for each element.

Worst Case (Reverse Sorted) -
In the worst case, each element must be compared with all previous elements and shifted accordingly.

The number of comparisons and shifts for each element:

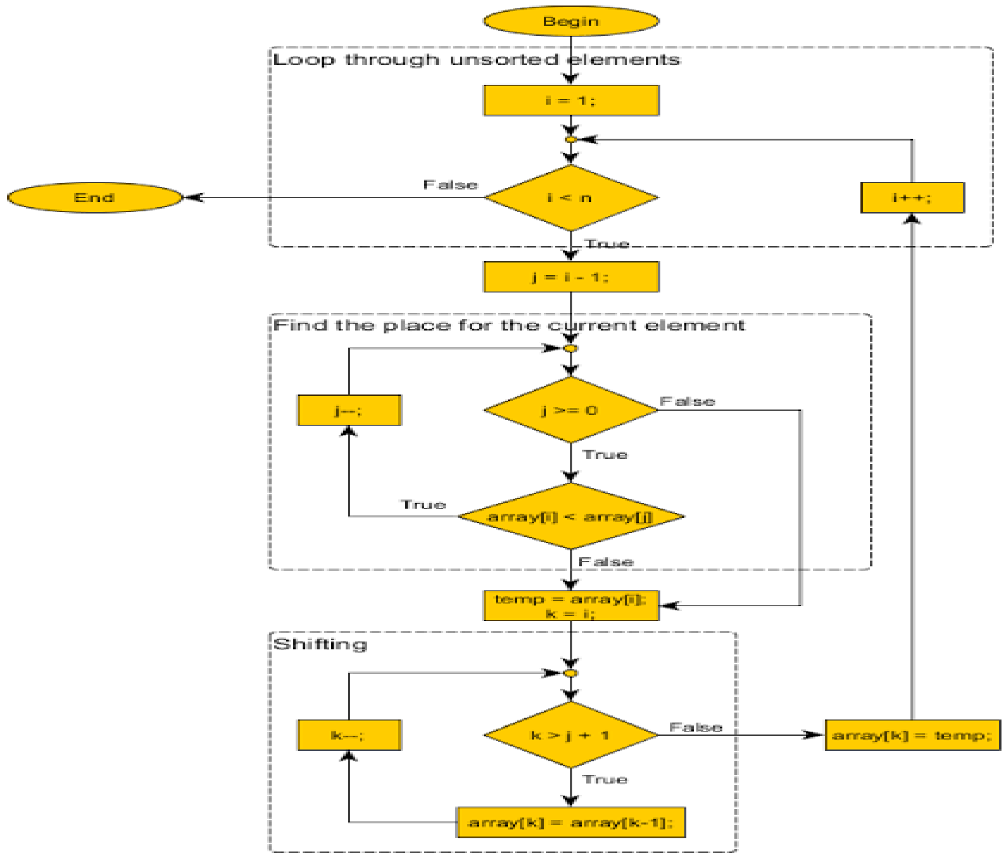
1st element → 0 comparisons
2nd element → 1 comparison
3rd element → 2 comparisons

Total comparisons:
1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2} \approx O(n^2)

Average Case -
On average, an element is compared with half of the previous elements.
The expected number of comparisons is still proportional .

2. Space Complexity -

Insertion Sort is an in-place sorting algorithm, meaning it does not require extra space for sorting.
It only uses a few additional variables, so the space complexity is .



2. Selection Sort

Introduction
Selection Sort is a simple comparison-based sorting algorithm that divides the array into two parts: a sorted portion and an unsorted portion. It repeatedly finds the smallest (or largest) element from the unsorted portion and swaps it with the first element of the unsorted portion. This process continues until the entire array is sorted.

Selection Sort is easy to implement but is not as efficient as more advanced sorting algorithms like Merge Sort or Quick Sort. It has a time complexity of in all cases, making it inefficient for large datasets. However, it is useful for small datasets due to its simplicity and in-place sorting capability (requiring only additional space).

Approach for Selection Sort
The approach to understanding selection sort involves analyzing its iterative and recursive implementations, as well as its time and space complexities. Selection sort is a straightforward sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and places it in its correct position in the sorted portion. The algorithm works by dividing the array into two parts: a sorted portion and an unsorted portion. It then finds the minimum element from the unsorted portion and swaps it with the first element of the unsorted portion. This process continues until the entire array is sorted.

While selection sort is typically implemented iteratively, it can also be implemented recursively. The recursive approach follows the same logic but uses function calls to find and place the minimum element. Both implementations share the same core idea but differ in their structure and space complexity due to recursion.

Steps of Selection Sort

1. Divide: The array is conceptually divided into a sorted portion (initially empty) and an unsorted portion (containing all elements).

2. Conquer: Iteratively or recursively, find the minimum element from the unsorted portion and swap it with the first element of the unsorted portion.

3. Combine: As each element is placed in the correct position, the sorted portion grows, and the unsorted portion shrinks until the entire array is sorted.

Iterative Selection Sort
The iterative approach uses a loop to scan the unsorted portion of the array, find the minimum element, and swap it with the first unsorted element.

Time Complexity:

Best Case: $O(n^2)$ (even if the array is already sorted).
Worst Case: $O(n^2)$ (when the array is sorted in reverse order).
Average Case: $O(n^2)$.

Space Complexity: $O(1)$ (in-place sorting).

Recursive Selection Sort
The recursive approach replaces the outer loop with recursive function calls. The base case is when the array is fully sorted.

Time Complexity:

Best Case: $O(n^2)$.
Worst Case: $O(n^2)$.
Average Case: $O(n^2)$.

Space Complexity: $O(n)$ due to the recursion call stack.

Analysis of Selection Sort
The Selection Sort algorithm follows a greedy approach by repeatedly selecting the smallest element and placing it in its correct position. It works as follows:

1. Divide: The array is conceptually divided into a sorted portion and an unsorted portion. Initially, the sorted portion is empty.

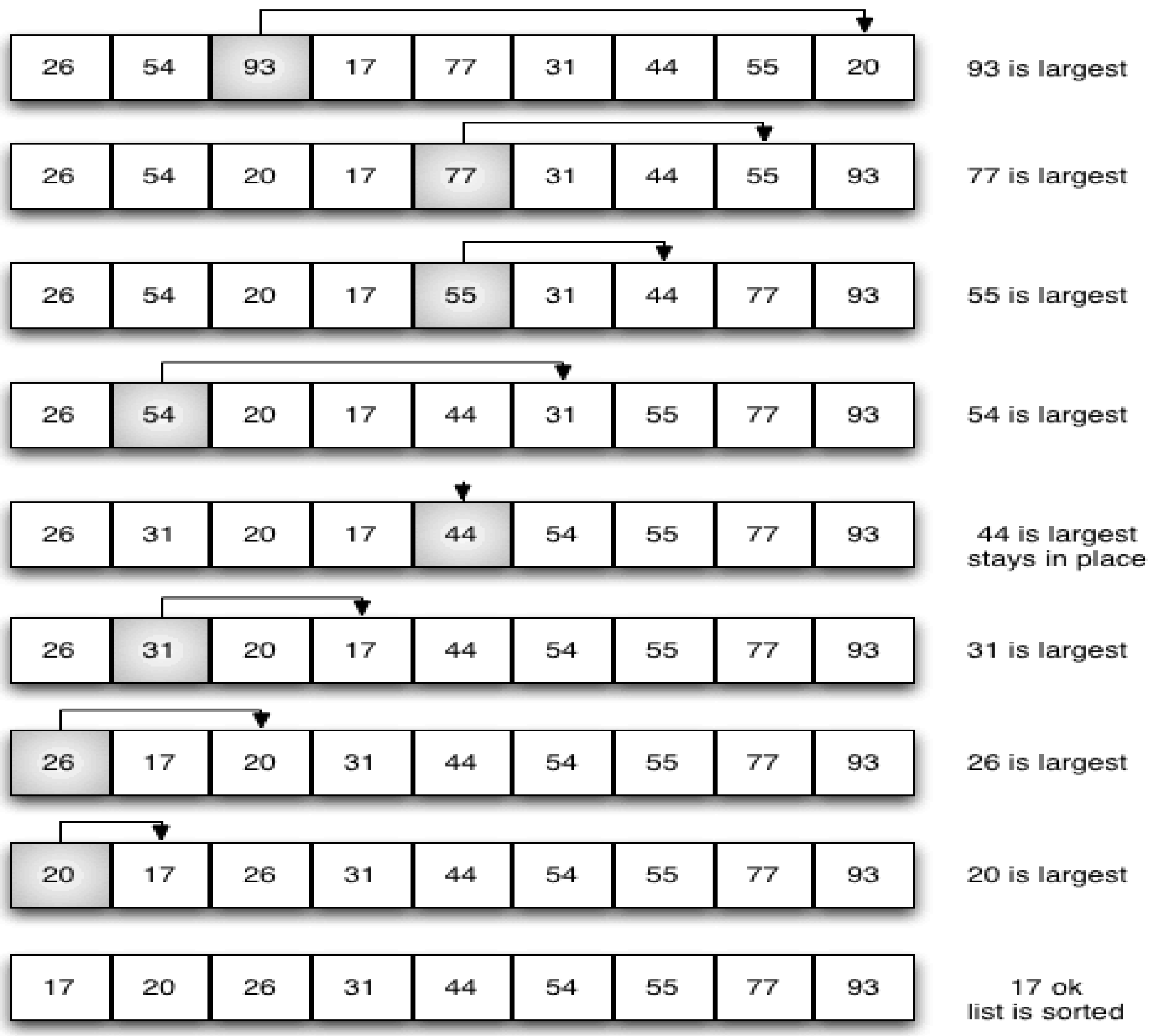
2. Conquer: Find the smallest element from the unsorted portion and swap it with the first element of the unsorted portion.

3. Combine: The sorted portion grows while the unsorted portion shrinks, and the process continues until the entire array is sorted.

Selection Sort is always implemented iteratively and does not have a recursive approach due to its straightforward swapping mechanism.

Steps of Selection Sort

1. Start from the first element and assume it is the minimum.
2. Scan the unsorted portion to find the actual minimum element.
3. Swap the minimum element with the first element of the unsorted portion.
4. Move to the next element and repeat the process until the entire array is sorted.



The provided content appears to be a step-by-step demonstration of a sorting algorithm, likely the Selection Sort. This algorithm works by repeatedly selecting the largest (or smallest) element from the unsorted portion of the list and swapping it with the element at the end of the unsorted portion. Here's a detailed explanation of the process:

1. Initial List: The process begins with the list `[26, 54, 93, 17, 77, 31, 44, 55, 20, 93]`. The algorithm identifies the largest element in the list, which is `93`, and swaps it with the last element, also `93`. Since they are the same, the list remains unchanged.

2. Second Iteration: The list is now `[26, 54, 20, 17, 77, 31, 44, 55, 93, 77]`. The largest element in the unsorted portion is `93`, which is already in the correct position. The next largest element is `77`, which is swapped with the last unsorted element, `77`, resulting in no change.

3. Third Iteration : The list becomes `[26, 54, 20, 17, 55, 31, 44, 77, 93, 55]`. The largest element in the unsorted portion is `77`, which is already correctly placed. The next largest element is `55`, which is swapped with the last unsorted element, `55`, again resulting in no change.

4. Fourth Iteration : The list is now `[26, 54, 20, 17, 44, 31, 55, 77, 93, 54]`. The largest element in the unsorted portion is `77`, which is correctly placed. The next largest element is `55`, which is already in place. The next largest element is `54`, which is swapped with the last unsorted element, `54`, resulting in no change.

5. Fifth Iteration : The list becomes `[26, 31, 20, 17, 44, 54, 55, 77, 93, 44]`. The largest element in the unsorted portion is `77`, which is correctly placed. The next largest element is `55`, which is already in place. The next largest element is `54`, which is also correctly placed. The next largest element is `44`, which stays in place as it is already at the correct position.

6. Subsequent Iterations : The process continues, with the algorithm repeatedly identifying the largest element in the unsorted portion and swapping it into its correct position. Each step reduces the size of the unsorted portion until the entire list is sorted.

7. Final Sorted List : The final sorted list is `[17, 20, 26, 31, 44, 54, 55, 77, 93, 93]`. The algorithm has successfully sorted the list in ascending order by repeatedly selecting and placing the largest elements in their correct positions.

This step-by-step process illustrates the Selection Sort algorithm's methodical approach to sorting a list by iteratively finding and placing the largest (or smallest) elements in their correct positions.

How Selection Sort Works

Selection Sort works by dividing the array into a sorted and an unsorted portion. It repeatedly selects the smallest (or largest) element from the unsorted portion and swaps it with the first unsorted element. Here’s a step-by-step breakdown:

1. Find the minimum element: Start with the first element and find the smallest element in the unsorted portion of the array.

2. Swap it with the first unsorted element: Swap the smallest element with the first unsorted element, thus extending the sorted portion.

3. Move to the next unsorted element: Repeat the process for the remaining unsorted portion of the array.

4. Repeat until the array is sorted: Continue until the entire array is sorted.

Pseudo Code for Selection Sort

```
Selection-Sort(A)
  for i = 1 to A.length - 1
    minIndex = i
    for j = i + 1 to A.length
      if A[j] < A[minIndex]
        minIndex = j
    swap(A[i], A[minIndex])
```

Explanation of the Pseudo Code

A: The array to be sorted.

i: The index of the first element in the unsorted portion.

minIndex: The index of the smallest element in the unsorted portion.
j: The index used to search for the smallest element.

The algorithm finds the smallest element in the unsorted portion and swaps it with the first unsorted element.

Complexity Analysis

1. Time Complexity Analysis

Best Case (Already Sorted) -
Even if the array is already sorted, Selection Sort still scans the entire unsorted portion to find the minimum element. The number of comparisons remains the same as in the worst case.

Time Complexity: $O(n^2)$

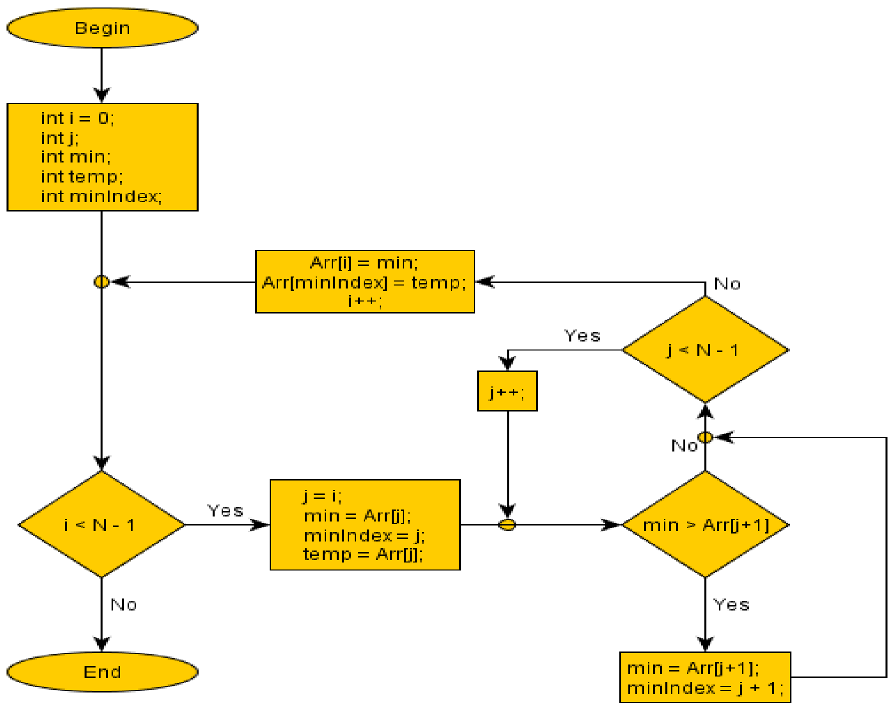
Worst Case (Reverse Sorted) -
The algorithm still has to scan the entire unsorted portion to find the minimum element.

Number of comparisons:
1st pass: (n - 1) comparisons
2nd pass: (n - 2) comparisons
(n-1)th pass: 1 comparison

Total comparisons:
(n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} \approx O(n^2)

Average Case -
Since the number of comparisons is always the same regardless of the order,
Time Complexity: $O(n^2)$

2. Space Complexity -
Selection Sort is an in-place sorting algorithm, meaning it does not require additional space for sorting.
It only uses a few extra variables for swapping.
Space Complexity: O(1)



3 Bubble Sort

Introduction:
Bubble Sort is a simple comparison-based sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. The largest (or smallest) elements gradually "bubble" to their correct position with each pass through the array.

Bubble Sort is easy to implement but is not efficient for large datasets due to its $O(n^2)$ time complexity. However, it is useful for small datasets and teaching purposes due to its simplicity and stable sorting behavior. It is an in-place sorting algorithm, meaning it does not require extra memory beyond the input array.

Approach for Bubble Sort

The approach to understanding Bubble Sort involves analyzing its iterative and recursive implementations, as well as its time and space complexities. Bubble Sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. The process continues until no swaps are needed, meaning the array is fully sorted.

Bubble Sort works by making multiple passes through the array. In each pass, the largest (or smallest) element moves (or "bubbles") to its correct position. This process gradually reduces the unsorted portion of the array while growing the sorted portion.

Although Bubble Sort is usually implemented iteratively, it can also be implemented recursively. The recursive approach follows the same logic but relies on function calls instead of loops. Both implementations share the same time complexity, but recursion introduces extra space usage due to the function call stack.

Steps of Bubble Sort

1. Divide: Conceptually divide the array into a sorted portion (initially empty) and an unsorted portion (containing all elements).

2. Conquer: Repeatedly compare adjacent elements in the unsorted portion. If they are in the wrong order, swap them. This process is repeated, causing the largest (or smallest) element to "bubble up" to its correct position.

3. Combine: As elements are placed in their correct positions, the sorted portion grows, and the unsorted portion shrinks until the entire array is sorted.

Iterative Bubble Sort
The iterative approach uses nested loops to repeatedly traverse the array, swapping adjacent elements when necessary. If no swaps occur in a full pass, the algorithm stops early.

Time Complexity:

Best Case: $O(n)$ (if the array is already sorted and an optimized version is used)
Worst Case: $O(n^2)$ (if the array is sorted in reverse order)
Average Case: $O(n^2)$

Space Complexity:
$O(1)$ (in-place sorting, requires no extra memory)

Recursive Bubble Sort

The recursive approach replaces the outer loop with recursive function calls. The base case occurs when no swaps are needed, meaning the array is sorted.

Time Complexity:

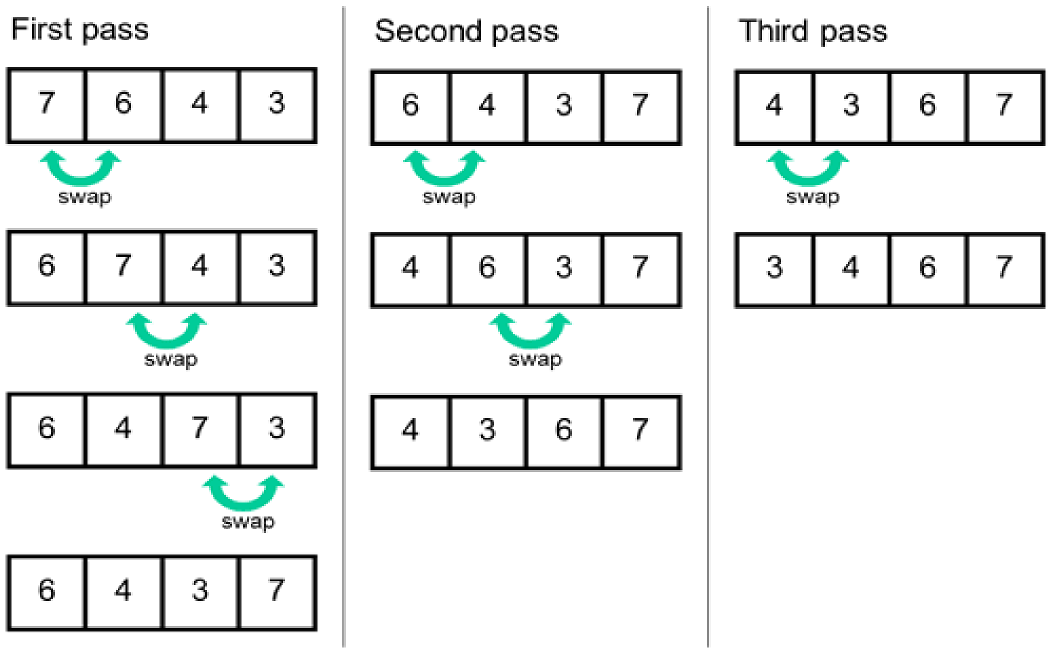
Best Case: $O(n^2)$

Worst Case: O(n^2)
Average Case: O(n^2)

Space Complexity:
O(n) (due to the recursion call stack)

Analysis of Bubble Sort

Bubble Sort follows a brute-force approach, repeatedly comparing and swapping adjacent elements if they are in the wrong order. The largest (or smallest) element "bubbles" to its correct position in each pass.



Bubble sort or a similar comparison-based sorting method. The algorithm processes a list of numbers through multiple passes, comparing adjacent elements and swapping them if they are in the wrong order.

In the first pass, the list starts as [7, 6, 4, 3]. After the first swap, it becomes [6, 7, 4, 3], and after the second swap, it changes to [6, 4, 7, 3]. The third swap results in [6, 4, 3, 7]. This process continues through subsequent passes, with the algorithm repeatedly swapping adjacent elements to move the largest unsorted element to its correct position at the end of the list.

By the third pass, the list is [4, 3, 6, 7], indicating that the largest elements are gradually being sorted towards the end. The algorithm continues to make swaps until the list is fully sorted. The final sequence shown is [6, 4, 3, 7], suggesting that the sorting process is still ongoing and may require additional passes to achieve the fully sorted order.

This step-by-step approach is characteristic of bubble sort, where each pass through the list reduces the number of unsorted elements by one, eventually leading to a sorted list.

How Bubble Sort Works

Bubble Sort works by dividing the array into a sorted and an unsorted portion. It repeatedly compares adjacent elements and swaps them if necessary until the entire array is sorted.

1. Compare adjacent elements: Start with the first two elements and swap them if they are in the wrong order.

2. Move to the next pair: Continue this process for all elements, ensuring the largest element "bubbles" to its correct position at the end.
3. Repeat for remaining elements: Ignore the last sorted element and repeat the process until no swaps are needed.

Steps of Bubble Sort

1. Start from the first element and compare it with the next element.
2. If they are in the wrong order, swap them.
3. Move to the next pair of elements and repeat the process.
4. The largest element moves to its correct position at the end.
5. Repeat the process for the remaining elements until the entire array is sorted.

Pseudo Code for Bubble Sort

```
Bubble-Sort(A)
  for i = 0 to A.length - 1
    swapped = false
    for j = 0 to A.length - i - 1
      if A[j] > A[j + 1]
        swap(A[j], A[j + 1])
        swapped = true
    if swapped == false
      break
```

Explanation of the Pseudo Code

A: The array to be sorted.
i: The current pass (outer loop).
j: The index used to compare adjacent elements (inner loop).
swapped: A flag to check if any swaps occurred (used for optimization).
The outer loop tracks passes to gradually place the largest elements at the end.
The inner loop performs swaps for adjacent elements.
If no swaps occur in a pass, the array is already sorted, and the algorithm terminates early (optimized version).

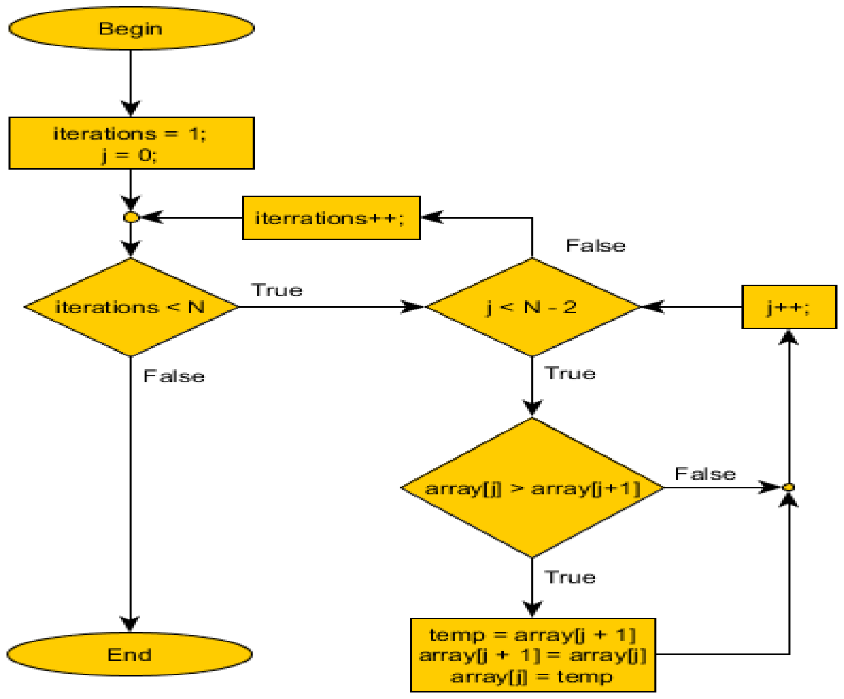
COMPLEXITY ANALYSIS

1. Time Complexity Analysis
Best Case (Already Sorted Array)
In the optimized version, if no swaps occur in the first pass, the algorithm terminates early.
Time Complexity: $O(n)$

Worst Case (Reverse Sorted Array)
The algorithm performs $(n-1) + (n-2) + ... + 1 = O(n^2)$ comparisons.
Time Complexity: $O(n^2)$

Average Case (Random Order)
The number of swaps and comparisons remains $O(n^2)$ on average.
Time Complexity: $O(n^2)$

2. Space Complexity
Bubble Sort is an in-place sorting algorithm that does not require extra memory beyond a few variables.
Space Complexity: $O(1)$



4 Merge Sort :

Introduction:
Merge sort is a divide and conquer technique of sorting the element and basically works on this technique. Merge is one of the most efficient sorting algorithms. This algorithm was invented by John von Neumann in 1945. Merge sort algorithm divides the whole set of numbers into the sub lists or we can say that by this technique we can divide the list of array into the sub lists. These algorithms typically follow a Divide-and-conquer approach they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. Merge sort is as important in the history of sorting as sorting in the history of computing. A detailed description of bottom-up merge sort, together with a timing analysis, appeared in a report by Goldstine and Neumann as early 1948.

Approach:
The approach is to find out the variation between the complexities of Recursive and Non-recursive are such that we have study both Recursive and Non-recursive Merge Sort Algorithm. There are many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of sub problems.
Conquer the sub problem by solving them recursively.
If the sub problem size is small enough, however, just solve the sub problems in a straight forward manner.
Combine the solutions to the subproblems into the solution for the original problem.
The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively,
it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

Analysis of Variation of Merge Sort

When the sequence to be sorted has length 1, in which case there is no work to be done, and since every sequence of length 1 is already in sorted order. The key operation of merge sort algorithm is the merging of two sorted sequence in 'combine' step. To perform the merging, we use an Auxiliary procedure (Straightforward manner). When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. Merge (A, p, q, r) it is easy to imagine a MERGE procedure that takes time ɵ (n) where n=r-p+1 is the number of elements being merged.

The following pseudo code implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a sentinel card, which contains a special value that we use to simplify our code. Here, we use 1 as the sentinel value, so that whenever a card with1is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly r - p C+1 cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

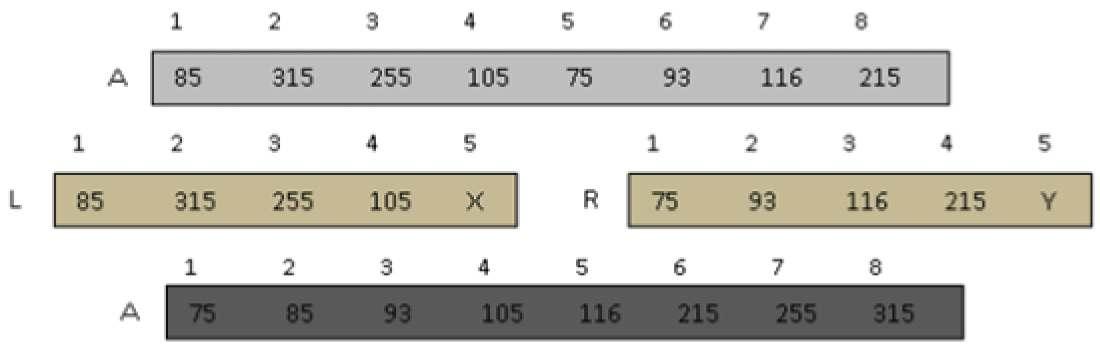
Merge(A, p, q, r)

```
1. n1 = q - p + 1
2. n2 = r - q
3. Create arrays L[1...n1+1] and R[1...n2+1]
4. for i = 1 to n1
     L[i] = A[p + i - 1]
5. for j = 1 to n2
     R[j] = A[q + j]
6. L[n1+1] = ∞  (Sentinel value)
7. R[n2+1] = ∞  (Sentinel value)
8. i = 1, j = 1
9. for k = p to r
     if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
     else
        A[k] = R[j]
        j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n1 of the subarray A[p…q], and line 2 computes the length n2 of the subarray A[q+1….r]. We create arrays L and R ("left" and "right"), of lengths n1 + 1 and n2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel.

The for loop of lines 4 copies the subarray A[p..q] into L[1…n1], and the for loop of lines 5 copies the subarray A[q + 1…r] into R[1..n2]. Lines 6 put the sentinels at the ends of the arrays L and R. At the start of each iteration of the for loop of line 8, the subarray A[p…k-1] contains the k - p smallest elements of L[1….n1+1] and R[1…..n2+1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the for loop of line 8, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.



The operation of lines 7-8 in the call MERGE(A, 1, 4, 8), when the subarray A[1 …8] contains the sequence (2,1,5,7,1,2,3,6,). After copying and inserting sentinels, the array L contains (2,1,5,7,X), and the array R contains (1,2, 3, 6,Y) . Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in AOE9 : : 16_, along with the two sentinels.

Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. (a)–(h) The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A,p,r) sorts the element in the sub array A[p…r]. If p≤r, the sub array has at most one element and is therefore already sorted. Otherwise, the divide steps simply compute an index q that partitions A[p….r] into two sub array A[p….q] & A[q+1….r] both containing (n/2) elements and n is the total number of elements in array. we let T .n/ be the running time on a problem of size n. If the problem size is small enough, say n≤c for some constant c, the straightforward solution takes constant time, which we write as ɵ(1) Suppose that our division of the problem yields a subproblems, each of which is 1/b the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which a ≠ b.) It takes time T (n/b) to solve one sub problem of size n=b, and so it takes time aT (n/b)

to solve a of them. If we take D(n) time to divide the problem into subproblems and C(n) time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

T (n) = {ɵ (1) if n ≤ c
{aT (n/b) +D(n) + C(n) otherwise

MergeSort(A, p, r)

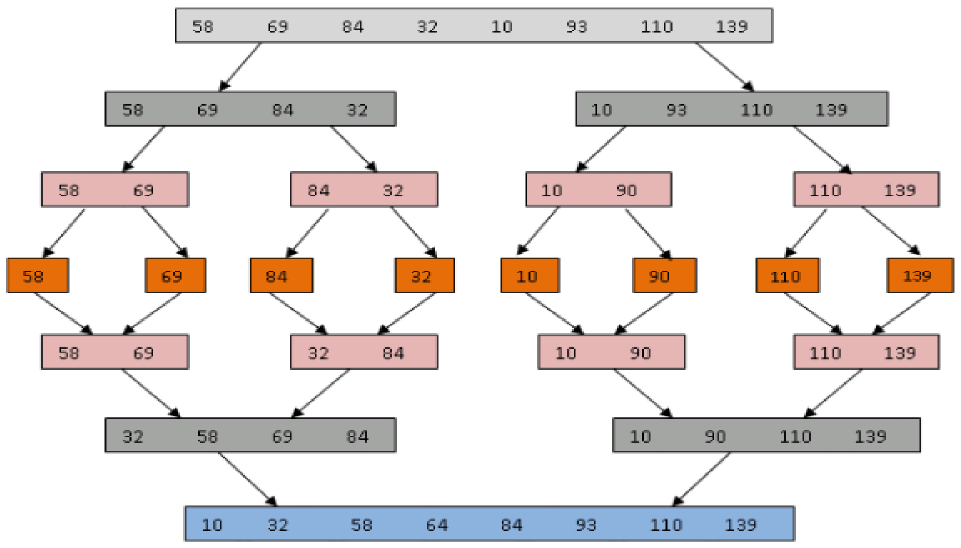
```
1. if p < r
2.    q = ⌊(p + r) / 2⌋
3.    MergeSort(A, p, q)
4.    MergeSort(A, q + 1, r)
5.    Merge(A, p, q, r)
```

In detail, the Merge Sort (A, p, r) procedure work as follows, divides the array into two arrays length of n/2 elements. This function works recursively again and again to further divide the n/2 sub array into length of n/4 elements of sub-array and calls recursively until only one element remains in the array. Merge sort (A, p, q, r) works as follows, Line 1 computes the length n1 of the subarray A[p…q], and line 2 computes the length n2 of the subarray A[q+1….r].

We create arrays L and R ("left" and "right"), of lengths n1 + 1 and n2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel. The for loop of lines 4 copies the subarray A[p..q] into L[1…n1], and the for loop of lines 5 copies the subarray A[q + 1…r] into R[1..n2].

Lines 6 put the sentinels at the ends of the arrays L and R. At the start of each iteration of the for loop of line 8, the subarray A[p…k-1] contains the k - p smallest elements of L[1….n1+1] and R[1…..n2+1], in sorted order.

Consider partitioning for a list of 8 elements:



Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A. When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm. A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let T (n) be the running time on a problem of size n. If the problem size is small enough, say n ≤ c for some constant c, the straightforward solution takes constant time, which we write as ɵ (1). Suppose that our division of the problem yields a sub problem, each of which is 1/b the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which a ≠ b.) If we take D (n) time to divide the problem into subproblems and C (n) time to combine the solutions to the sub problem of the giving array in sorted manner form. We have a reason as follows to set up the recurrence for T (n), the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have n > 1 elements, we break down the running time as follows. n the divide step just computes the middle of the subarray, which takes constant time. Thus, D (n) = ɵ(1). We recursively solve two subproblems, each of size n/2, which contributes2T (n/2) to the running time. We have already noted that the MERGE procedure on an n-element subarray takes time ɵ (n), and so C (n) = ɵ (n). The below Figure shows the recursive method of solving an array to be a sorted manner, and show how the problem becomes complex and run time complexity becomes more.

At a high level, the implementation involves two activities, partitioning and merging, each represented by a corresponding function. The number of partitioning steps equals the number of merge steps, partitioning taking place during the recursive descent and merging during the recursive ascent as the calls back out.

All the element comparisons take place during the merge phase. Logically, we may consider this as if the algorithm re-merged each level before proceeding to the next:

So merging the sub lists involves (log n) passes. On each pass, each list element is used in (at most) one comparison, so the number of element comparisons per pass is N. Hence, the number of comparisons for Merge Sort is Θ (n log n).

Merge Sort comes very close to the theoretical optimum number of comparisons. A closer analysis shows that for a list of N elements, the average number of element comparisons using Merge Sort is actually:

ɵ(N nog n) -1.1583N +1

Recall that the theoretical minimum is:

N log N -1.44N +ɵ(1)

For a linked list, Merge Sort is the sorting algorithm of choice, providing nearly optimal comparisons and requiring NO element assignments (although there is a considerable amount of pointer manipulation), and requiring NO significant additional storage.

For a contiguous list, Merge Sort would require either using Θ(N) additional storage for the sublists or using a considerably complex algorithm to achieve the merge with a small amount of additional storage.

The divide step just computes the middle of the subarray, which takes constant time. Thus, D (n) = ɵ (1). We recursively solve two subproblems, each of size n=2, which contributes 2T (n/2) to the running time. We have already noted that the MERGE procedure on an n-element subarray takes time ɵ (n), and so c(n) = ɵ (n). When we add the functions D (n) and C(n) for the merge sort analysis, we are adding a function that is ɵ (n) and a function that is ɵ (1) This sum is a linear function of n, that is, ɵ(n). Adding it to the 2T (n/2) term from the "conquer".

COMPLEXITY ANALYSIS OF MERGE SORT

1. Time Complexity Analysis

Merge Sort follows the Divide and Conquer approach, repeatedly dividing the array into two halves and merging them back in sorted order.

Best Case (Already Sorted Array)

Even if the array is sorted, Merge Sort still divides it recursively and merges it back.

The number of comparisons remains the same as in the worst case.

Time Complexity: O(n log n)

Worst Case (Reverse Sorted Array)

The algorithm still follows the same recursive divisions and merging process.

Every element comparison and merge operation remains the same.

Time Complexity: O(n log n)

Average Case (Random Order)

Regardless of the initial order, Merge Sort always divides the array into halves and merges them.

The number of comparisons and merges remains O(n log n).

Time Complexity: O(n log n)

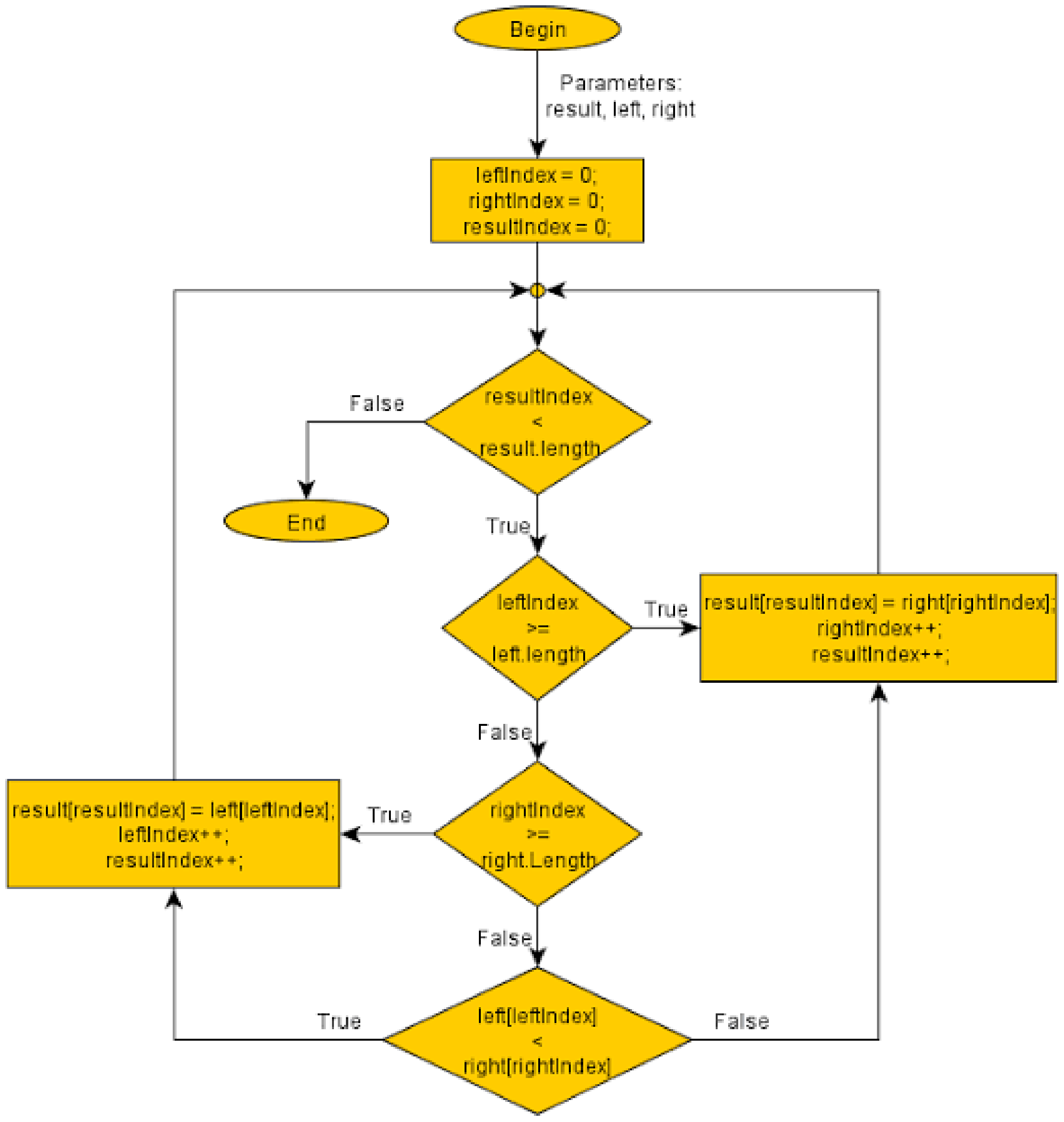
2. Space Complexity Analysis

Merge Sort requires additional memory for temporary arrays during merging.

It uses O(n) extra space to store the left and right subarrays.

Since recursion depth goes up to log n, the stack space used is O(log n).

Overall Space Complexity: O(n) (not in-place).



5 Quick Sort :

Introduction to Quick Sort
Quick Sort is a highly efficient Divide and Conquer sorting algorithm developed by Tony Hoare in 1959. It is one of the most widely used sorting algorithms due to its superior performance in the average case and its in-place sorting capability.

Quick Sort works by selecting a pivot element, partitioning the array into two subarrays (elements smaller than the pivot on the left and elements greater than the pivot on the right), and recursively applying the same process to the subarrays. This process continues until the array is completely sorted.

The algorithm follows these three steps:

1. Divide: Choose a pivot and partition the array into two halves.

2. Conquer: Recursively sort the left and right halves.

3. Combine: Since sorting is done in-place, no additional merging is required.

Unlike Merge Sort, which requires additional space for merging, Quick Sort sorts the array in-place, making it more space-efficient. However, its worst-case performance occurs when the pivot is poorly chosen, leading to an unbalanced partition. Despite this, with proper pivot selection strategies (like randomized quick sort or median-of-three), Quick Sort performs efficiently in most cases.

Quick Sort plays a significant role in computing history and is still widely used in various applications, including database indexing and large-scale data processing, due to its efficiency and adaptability.

Approach

Quick Sort follows the Divide and Conquer paradigm to efficiently sort an array. The algorithm works by selecting a pivot element and partitioning the array into two halves: one with elements smaller than the pivot and the other with elements greater than the pivot. The process is then applied recursively to both halves until the entire array is sorted.

Steps in Quick Sort:

1. Divide:

Choose a pivot element from the array (it can be the first element, last element, median, or a random element).

Partition the array into two subarrays:

1. Elements less than or equal to the pivot.
2. Elements greater than the pivot.

2. Conquer:

Recursively apply Quick Sort to the left and right subarrays.

3. Combine:

Since sorting happens in-place, no explicit merging is needed. The partitioned elements naturally combine into a sorted order.

RECURSIVE AND NON-RECURSIVE QUICK SORT

Recursive Quick Sort:

In the recursive approach, Quick Sort calls itself to sort the left and right partitions until the base case (single-element or empty subarray) is reached.

The recursion depth depends on how well the pivot is chosen.

Worst case: When the pivot always results in unbalanced partitions ($O(n^2)$).

Best and average cases: When partitions are balanced, leading to O(n log n) complexity.

Non-Recursive Quick Sort (Iterative Quick Sort):

The recursive calls are replaced with an explicit stack to manage subarray partitions.

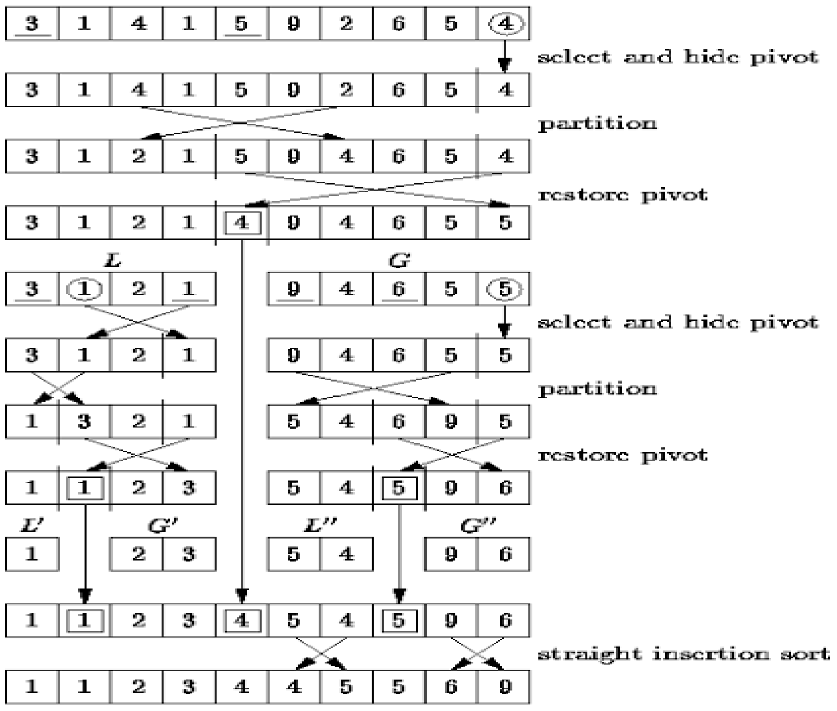
Instead of using function calls, the algorithm pushes and pops subarrays onto a stack, simulating the recursion manually.

This method avoids stack overflow issues for large inputs and is useful in memory-constrained environments.

Analysis of Bubble Sort

Quick Sort follows a divide-and-conquer approach. It selects a pivot element and partitions the array such that elements smaller than the pivot move to its left and elements greater than the pivot move to its right.

The process is recursively applied to the left and right subarrays until the entire array is sorted.



Step 1: Choosing the Pivot
The algorithm begins with the unsorted array [3, 1, 4, 1, 5, 9, 2, 6, 5, 4], and the last element 4 is chosen as the pivot. The pivot is temporarily hidden to facilitate partitioning.

Step 2: Partitioning
Elements are rearranged such that values smaller than the pivot move to the left and values greater move to the right. The partitioned array after swapping becomes [3, 1, 2, 1, 4, 9, 5, 6, 5, 4].

Step 3: Restoring the Pivot
After partitioning, the pivot 4 is restored to its correct position, resulting in [3, 1, 2, 1, 4, 9, 5, 6, 5]. The array is now split into two subarrays: L (left subarray) [3, 1, 2, 1] and G (right subarray) [9, 6, 5, 5].

Step 4: Recursively Sorting Left Subarray
The new pivot 1 is chosen.
Partitioning results in [1, 1, 3, 2].
The pivot is restored to its correct position, giving [1, 1, 2, 3].
This small subarray is now sorted.

Step 5: Recursively Sorting Right Subarray
The pivot 5 is selected.
Partitioning moves smaller elements before 5, resulting in [4, 5, 5, 9, 6].
After restoring the pivot, the right subarray becomes [4, 5, 5, 6, 9]

Step 6: Final Sorting
After recursively sorting all partitions, the final sorted array is [1, 1, 2, 3, 4, 5, 5, 6, 9].

At the last step, a straight insertion sort is applied as a final refinement, ensuring complete order. Quick Sort efficiently partitions and sorts elements in place using the divide-and-conquer approach.

How Quick Sort Works
Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a pivot element and partitioning the array into two subarrays—one with elements smaller than the pivot and the other with elements greater than the pivot. This process is recursively applied to each subarray until the entire array is sorted.

Steps of Quick Sort

1. Select a Pivot: Choose a pivot element (typically the last, first, or a random element).
2. Partition the Array: Rearrange elements so that all elements smaller than the pivot are on the left and larger elements are on the right.
3. Recursively Sort Subarrays: Apply Quick Sort to the left and right subarrays.
4. Combine the Sorted Subarrays: Once sorting is complete, the entire array is in order.

Pseudo Code for Quick Sort

```
QuickSort(A, low, high)
    if low < high
        pivotIndex = Partition(A, low, high)
        QuickSort(A, low, pivotIndex - 1)  # Sort left subarray
        QuickSort(A, pivotIndex + 1, high) # Sort right subarray

Partition(A, low, high)
    pivot = A[high]  # Choosing last element as pivot
    i = low - 1
    for j = low to high - 1
        if A[j] < pivot
            i = i + 1
            swap(A[i], A[j])
    swap(A[i + 1], A[high])  # Place pivot in correct position
    return i + 1  # Return pivot index
```

Explanation of the Pseudo Code

A: The array to be sorted.
low, high: The start and end indices of the current subarray.
pivot: The selected element used to partition the array.
i: The index that tracks the smaller elements in partitioning.
Partition Function: Moves elements smaller than the pivot to the left and larger elements to the right.
Recursive Calls: QuickSort is applied to the left and right partitions.

COMPLEXITY ANALYSIS

1. Time Complexity Analysis

Best Case (Balanced Partitions)
The array is evenly divided in each recursive call.
Time Complexity: O(n log n)

Worst Case (Unbalanced Partitions)
If the pivot is always the smallest or largest element (sorted or reverse sorted array), partitions become one-sided.
Time Complexity: $O(n^2)$

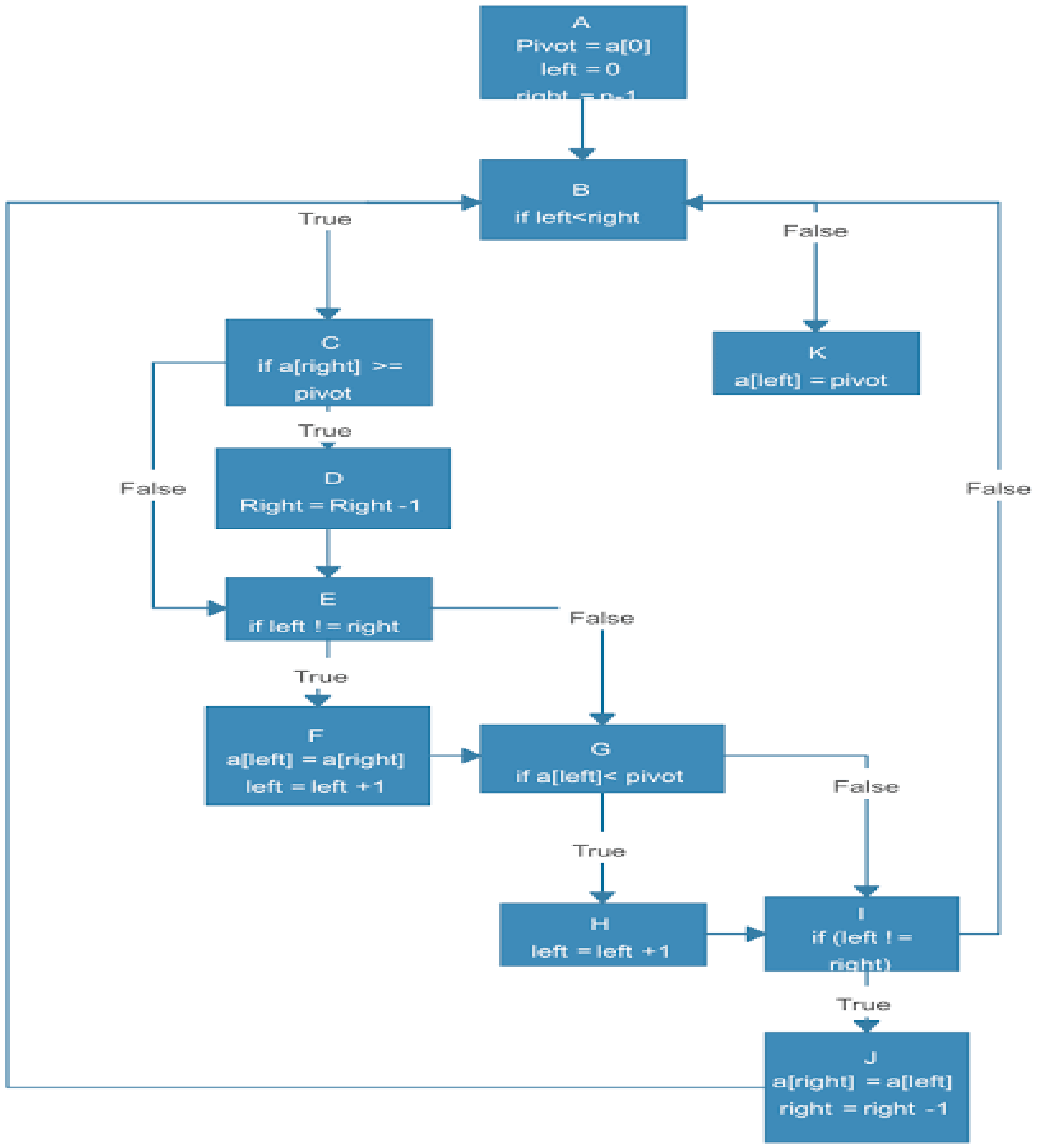
Average Case (Random Order)
On average, Quick Sort performs O(n log n) operations.
Time Complexity: O(n log n)

2. Space Complexity

Quick Sort is an in-place sorting algorithm, requiring only a few extra variables.

Space Complexity: O(1) (In-place)

In the worst case (unbalanced partitions), recursive calls may require O(n) extra stack space.



REFERENCES

1. Rohit Yadav et al. – "Analysis of Recursive and Non-recursive Merge Sort Algorithm"
Journal: International Journal of Advanced Research in Computer Science and Software Engineering
Volume & Issue: 3(11)
Date: November 2013
Pages: 977-981

2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford
Book: Introduction to Algorithms (3rd ed.)
Publisher: MIT Press
Year: 2009

3. Amazon Web Services
Source: Retrieved on March 1, 2011
Website: http://aws.amazon.com/

4. Radenski, A.
Paper: "Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs"
Conference: PDPTA'11, International Conference on Parallel and Distributed Processing Techniques and Applications
Publisher: CSREA Press (H. Arabnia, ed.)
Year: 2011
Pages: 367-373

5. R. Sedgewick
Book: Algorithms (2nd Edition)
Publisher: Addison-Wesley Publishing Company
Location: Reading, Mass
Year: 1988

6. Online Document:
Title: Parallel Merge
Source: http://penguin.ewu.edu/~trolfe/ParallelMerge/ParallelMerge.doc

7. V. Estivill-Castro and D. Wood
Paper: "A Survey of Adaptive Sorting Algorithms"
Journal: Computing Surveys
Volume: 24
Pages: 441-476
Year: 1992

8. A. M. Moffat and O. Peterson
Paper: "An Overview of Adaptive Sorting"
Journal: The Australian Computer Journal
Volume: 24
Pages: 70-77
Year: 1992

9. J. Katajainen, T. Pasanen, and J. Teubola
Paper: "Practical In-Place Merge Sort"
Journal: Nordic Journal of Computing
Volume: 3
Pages: 27-40
Year: 1996

10. R. Sedgewick (Duplicate Entry)
Book: Algorithms (2nd Edition)
Publisher: Addison-Wesley Publishing Company
Location: Reading, Mass
Year: 1988