Ain Shams University

Faculty of Engineering

Computer Engineering and Software Systems

I-CHEP

CSE331 – Data Structures and Algorithms

Fall 2024

Presented by:

Abdelrahman Mostafa Ali-Eldin	22P0150
Martin Magued Wadie	22P0193
Mohamed Ashraf Mohamed	22P0210
Mohamed Bashir Mohamed	22P0223
Seif Aly Othman Fahmy Youssef	22P0182

Presented to:

Dr. Hesham Farag

Table Of Contents

1. Introduction: -

Sorting is a fundamental operation in computer science, crucial for organizing data efficiently in various applications. With a wide range of sorting algorithms available, each with unique characteristics and performance metrics, selecting the most efficient algorithm for a given context becomes essential. This report outlines the development of an application designed to compare the efficiency of different sorting algorithms, providing insights into their performance and asymptotic behavior.

The primary objective of this application is to enable users to analyze and evaluate sorting algorithms based on their efficiency in sorting predefined or user-generated test data. Efficiency will be measured in terms of the number of computational steps each algorithm takes to complete the sorting process, offering a quantitative comparison. The application allows users to compare algorithms with one another or against their theoretical asymptotic efficiencies, such as O(n2)O(n^2) or O(nlogn)O(n \log n), to understand how well the practical performance aligns with theoretical expectations.

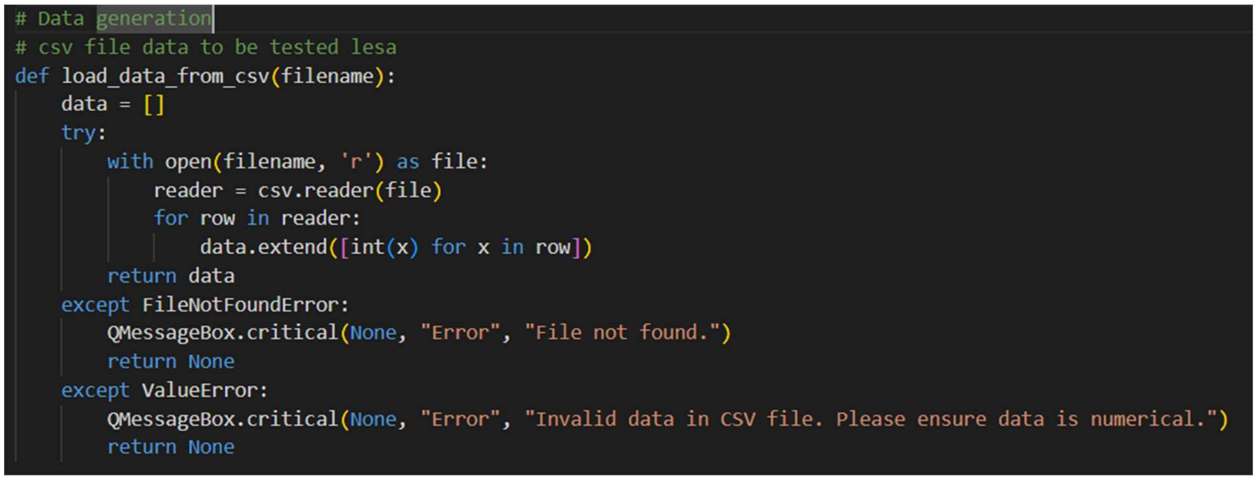
To achieve this goal, the application incorporates a predefined set of commonly used sorting algorithms, including insertion sort, merge sort, bubble sort, quick sort, and heap sort. Users can select specific algorithms for comparison and visualize the results through graphical representations. Additionally, test data can be generated either through the application itself or using external tools like Microsoft Excel, providing flexibility and convenience.

This report provides a comprehensive overview of the design, implementation, and testing of the application. By facilitating the evaluation of sorting algorithms in a practical and interactive manner, this tool aims to enhance understanding of their efficiencies and support informed decision-making in real-world scenarios.

2. Data generation: -

2.1 Load_data_from_csv: -

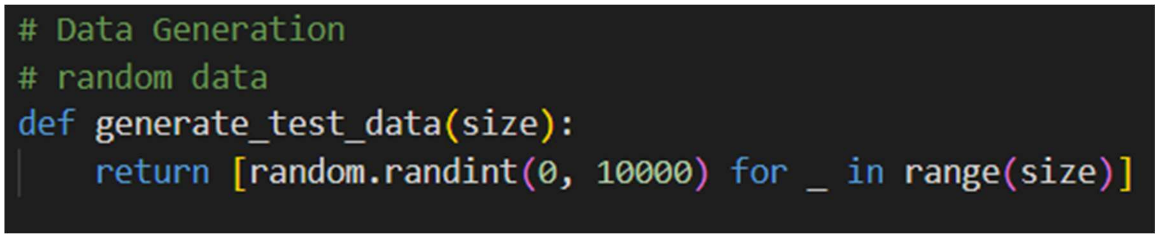
The load_data_from_csv(filename) function is designed to read numerical data from a CSV file and return it as a list. It opens the file in read mode and processes it line by line using Python's csv.reader module. Each element in the rows is converted into an integer and added to the data list. If the file does not exist, the function catches the FileNotFoundError exception and displays an error message using a QMessageBox. Similarly, if non-numerical data is encountered (causing a ValueError), the function alerts the user about the invalid data. In both error cases, it returns None. When successful, the function returns the numerical data, ensuring that only valid inputs are processed.



```python
# Data generation
# csv file data to be tested lesa
def load_data_from_csv(filename):
    data = []
    try:
        with open(filename, 'r') as file:
            reader = csv.reader(file)
            for row in reader:
                data.extend([int(x) for x in row])
        return data
    except FileNotFoundError:
        QMessageBox.critical(None, "Error", "File not found.")
        return None
    except ValueError:
        QMessageBox.critical(None, "Error", "Invalid data in CSV file. Please ensure data is numerical.")
        return None
```

2.2 Generate Random data: -

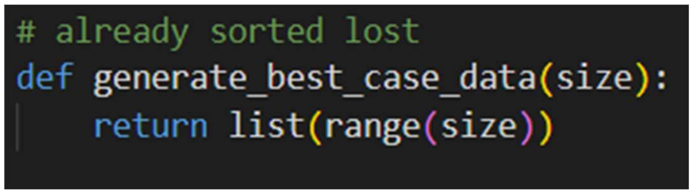
The generate_test_data(size) function generates a list of random integers, which can be used to test sorting algorithms under typical scenarios. The function takes the desired size of the dataset as input and uses Python's random.randint(0, 10000) to generate random numbers between 0 and 10,000. The resulting list is returned, offering a dataset with unpredictable values that simulates real-world conditions for sorting performance evaluation.



```python
# Data Generation
# random data
def generate_test_data(size):
    return [random.randint(0, 10000) for _ in range(size)]
```

2.3 Generating Best-Case Data: -

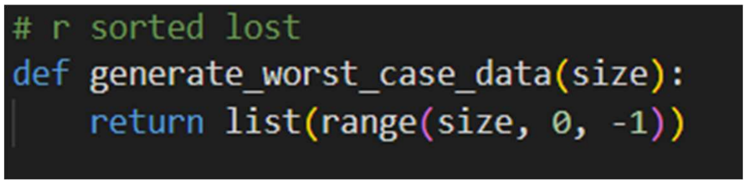
The generate_best_case_data(size) function creates a sorted list in ascending order, representing the "best-case scenario" for sorting algorithms like insertion sort. It takes the size of the list as input and generates a sequence of integers from 0 up to size - 1 using range(). The sorted list is then returned, providing input data where no further sorting is required, ideal for testing how efficiently algorithms handle already sorted data.



```python
# already sorted lost
def generate_best_case_data(size):
    return list(range(size))
```

2.4 Generating Worst-Case Data: -

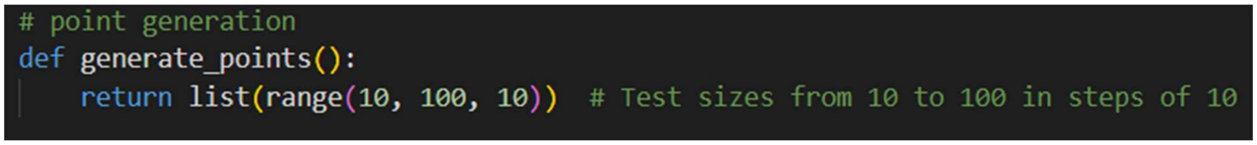
The generate_worst_case_data(size) function creates a list sorted in descending order, which represents the "worst-case scenario" for many sorting algorithms. The function accepts the desired size of the list and generates numbers starting from size down to 1. The descending order of the list challenges algorithms to rearrange the data entirely, testing their efficiency in handling highly disordered inputs.



```python
# r sorted lost
def generate_worst_case_data(size):
    return list(range(size, 0, -1))
```

2.5 Generating points: -

The generate_points() function generates a sequence of integers representing different sizes of input data for testing purposes. Using range (10, 100, 10), it creates a list of numbers starting at 10 and incrementing by 10 up to (but not including) 100. The resulting list, [10, 20, 30, 40, 50, 60, 70, 80, 90], defines specific input sizes for performance measurement, enabling algorithm efficiency comparisons as the dataset size increases.



```python
# point generation
def generate_points():
    return list(range(10, 100, 10))  # Test sizes from 10 to 100 in steps of 10
```

2.6 Purpose: -

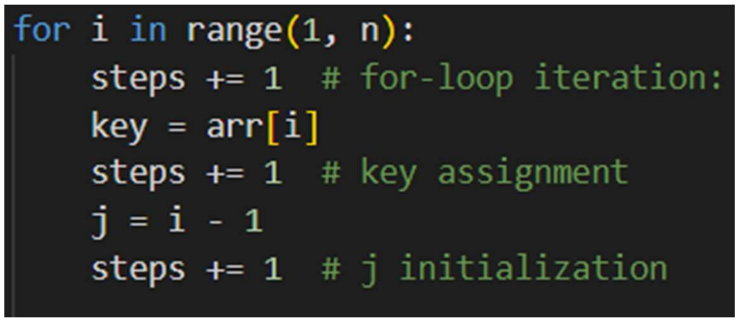
Together, these functions form a comprehensive toolkit for testing and comparing sorting algorithms under a variety of conditions. By enabling users to load real data, generate test datasets with different characteristics, and define input sizes systematically, they ensure thorough and flexible evaluation of sorting performance. This modular approach supports both general and edge-case testing, making it easier to assess the behavior of algorithms across diverse scenarios.

3. Sorting Algorithms: -

3.1 Insertion Sort: -

The (insertion_sort) function implements the insertion sort algorithm, a simple and intuitive sorting method. It works by dividing the array into a sorted and unsorted section and iteratively inserting elements from the unsorted section into their correct position in the sorted section. The function also counts the number of computational steps taken to sort the array, providing a measure of its efficiency.

The function begins by initializing a steps counter to zero and determining the length of the array n. The outer loop starts at the second element (index 1) and iterates through the array until the last element. For each iteration, the loop itself increments the steps counter by one, representing the overhead of the for loop's iteration logic. At each iteration, the current element (arr[i]) is assigned to the variable key, which holds the value being placed in its correct position. This assignment adds another increment to the steps counter. Additionally, a variable j is initialized to track the index of the last element in the sorted portion of the array, adding yet another step to the counter.

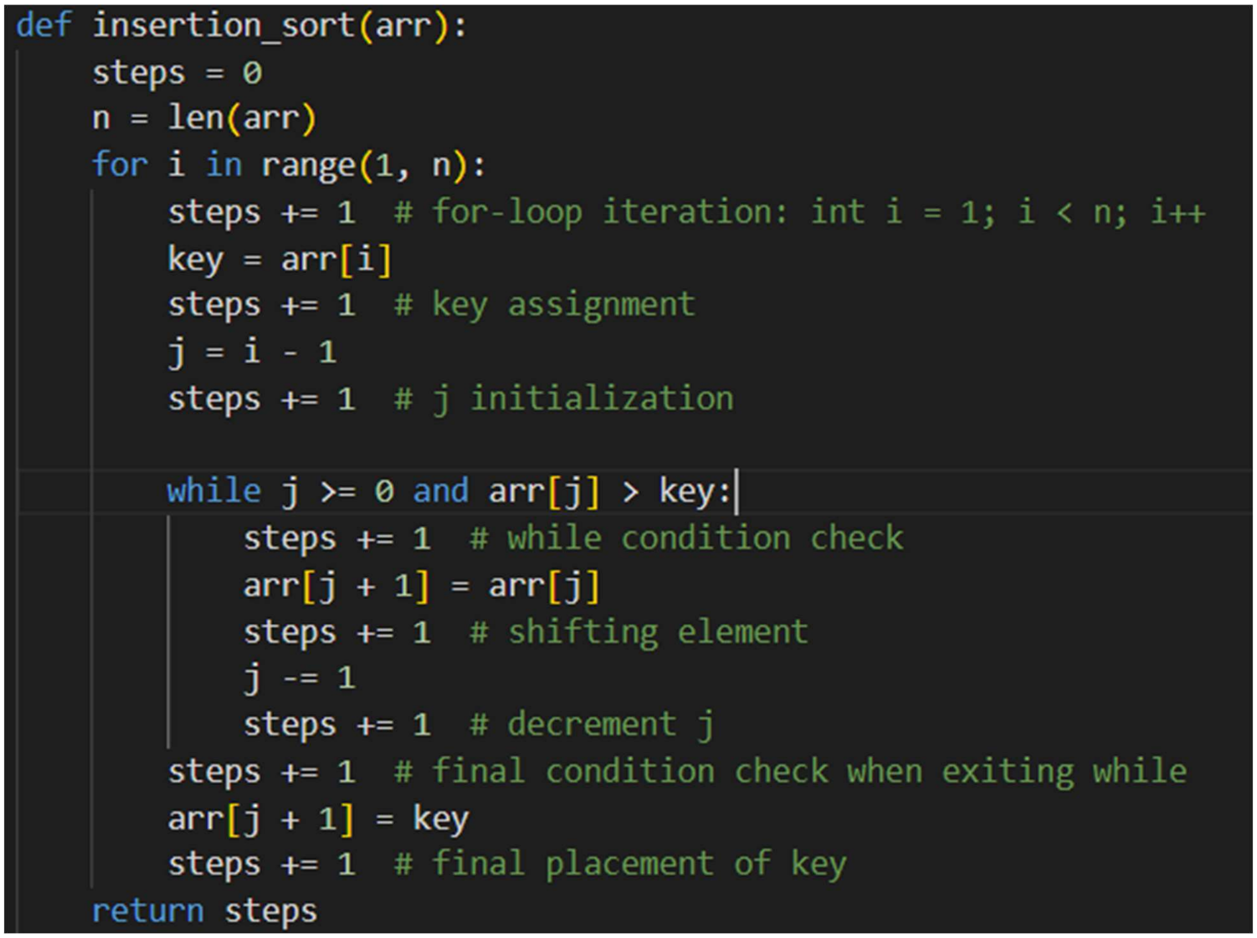


```
for i in range(1, n):
    steps += 1  # for-loop iteration:
    key = arr[i]
    steps += 1  # key assignment
    j = i - 1
    steps += 1  # j initialization
```

Within the inner while loop, the algorithm checks if the elements in the sorted portion of the array are greater than the key. This check counts as one step for every iteration of the while loop. If the condition is true, the element at arr[j] is shifted one position to the right to make space for the key, incrementing the steps counter for the assignment operation. The index j is then decremented to move backward through the sorted section, which is also counted as a step. The loop continues until the correct position for the key is found or until the beginning of the array is reached.

When the while loop exits, the algorithm places the key into its correct position in the array at arr[j + 1]. This operation is counted as an additional step. The process repeats for every element in the unsorted section of the array until the entire array is sorted. Notably, the steps counter also include the final condition check when the while loop terminates.

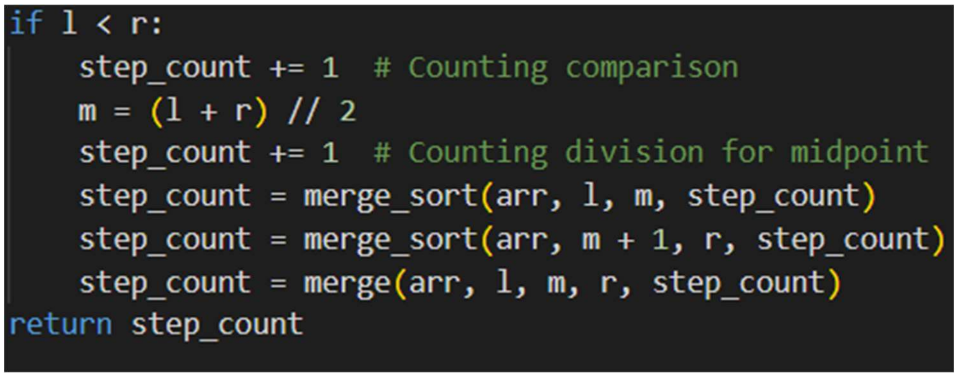
After the array is fully sorted, the function returns the total number of steps taken during the sorting process. These steps include all iterations of the for and while loops, as well as each assignment and comparison operation. This step count provides a quantitative measure of the algorithm's efficiency for a given input, making it useful for comparing the performance of insertion sort with other sorting algorithms or with its theoretical asymptotic behavior.



```python
def insertion_sort(arr):
    steps = 0
    n = len(arr)
    for i in range(1, n):
        steps += 1  # for-loop iteration: int i = 1; i < n; i++
        key = arr[i]
        steps += 1  # key assignment
        j = i - 1
        steps += 1  # j initialization

        while j >= 0 and arr[j] > key:
            steps += 1  # while condition check
            arr[j + 1] = arr[j]
            steps += 1  # shifting element
            j -= 1
            steps += 1  # decrement j
        steps += 1  # final condition check when exiting while
        arr[j + 1] = key
        steps += 1  # final placement of key
    return steps
```

3.2 Merge Sort: -

The merge_sort function implements the merge sort algorithm, a divide-and-conquer sorting method that recursively splits the array into smaller subarrays, sorts them, and then merges the sorted subarrays back together. The function also includes a step_count parameter to track the number of computational steps performed, providing a detailed measure of the algorithm's efficiency.



```
if l < r:
    step_count += 1  # Counting comparison
    m = (l + r) // 2
    step_count += 1  # Counting division for midpoint
    step_count = merge_sort(arr, l, m, step_count)
    step_count = merge_sort(arr, m + 1, r, step_count)
    step_count = merge(arr, l, m, r, step_count)
return step_count
```

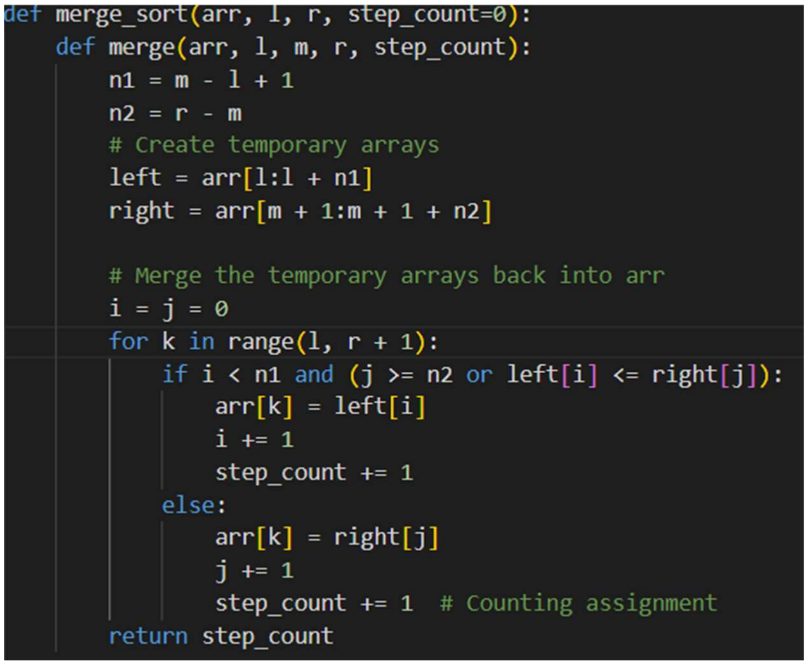
The recursive process begins by comparing the left (l) and right (r) indices of the current segment of the array. If l < r, the function increments the step_count to account for the comparison. The midpoint m is then calculated as (l + r) // 2, which adds another step for the division operation. The array is divided into two halves: the left subarray from index l to m and the right subarray from index m + 1 to r. The function then recursively calls itself to sort these two halves. Each recursive call adds steps for comparison and division, tracking the computational effort at each level of recursion.

Once the subarrays are sorted, the merge function is called to combine them into a single sorted segment. This process begins by determining the sizes of the left and right subarrays (n1 and n2) and creating temporary arrays to hold their values. These operations set up the merging process but do not directly contribute to the sorting step count.

The merging process uses two pointers, i and j, to traverse the left and right temporary arrays, respectively. For each position k in the original array, the algorithm compares the current elements of the left and right subarrays. If the left element is smaller or the right subarray is exhausted, the left element is placed in the original array, and i is incremented. Otherwise, the right element is placed in the original array, and j is incremented. Each assignment during this merging process increments the step_count. The function continues until all elements from both subarrays are merged.

Once the subarrays are sorted, the merge function is called to combine them into a single sorted segment. This process begins by determining the sizes of the left and right subarrays (n1 and n2) and creating temporary arrays to hold their values. These operations set up the merging process but do not directly contribute to the sorting step count.

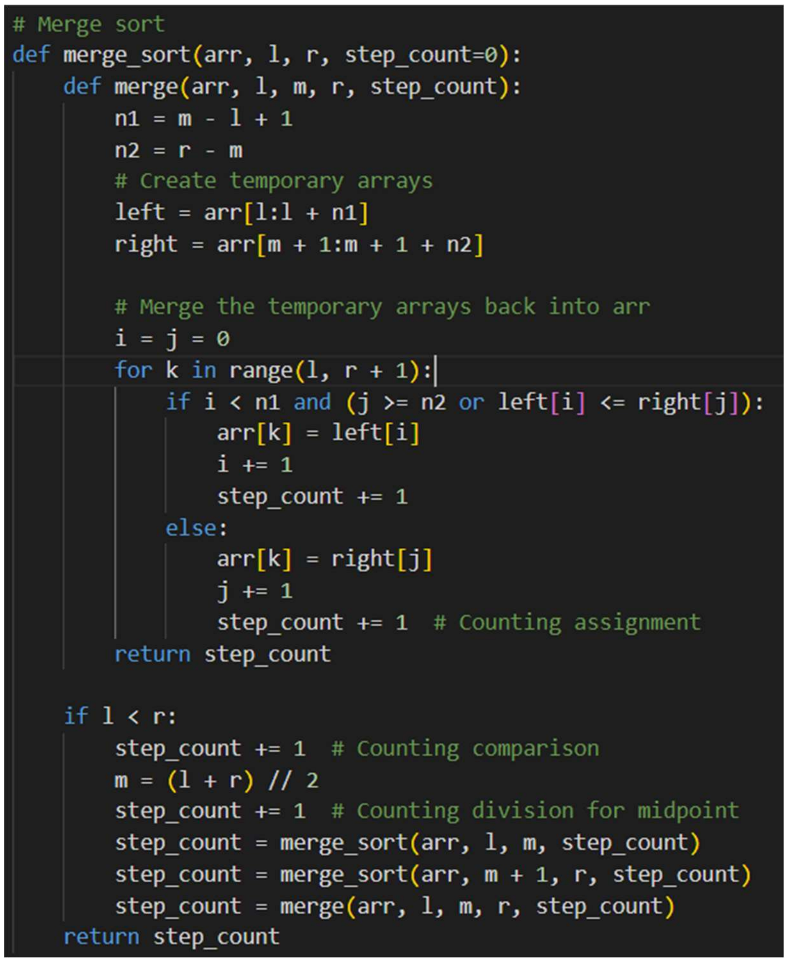
The merging process uses two pointers, i and j, to traverse the left and right temporary arrays, respectively. For each position k in the original array, the algorithm compares the current elements of the left and right subarrays. If the left element is smaller or the right subarray is exhausted, the left element is placed in the original array, and i is incremented. Otherwise, the right element is placed in the original array, and j is incremented. Each assignment during this merging process increments the step_count. The function continues until all elements from both subarrays are merged.



```python
def merge_sort(arr, l, r, step_count=0):
    def merge(arr, l, m, r, step_count):
        n1 = m - l + 1
        n2 = r - m
        # Create temporary arrays
        left = arr[l:l + n1]
        right = arr[m + 1:m + 1 + n2]

        # Merge the temporary arrays back into arr
        i = j = 0
        for k in range(l, r + 1):
            if i < n1 and (j >= n2 or left[i] <= right[j]):
                arr[k] = left[i]
                i += 1
                step_count += 1
            else:
                arr[k] = right[j]
                j += 1
                step_count += 1  # Counting assignment
        return step_count
```

The recursive nature of merge sort ensures that each level of the recursion divides the array until each segment contains a single element. These individual elements are inherently sorted. The merging process then combines these small segments back into larger sorted segments, eventually resulting in a fully sorted array. During each recursive call, the step_count accumulates the steps taken for comparisons, assignments, and merging operations, capturing the computational workload at each level.

After the entire array is sorted, the merge_sort function returns the total step_count, which reflects the cumulative number of steps taken throughout the sorting process. This count includes all comparisons, assignments, and recursive operations, making it a comprehensive measure of merge sort's performance. By tracking these steps, the function provides valuable insights into how merge sort performs for different input sizes and data configurations.



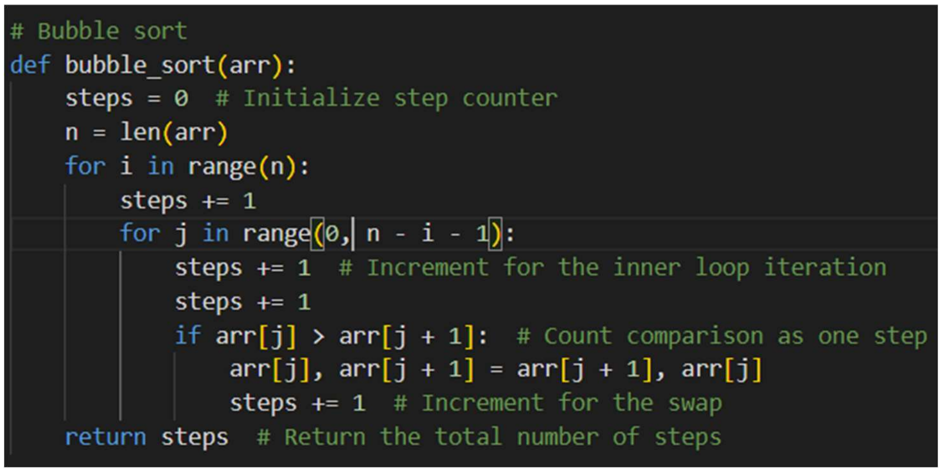
```python
# Merge sort
def merge_sort(arr, l, r, step_count=0):
    def merge(arr, l, m, r, step_count):
        n1 = m - l + 1
        n2 = r - m
        # Create temporary arrays
        left = arr[l:l + n1]
        right = arr[m + 1:m + 1 + n2]

        # Merge the temporary arrays back into arr
        i = j = 0
        for k in range(l, r + 1):
            if i < n1 and (j >= n2 or left[i] <= right[j]):
                arr[k] = left[i]
                i += 1
                step_count += 1
            else:
                arr[k] = right[j]
                j += 1
                step_count += 1  # Counting assignment
        return step_count

    if l < r:
        step_count += 1  # Counting comparison
        m = (l + r) // 2
        step_count += 1  # Counting division for midpoint
        step_count = merge_sort(arr, l, m, step_count)
        step_count = merge_sort(arr, m + 1, r, step_count)
        step_count = merge(arr, l, m, r, step_count)
    return step_count
```

3.3 Bubble sort: -

The bubble_sort function implements the bubble sort algorithm, one of the simplest sorting techniques. It works by repeatedly traversing the array and swapping adjacent elements that are in the wrong order. The algorithm continues this process until the array is sorted. To measure the algorithm's efficiency, the function counts the number of computational steps taken during execution, providing insight into its performance.

The algorithm begins by initializing a step counter to zero and determining the length n of the array. The outer for loop iterates through the entire array, with the variable i indicating the current pass number. For each pass, the steps counter is incremented by one to account for the loop's iteration logic. The number of iterations for the inner loop decreases with each pass because the largest elements gradually "bubble up" to their correct positions, reducing the need for comparisons in subsequent passes.

Within the outer loop, an inner for loop iterates through the unsorted portion of the array. For each pair of adjacent elements, the algorithm checks whether the current element (arr[j]) is greater than the next element (arr[j + 1]). This comparison increments the steps counter. If the elements are out of order, they are swapped to move the larger element toward the end of the array. The swap operation also increments the steps counter, reflecting the computational effort required to reassign the values.



```python
# Bubble sort
def bubble_sort(arr):
    steps = 0  # Initialize step counter
    n = len(arr)
    for i in range(n):
        steps += 1
        for j in range(0, n - i - 1):
            steps += 1  # Increment for the inner loop iteration
            steps += 1
            if arr[j] > arr[j + 1]:  # Count comparison as one step
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                steps += 1  # Increment for the swap
    return steps  # Return the total number of steps
```

3.4 Selection Sort: -

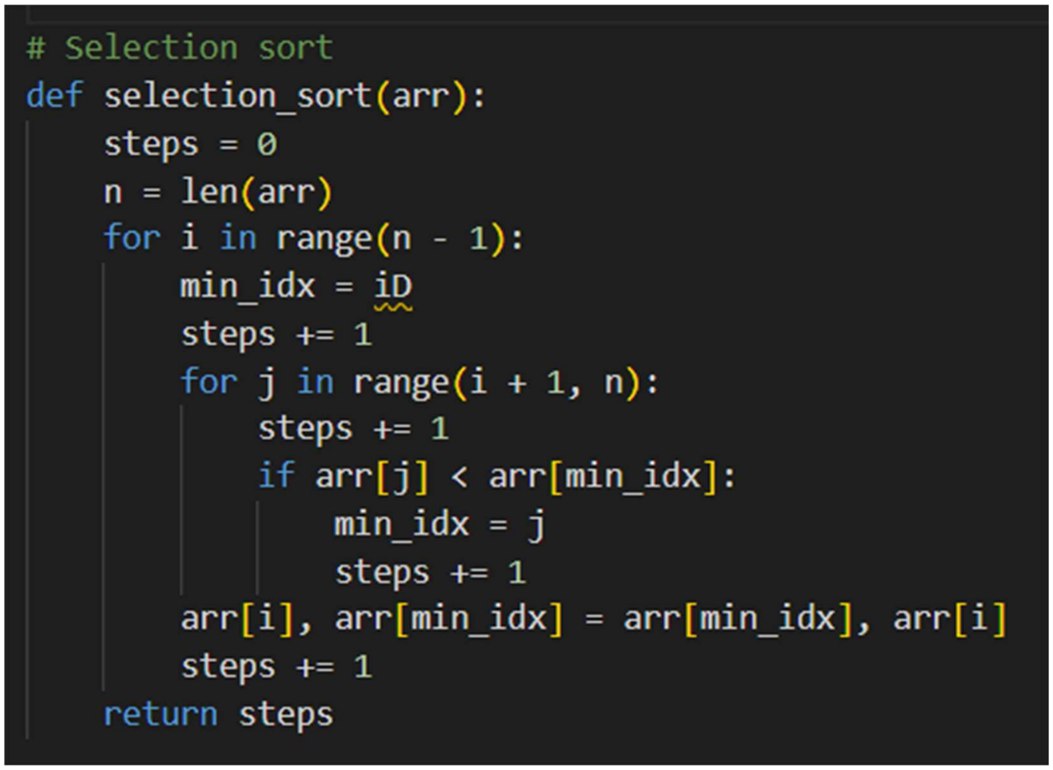
The selection_sort function implements the selection sort algorithm, a straightforward sorting technique that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and swapping it with the first element of that portion. This implementation also includes a steps counter to measure the computational steps required during the sorting process, providing insights into its efficiency.

The algorithm begins by initializing the steps counter to zero and calculating the length n of the array. The outer loop iterates from the first element to the second-to-last element of the array. For each iteration, the variable i represents the current position in the sorted portion of the array, and min_idx is initialized to i, indicating the index of the smallest element found so far. This initialization is counted as a step. The outer loop's purpose is to ensure that all elements of the array are processed, one by one, until the array is fully sorted.

Within the outer loop, an inner loop traverses the unsorted portion of the array, starting from i + 1 to n. For each iteration, the algorithm compares the current element arr[j] with the smallest element found so far, arr[min_idx]. This comparison is counted as a step. If a smaller element is found, min_idx is updated to the current index j, which is also counted as a step. The inner loop ensures that by the end of each outer loop iteration, the smallest element in the unsorted portion is identified and its index stored in min_idx.

After the inner loop completes, the algorithm swaps the smallest element in the unsorted portion (located at min_idx) with the first element of that portion (located at i). This swap is counted as a step and ensures that the smallest element is moved to its correct position in the sorted portion of the array. The process is repeated for each position in the array until all elements are sorted.

Once the array is fully sorted, the function returns the total number of steps recorded during the sorting process. These steps include all comparisons, assignments, and loop iterations, offering a detailed view of the computational effort required. While selection sort is simple and easy to implement, its inefficiency becomes evident in larger datasets, as it requires a significant number of comparisons even for already sorted or partially sorted arrays. By counting steps, the algorithm's limitations in terms of performance can be better understood and analyzed.

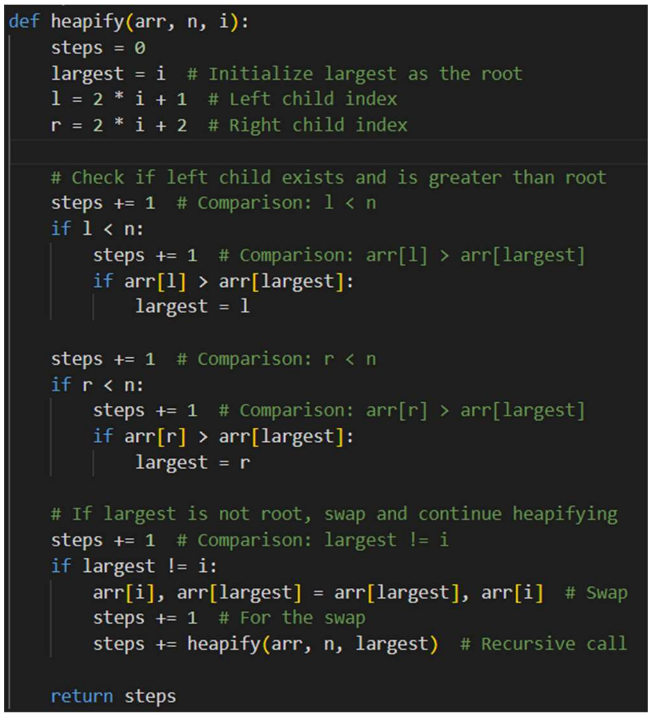


```python
# Selection sort
def selection_sort(arr):
    steps = 0
    n = len(arr)
    for i in range(n - 1):
        min_idx = iD
        steps += 1
        for j in range(i + 1, n):
            steps += 1
            if arr[j] < arr[min_idx]:
                min_idx = j
                steps += 1
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
        steps += 1
    return steps
```

3.5 Heap Sort: -

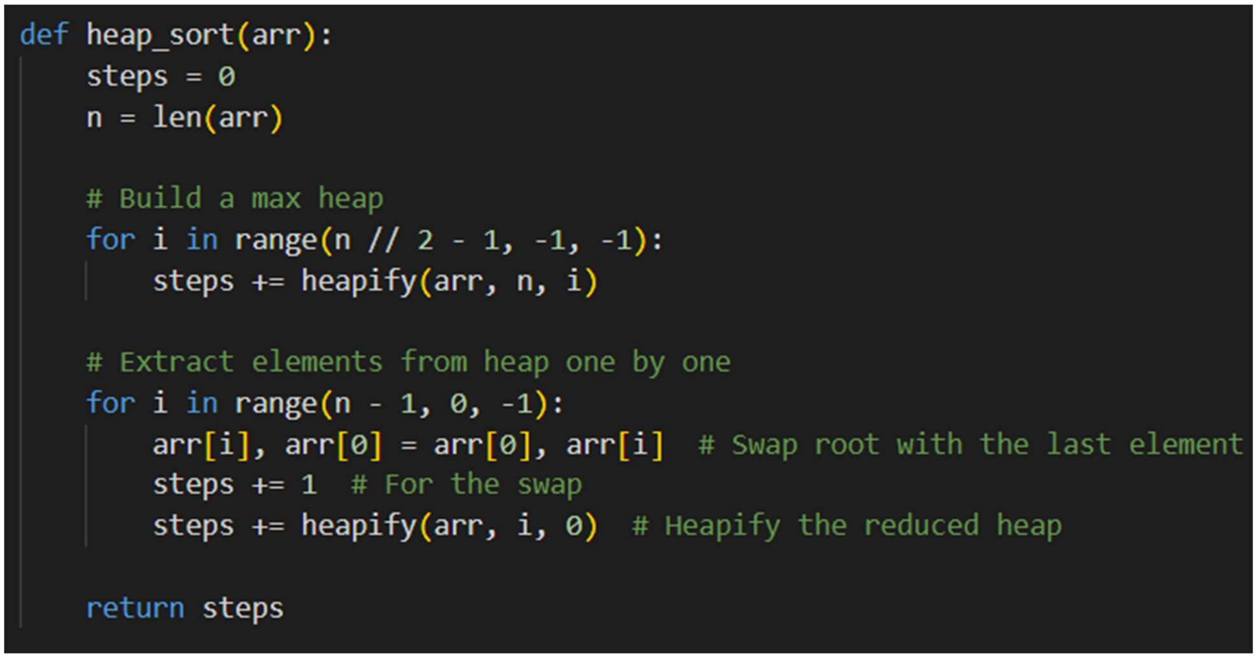
The first phase of heap sort involves transforming the input array into a max heap, where each parent node is greater than or equal to its child nodes. This is achieved through the heapify function, which ensures the heap property for a given subtree. The outer loop in heap_sort iterates from the last non-leaf node to the root node (indices n // 2 - 1 to 0), calling heapify for each node. The steps counter tracks the computational effort of each call, including comparisons, swaps, and recursive operations performed within heapify.

The heapify function plays a critical role in maintaining the max heap property. For a given node i, it compares the node with its left and right children (if they exist) to identify the largest value. These comparisons are tracked as steps. If a child node is larger than the parent, the largest node is swapped with the parent, and heapify is recursively called on the affected subtree. The recursive calls ensure that the heap property is preserved throughout the subtree. Each swap and recursive call contributes to the overall steps count.



```python
def heapify(arr, n, i):
    steps = 0
    largest = i  # Initialize largest as the root
    l = 2 * i + 1  # Left child index
    r = 2 * i + 2  # Right child index

    # Check if left child exists and is greater than root
    steps += 1  # Comparison: l < n
    if l < n:
        steps += 1  # Comparison: arr[l] > arr[largest]
        if arr[l] > arr[largest]:
            largest = l

    steps += 1  # Comparison: r < n
    if r < n:
        steps += 1  # Comparison: arr[r] > arr[largest]
        if arr[r] > arr[largest]:
            largest = r

    # If largest is not root, swap and continue heapifying
    steps += 1  # Comparison: largest != i
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        steps += 1  # For the swap
        steps += heapify(arr, n, largest)  # Recursive call

    return steps
```

Once the max heap is built, the sorting phase begins. The algorithm repeatedly swaps the root of the heap (the largest element) with the last element of the unsorted portion of the array. This operation places the largest element in its correct position in the sorted portion. After each swap, the size of the heap is reduced by one, and the heapify function is called on the root to restore the heap property in the reduced heap. Each swap and subsequent heapify call increments the steps counter, capturing the computational effort of these operations.



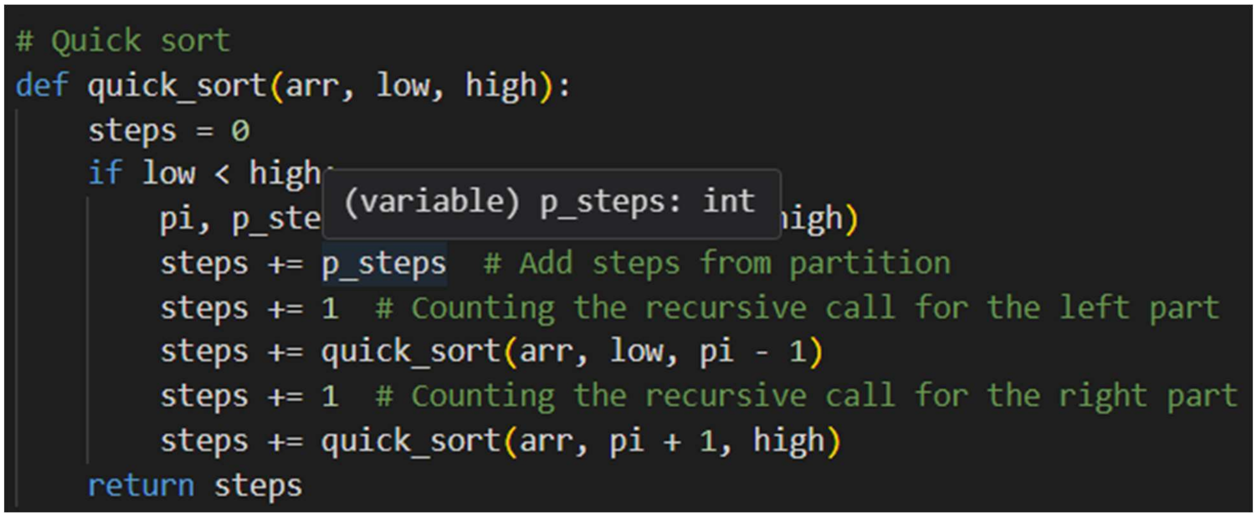
```python
def heap_sort(arr):
    steps = 0
    n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        steps += heapify(arr, n, i)

    # Extract elements from heap one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap root with the last element
        steps += 1  # For the swap
        steps += heapify(arr, i, 0)  # Heapify the reduced heap

    return steps
```

After all elements are extracted and the array is fully sorted, the function returns the total steps count. This count includes the operations performed during the heap construction phase and the sorting phase, providing a comprehensive measure of the algorithm’s computational workload. Heap sort is known for its efficiency and stable performance, with a time complexity of O(nlog n)O(n \log n)O(nlogn). The step counting mechanism in this implementation highlights the structured nature of the algorithm and its performance advantages compared to simpler sorting techniques.

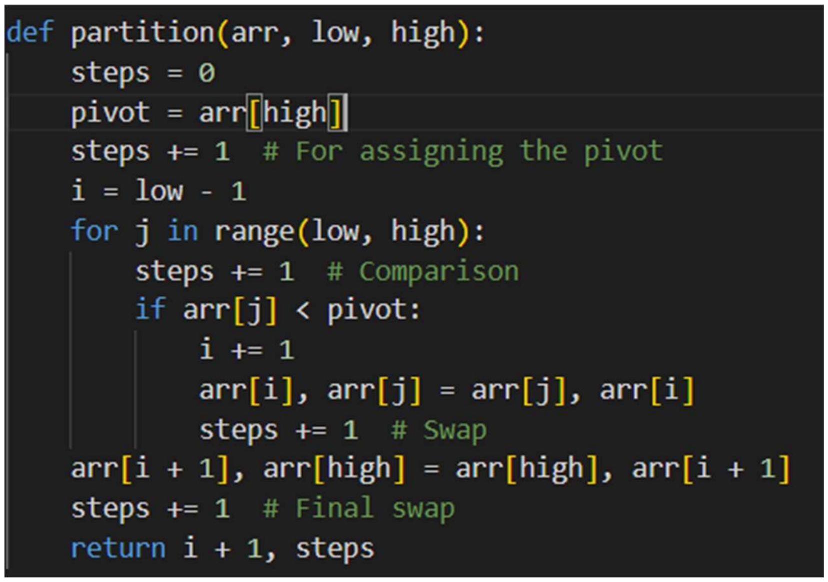
3.6 Quick Sort: -

The quick_sort function implements the quick sort algorithm, a widely used and efficient sorting technique based on the divide-and-conquer paradigm. It recursively partitions the input array into smaller subarrays around a chosen pivot element and sorts these subarrays. This implementation also tracks the number of computational steps performed using a steps counter, providing detailed insights into its efficiency.

The quick_sort function begins by checking whether the current segment of the array (low to high) contains more than one element. If this condition is met, the function proceeds to partition the array and recursively sort the left and right subarrays. Each recursive call contributes to the steps count, capturing the effort of subdividing the problem into smaller parts. The base case, where low is not less than high, terminates recursion without adding additional steps, ensuring that the algorithm completes efficiently.



```python
# Quick sort
def quick_sort(arr, low, high):
    steps = 0
    if low < high
        pi, p_ste (variable) p_steps: int high)
        steps += p_steps  # Add steps from partition
        steps += 1  # Counting the recursive call for the left part
        steps += quick_sort(arr, low, pi - 1)
        steps += 1  # Counting the recursive call for the right part
        steps += quick_sort(arr, pi + 1, high)
    return steps
```



```python
def partition(arr, low, high):
    steps = 0
    pivot = arr[high]
    steps += 1  # For assigning the pivot
    i = low - 1
    for j in range(low, high):
        steps += 1  # Comparison
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
            steps += 1  # Swap
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    steps += 1  # Final swap
    return i + 1, steps
```

The partition function is central to the quick sort process. It selects a pivot element, typically the last element in the current segment of the array and rearranges the array such that all elements smaller than the pivot are moved to its left and all elements greater than the pivot are moved to its right. The function begins by assigning the pivot and initializing the pointer i to track the boundary of the smaller elements, incrementing the steps counter for these operations.

During the partitioning process, the function iterates through the array from low to high - 1. For each element, a comparison is made with the pivot, which increments the steps counter. If the current element is smaller than the pivot, the pointer i is incremented, and the element is swapped with the one at position i. Each swap adds to the steps counter, reflecting the computational effort of reordering elements. After completing the iteration, the pivot is swapped with the element at position i + 1, placing it in its correct position in the sorted array. This final swap also contributes to the steps count.

After partitioning, the array is divided into two subarrays: one to the left of the pivot (containing elements smaller than the pivot) and one to the right (containing elements larger than the pivot). The quick_sort function is called recursively on these subarrays. Each recursive call adds steps for the partitioning process and further recursive invocations, ensuring that all elements are sorted into their correct positions.

Once the entire array is sorted, the total number of steps recorded during the sorting process is returned. This count includes comparisons, swaps, and recursive operations, providing a detailed measure of the algorithm’s efficiency. Quick sort is known for its average-case time complexity of O(nlog⁡n)O(n \log n)O(nlogn) and its ability to perform well on large datasets. The inclusion of step counting in this implementation highlights the algorithm's structured approach and demonstrates its performance advantages over simpler sorting techniques, particularly for large or randomly ordered inputs.

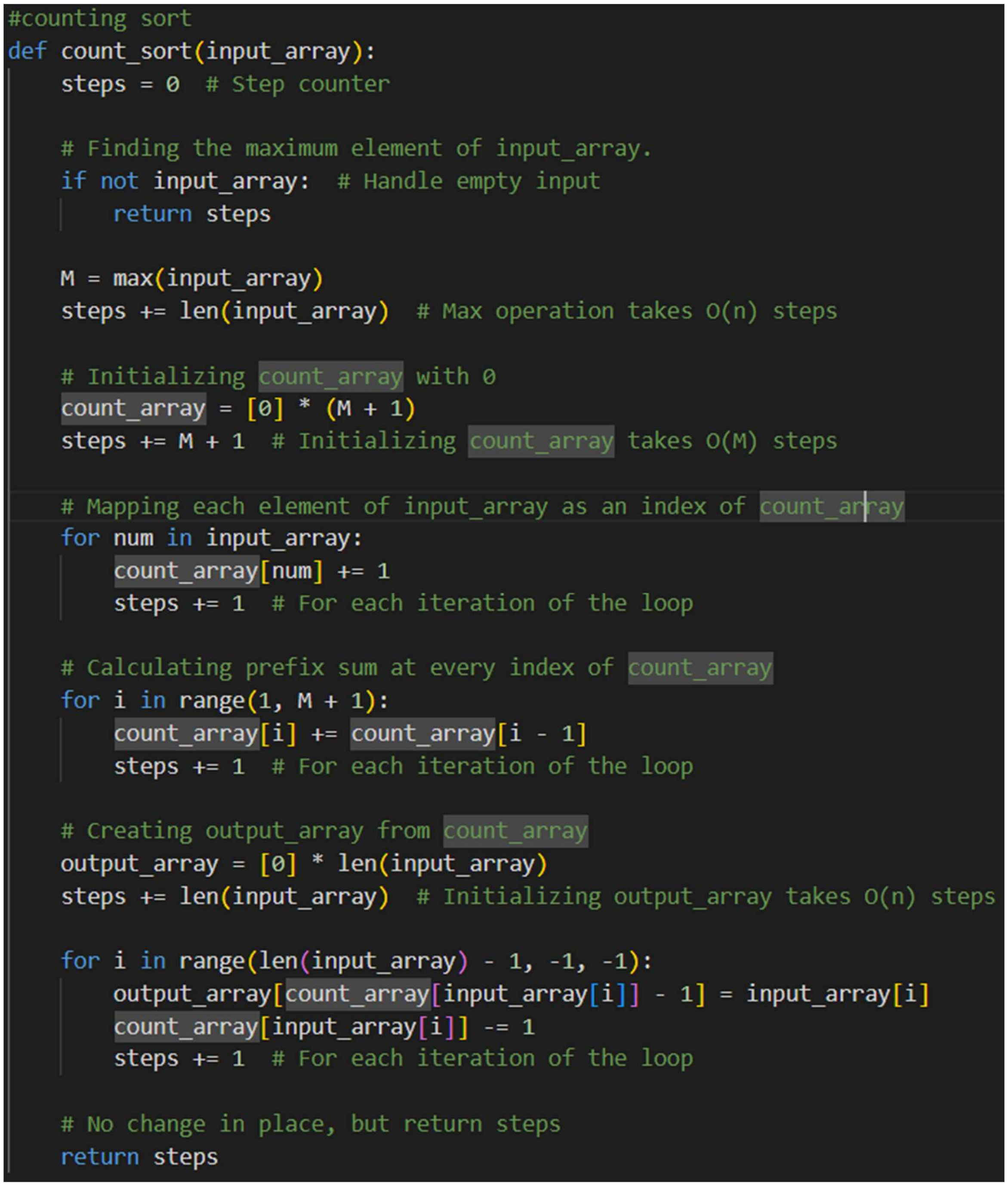
3.7 Counting Sort: -

The count_sort and countingSort functions implement two versions of counting sort, an integer sorting algorithm that relies on counting the occurrences of each element within a range of values. Both versions also include a step counter to track the number of operations performed during the sorting process, providing a measure of the algorithm's efficiency.

The first function, count_sort, operates by counting the frequency of each element in the input array and using that frequency to place elements in sorted order. It begins by handling the edge case where the input array is empty. If the array is not empty, the maximum value (M) is determined, and the steps counter is incremented by the number of elements in the array to reflect the time taken to find the maximum.

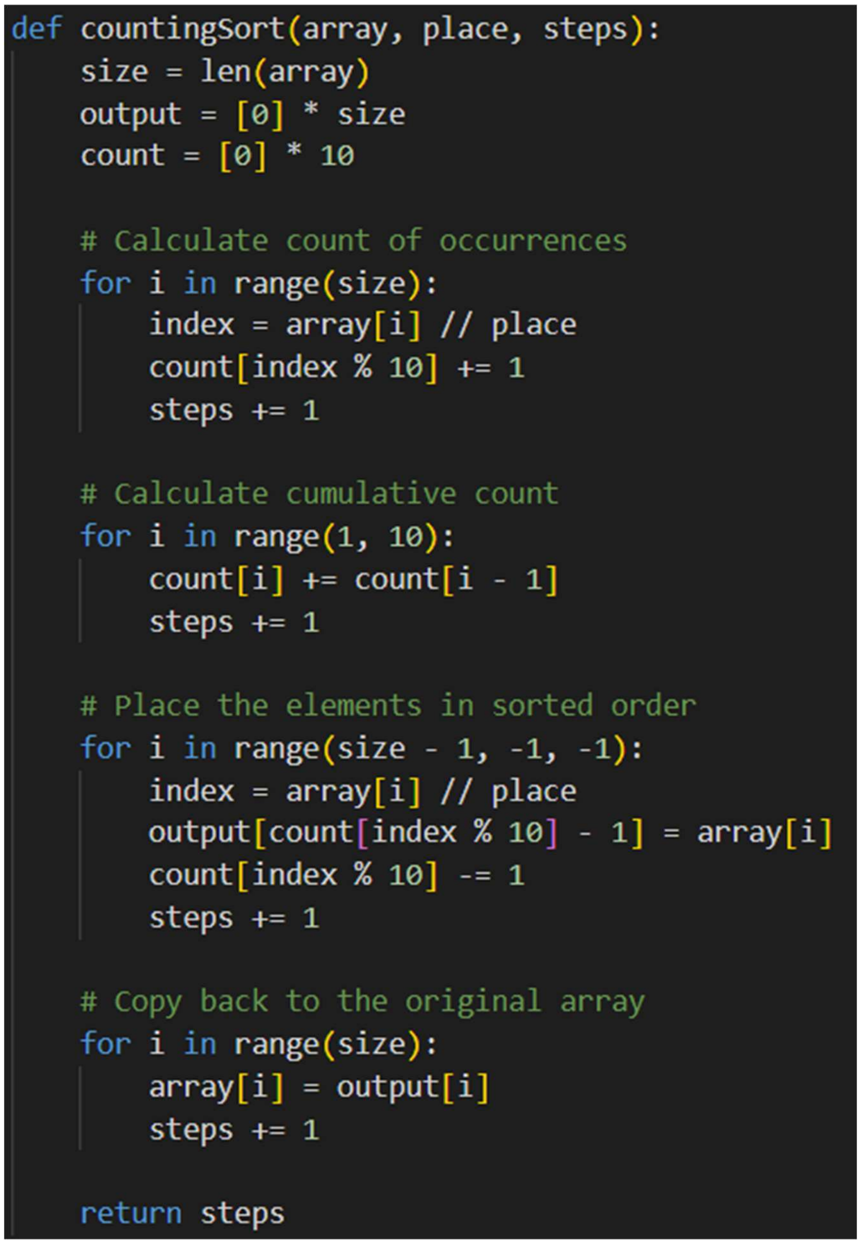
Next, a count_array is initialized with zeros, and its size is determined by the maximum value M plus one. This initialization is counted as M + 1 steps. The algorithm then iterates over the input_array, incrementing the corresponding indices in the count_array based on the value of each element. This mapping is done for each element of the array, with one step added for each iteration.

After that, the algorithm calculates the cumulative sum of the count_array, updating each position to hold the total number of elements less than or equal to that index. This cumulative summing process is accounted for as one step for each element in the count_array. Then, an output_array is created and filled with the sorted elements, placing each element based on its position in the cumulative count_array. This final pass through the array involves one step for each element placed in the output_array.



```python
#counting sort
def count_sort(input_array):
    steps = 0  # Step counter

    # Finding the maximum element of input_array.
    if not input_array:  # Handle empty input
        return steps

    M = max(input_array)
    steps += len(input_array)  # Max operation takes O(n) steps

    # Initializing count_array with 0
    count_array = [0] * (M + 1)
    steps += M + 1  # Initializing count_array takes O(M) steps

    # Mapping each element of input_array as an index of count_array
    for num in input_array:
        count_array[num] += 1
        steps += 1  # For each iteration of the loop

    # Calculating prefix sum at every index of count_array
    for i in range(1, M + 1):
        count_array[i] += count_array[i - 1]
        steps += 1  # For each iteration of the loop

    # Creating output_array from count_array
    output_array = [0] * len(input_array)
    steps += len(input_array)  # Initializing output_array takes O(n) steps

    for i in range(len(input_array) - 1, -1, -1):
        output_array[count_array[input_array[i]] - 1] = input_array[i]
        count_array[input_array[i]] -= 1
        steps += 1  # For each iteration of the loop

    # No change in place, but return steps
    return steps
```

The steps counter provides an overall measure of the algorithm’s computational effort. The count reflects the time complexity of counting sort, which is generally O(n+k)O(n + k)O(n+k), where n is the number of elements in the input array and k is the range of the input values (for the non-radix version) or the number of digits in the numbers (for the radix version). The step counting mechanism allows for a detailed analysis of the operations involved, making it easier to assess the algorithm's performance for different input sizes and ranges of values.



```python
def countingSort(array, place, steps):
    size = len(array)
    output = [0] * size
    count = [0] * 10

    # Calculate count of occurrences
    for i in range(size):
        index = array[i] // place
        count[index % 10] += 1
        steps += 1

    # Calculate cumulative count
    for i in range(1, 10):
        count[i] += count[i - 1]
        steps += 1

    # Place the elements in sorted order
    for i in range(size - 1, -1, -1):
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1
        steps += 1

    # Copy back to the original array
    for i in range(size):
        array[i] = output[i]
        steps += 1

    return steps
```

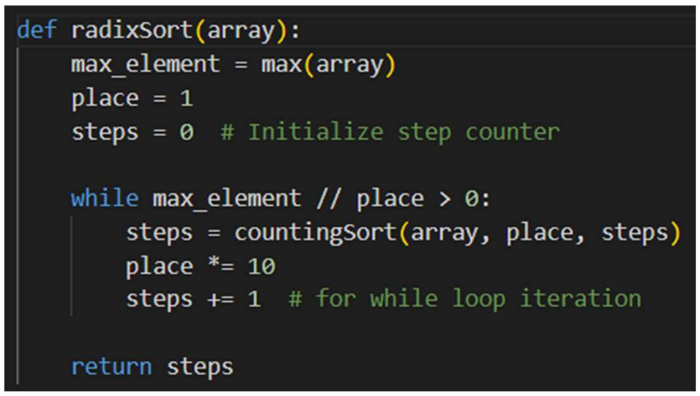
3.8 Radix Sort: -

The radixSort function implements the radix sort algorithm, which sorts numbers by processing individual digits starting from the least significant digit (LSD) to the most significant digit (MSD). This approach uses a stable sorting algorithm, typically counting sort, to sort the digits at each place value (ones, tens, hundreds, etc.). In this implementation, a step counter is utilized to track the number of operations performed throughout the sorting process.

The radix sort begins by identifying the largest number in the input array (max_element). This value is used to determine the number of iterations required to process all digits in the array, as each iteration handles one digit (place) at a time. The place variable starts at 1 (representing the ones place) and is multiplied by 10 after each iteration to move to the next higher place (tens, hundreds, etc.).

The algorithm enters a while loop that continues as long as there are digits to process in the largest number (max_element // place > 0). In each iteration, the countingSort function is called to sort the elements based on the current digit (represented by place). The steps counter is updated by the number of steps returned by the countingSort function, which includes comparisons, assignments, and any other operations involved in sorting the current digit. The place is then multiplied by 10 to move to the next digit, and one additional step is added for the while loop iteration itself.

The radixSort function returns the total number of steps performed during the sorting process. This step count reflects the operations involved in sorting each digit of the numbers, with a time complexity of O(n·k)O(n \cdot k), where n is the number of elements and k is the number of digits in the largest number. The step counter provides a detailed analysis of the algorithm's performance.



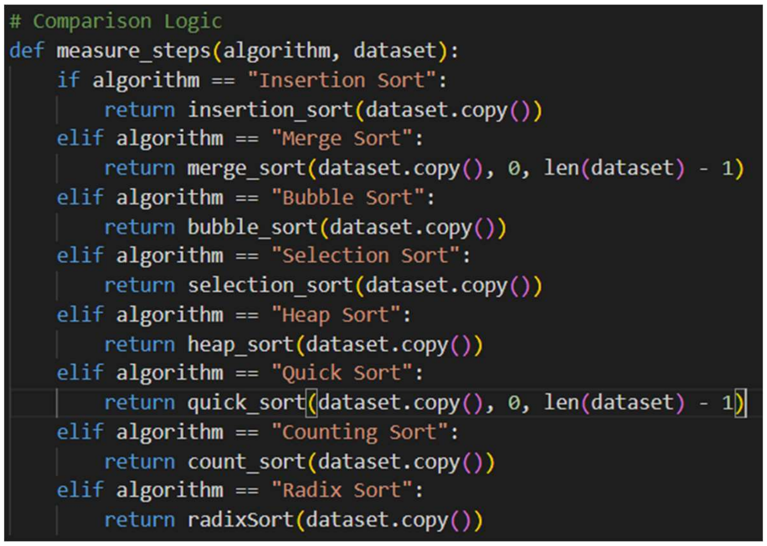
```python
def radixSort(array):
    max_element = max(array)
    place = 1
    steps = 0  # Initialize step counter

    while max_element // place > 0:
        steps = countingSort(array, place, steps)
        place *= 10
        steps += 1  # for while loop iteration

    return steps
```

4. Measuring steps: -

The measure_steps function is designed to evaluate and compare the efficiency of different sorting algorithms by counting the number of steps each algorithm takes to sort a given dataset. It takes two parameters: algorithm (a string specifying which sorting algorithm to use) and dataset (the list of data to be sorted). The function uses a series of conditional checks to determine which sorting algorithm to execute based on the algorithm argument.

The function first checks the algorithm parameter. Depending on its value, the function selects the appropriate sorting algorithm and applies it to a copy of the provided dataset. This ensures that the original dataset remains unmodified. For instance, if the algorithm is "Insertion Sort," the function will call insertion_sort on a copy of the dataset. Similarly, for other algorithms like "Merge Sort," "Bubble Sort," "Quick Sort," and others, the function will invoke their respective functions. Each of these functions sorts the dataset and returns the total number of steps the algorithm took to complete the sorting process.

After executing the selected sorting algorithm, the measure_steps function returns the total step count, which is a numerical measure of how many operations (comparisons, assignments, and other actions) the algorithm performed during the sorting process. This step count provides insight into the computational effort of each algorithm, allowing comparisons of their efficiencies.



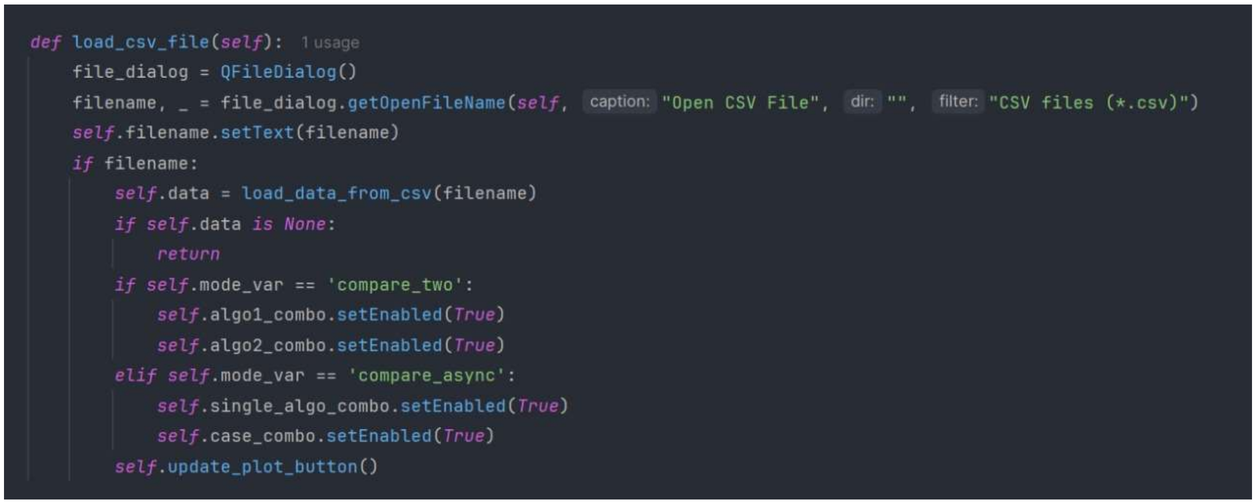
```python
# Comparison Logic
def measure_steps(algorithm, dataset):
    if algorithm == "Insertion Sort":
        return insertion_sort(dataset.copy())
    elif algorithm == "Merge Sort":
        return merge_sort(dataset.copy(), 0, len(dataset) - 1)
    elif algorithm == "Bubble Sort":
        return bubble_sort(dataset.copy())
    elif algorithm == "Selection Sort":
        return selection_sort(dataset.copy())
    elif algorithm == "Heap Sort":
        return heap_sort(dataset.copy())
    elif algorithm == "Quick Sort":
        return quick_sort(dataset.copy(), 0, len(dataset) - 1)
    elif algorithm == "Counting Sort":
        return count_sort(dataset.copy())
    elif algorithm == "Radix Sort":
        return radixSort(dataset.copy())
```

5. Graphical User Interface (GUI): -

5.1 Load CSV File

Function Overview: The load_csv_file method is a key component of the program's file handling system. This function is responsible for facilitating user interaction to load CSV files into the application, which serves as the primary data input method for subsequent processing and analysis.

Code:



```
def load_csv_file(self):  1 usage
    file_dialog = QFileDialog()
    filename, _ = file_dialog.getOpenFileName(self, caption: "Open CSV File", dir: "", filter: "CSV files (*.csv)")
    self.filename.setText(filename)
    if filename:
        self.data = load_data_from_csv(filename)
        if self.data is None:
            return
        if self.mode_var == 'compare_two':
            self.algo1_combo.setEnabled(True)
            self.algo2_combo.setEnabled(True)
        elif self.mode_var == 'compare_async':
            self.single_algo_combo.setEnabled(True)
            self.case_combo.setEnabled(True)
        self.update_plot_button()
```

Functionality Breakdown:

- **File Selection:** The method uses QFileDialog to prompt the user with a file selection dialog box, allowing them to browse and select a CSV file. This ensures a user-friendly and intuitive method for importing data.
- **File Path Retrieval:** The selected file's path is stored in the filename variable. The line filename, _ = file_dialog.getOpenFileName(self, "Open CSV File", "", "CSV files (*.csv)") ensures that only CSV files are filtered and displayed in the dialog, thereby reducing the risk of loading unsupported file formats.
- **UI Update:** After a file is selected, the path is displayed on the GUI using self.filename.setText(filename), providing users with visual confirmation of the file chosen.

- **Data Loading:** The selected CSV file is processed through the load_data_from_csv function, which parses and loads the data into memory. If the data fails to load (indicated by a None return value), the function exits early, preventing further execution.
- **Mode Handling:** Depending on the operational mode (self.mode_var), different sections of the GUI are activated:
    - In 'compare_two' mode, two algorithm selection combo boxes (self.algo1_combo and self.algo2_combo) are enabled, allowing users to select and compare two algorithms.
    - In 'compare_async' mode, a single algorithm combo box (self.single_algo_combo) and a test case combo box (self.case_combo) are enabled, facilitating asynchronous comparison against different datasets or cases.
- **Plot Update:** Finally, the update_plot_button method is called to refresh the plot based on the newly loaded data, ensuring the graphical representation reflects the latest input.

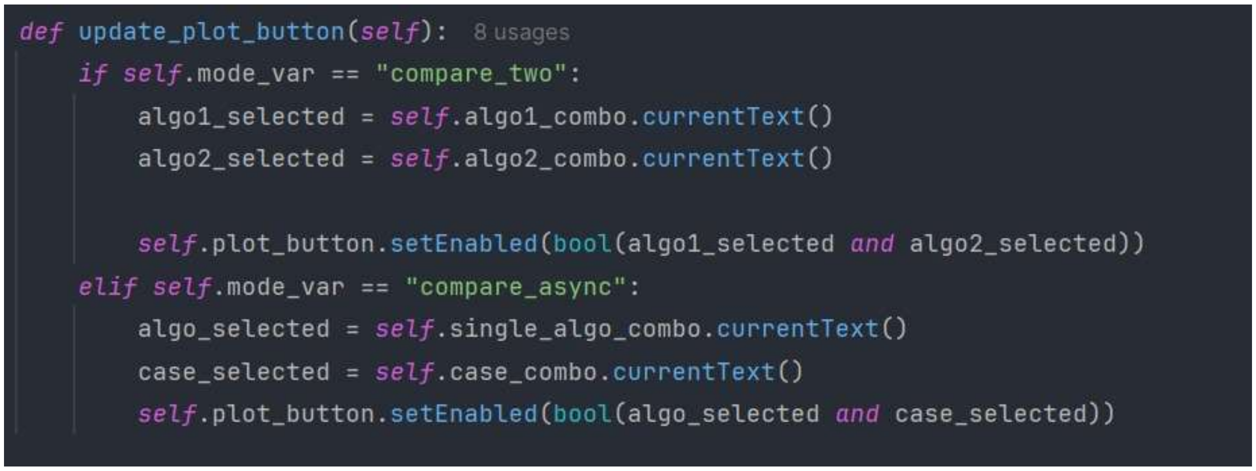
Design Considerations:

- **Error Handling:** The early return (if self.data is None: return) ensures that any issues during data loading do not disrupt the user interface or subsequent operations.
- **User Experience:** By dynamically enabling or disabling UI components based on mode selection, the function enhances interactivity and prevents users from accessing irrelevant options.

5.2 Update Plot Button Logic

The update_plot_button method is designed to dynamically control the state (enabled or disabled) of the "Plot" button in the application's user interface. This functionality ensures that the button is only enabled when the necessary prerequisites for plotting are satisfied, depending on the current mode of operation.

Code:



```python
def update_plot_button(self):  8 usages
    if self.mode_var == "compare_two":
        algo1_selected = self.algo1_combo.currentText()
        algo2_selected = self.algo2_combo.currentText()

        self.plot_button.setEnabled(bool(algo1_selected and algo2_selected))
    elif self.mode_var == "compare_async":
        algo_selected = self.single_algo_combo.currentText()
        case_selected = self.case_combo.currentText()
        self.plot_button.setEnabled(bool(algo_selected and case_selected))
```

Functionality

- **Input Parameters:** The method does not take any external parameters; it operates on the internal state of the class.
- **Modes of Operation:**
    - **compare_two Mode:**
        - The method retrieves the current selections from two dropdown menus (algo1_combo and algo2_combo) that represent two algorithms to be compared.
        - The button is enabled if and only if both algorithms are selected.
    - **compare_async Mode:**
        - The method retrieves the current selections from the single_algo_combo (a dropdown for selecting a single algorithm) and case_combo (a dropdown for selecting a case).
        - The button is enabled if and only if both an algorithm and a case are selected.

Implementation Details

- **State Management:**
  - The method uses boolean expressions to evaluate whether the necessary selections have been made in each mode.
  - The result of this evaluation is passed to the setEnabled method of the plot_button, which determines the button's active state.
- **Dynamic Interactivity:** By reevaluating the state of the button each time this method is called, the UI remains responsive to user actions, preventing invalid operations.

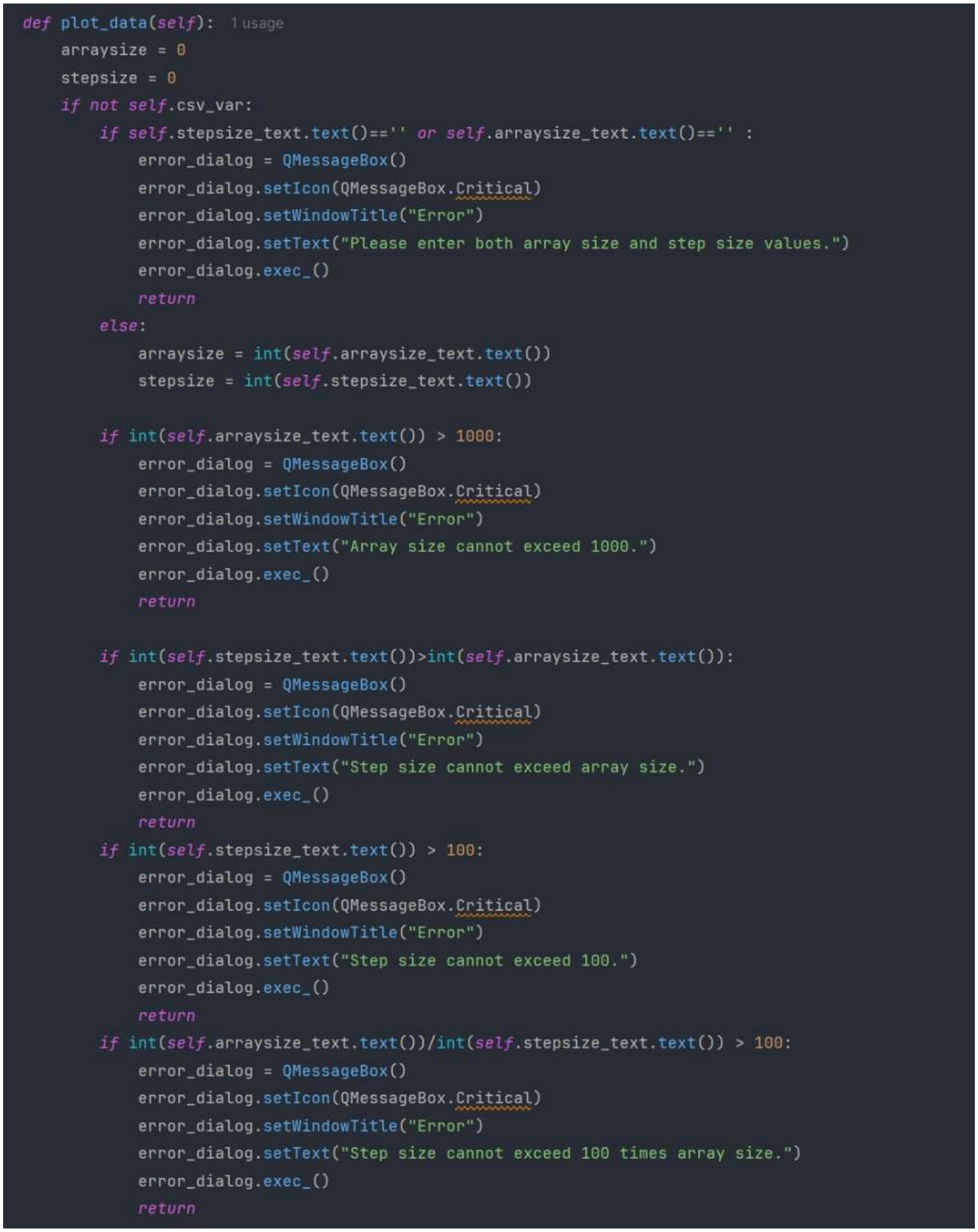
Advantages

- **User Experience:** This implementation improves the user experience by ensuring that the "Plot" button is only active when valid selections are made, reducing the potential for errors.
- **Code Maintainability:** The conditional logic is clearly organized by mode, making the method easy to understand and extend.

5.3 Error Handling

The plot_data method plays a critical role in validating user input and ensuring that the required conditions are met before initiating the data plotting process. It leverages user input to dynamically determine the parameters for plotting, while also implementing error handling to prevent invalid operations.

Code:



```
def plot_data(self):  1 usage
    arraysize = 0
    stepsize = 0
    if not self.csv_var:
        if self.stepsize_text.text()=='' or self.arraysize_text.text()=='' :
            error_dialog = QMessageBox()
            error_dialog.setIcon(QMessageBox.Critical)
            error_dialog.setWindowTitle("Error")
            error_dialog.setText("Please enter both array size and step size values.")
            error_dialog.exec_()
            return
        else:
            arraysize = int(self.arraysize_text.text())
            stepsize = int(self.stepsize_text.text())

        if int(self.arraysize_text.text()) > 1000:
            error_dialog = QMessageBox()
            error_dialog.setIcon(QMessageBox.Critical)
            error_dialog.setWindowTitle("Error")
            error_dialog.setText("Array size cannot exceed 1000.")
            error_dialog.exec_()
            return

        if int(self.stepsize_text.text())>int(self.arraysize_text.text()):
            error_dialog = QMessageBox()
            error_dialog.setIcon(QMessageBox.Critical)
            error_dialog.setWindowTitle("Error")
            error_dialog.setText("Step size cannot exceed array size.")
            error_dialog.exec_()
            return
        if int(self.stepsize_text.text()) > 100:
            error_dialog = QMessageBox()
            error_dialog.setIcon(QMessageBox.Critical)
            error_dialog.setWindowTitle("Error")
            error_dialog.setText("Step size cannot exceed 100.")
            error_dialog.exec_()
            return
        if int(self.arraysize_text.text())/int(self.stepsize_text.text()) > 100:
            error_dialog = QMessageBox()
            error_dialog.setIcon(QMessageBox.Critical)
            error_dialog.setWindowTitle("Error")
            error_dialog.setText("Step size cannot exceed 100 times array size.")
            error_dialog.exec_()
            return
```

Functionality

- **Input Parameters:** The method uses internal state and user input fields for validation:
    - arraysize_text and stepsize_text fields: Text inputs for the array size and step size, respectively.
    - csv_var: A boolean variable indicating whether data is imported from a CSV file.
- **Validation Logic:**
    - The method first checks if the csv_var is false, meaning the user must manually input the arraysize and stepsize.
    - If either input field is empty, an error message dialog is displayed, prompting the user to provide both values.
- **Error Conditions:**
    - **Array Size Exceeds Maximum:** If the array size exceeds 1000, an error message is displayed.
    - **Step Size Exceeds Array Size:** If the step size is greater than the array size, an error message is displayed.
    - **Step Size Exceeds Maximum:** If the step size exceeds 100, the user is notified of the limit.
    - **Step Size-Array Size Ratio:** If the ratio of step size to array size exceeds a threshold of 100 times, an error is triggered.

Implementation Details

- **Dynamic Error Handling:**
    - Error dialogs are implemented using QMessageBox with a critical icon and descriptive text to clearly communicate the issue to the user.

    - The exec_() method ensures that the dialog is modal, requiring user acknowledgment before proceeding.
- **Validation Order:**
    - The checks are performed sequentially, and the process terminates upon encountering the first error. This approach ensures clarity by addressing issues one at a time.

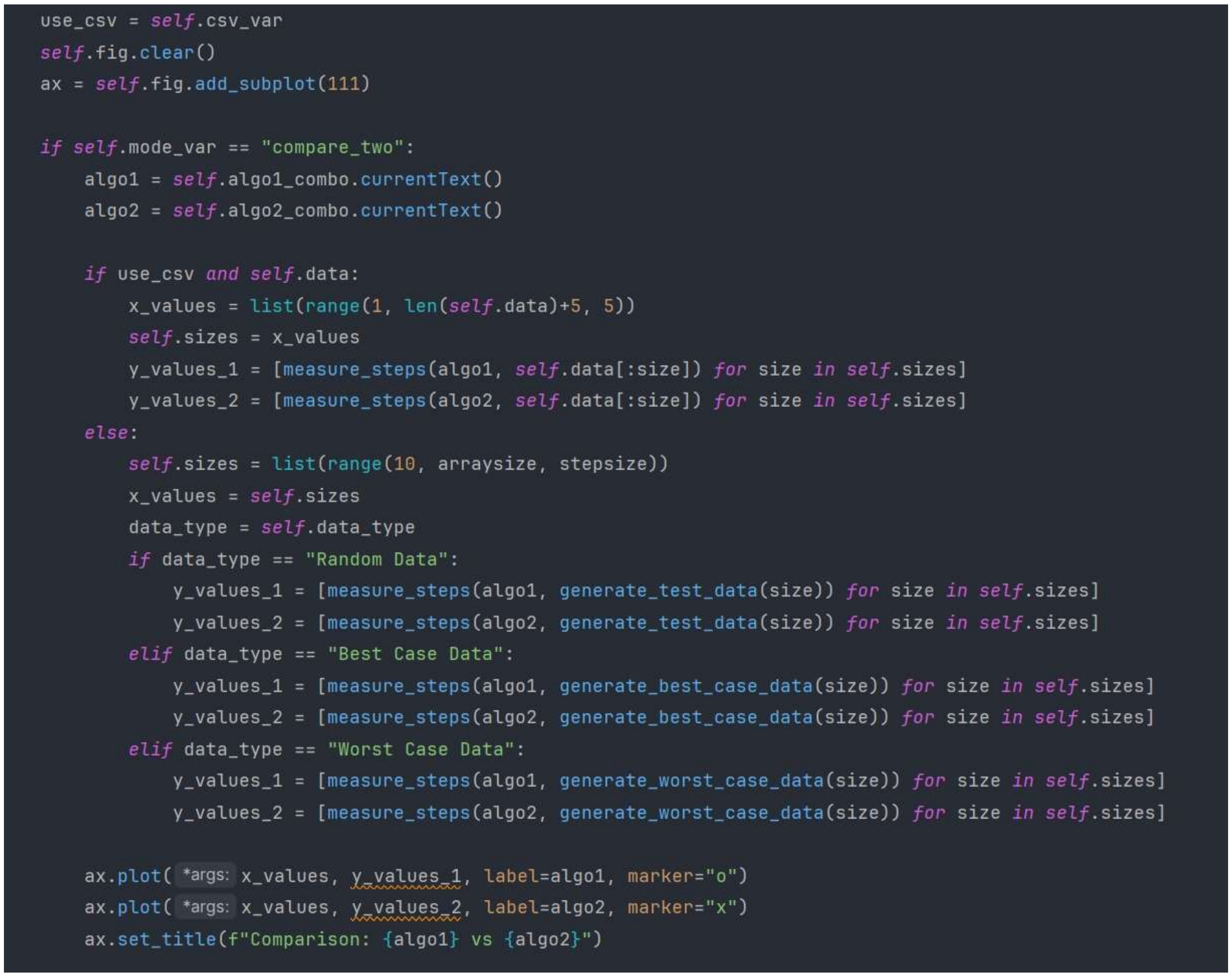
Advantages

- **Robust Input Validation:** Prevents incorrect or nonsensical values from being processed, ensuring the integrity of the plotting operation.
- **User Guidance:** The clear and descriptive error messages help users understand the requirements for valid input.
- **Scalability:** The structured validation approach allows for easy addition of new checks or constraints.

5.4 Plot Two Algorithms

The code implements a method to plot performance comparisons between two algorithms under various scenarios. This functionality visualizes the relationship between data size and algorithm performance, aiding in understanding and analysis.

Code:



```python
use_csv = self.csv_var
self.fig.clear()
ax = self.fig.add_subplot(111)

if self.mode_var == "compare_two":
    algo1 = self.algo1_combo.currentText()
    algo2 = self.algo2_combo.currentText()

    if use_csv and self.data:
        x_values = list(range(1, len(self.data)+5, 5))
        self.sizes = x_values
        y_values_1 = [measure_steps(algo1, self.data[:size]) for size in self.sizes]
        y_values_2 = [measure_steps(algo2, self.data[:size]) for size in self.sizes]
    else:
        self.sizes = list(range(10, arraysize, stepsize))
        x_values = self.sizes
        data_type = self.data_type
        if data_type == "Random Data":
            y_values_1 = [measure_steps(algo1, generate_test_data(size)) for size in self.sizes]
            y_values_2 = [measure_steps(algo2, generate_test_data(size)) for size in self.sizes]
        elif data_type == "Best Case Data":
            y_values_1 = [measure_steps(algo1, generate_best_case_data(size)) for size in self.sizes]
            y_values_2 = [measure_steps(algo2, generate_best_case_data(size)) for size in self.sizes]
        elif data_type == "Worst Case Data":
            y_values_1 = [measure_steps(algo1, generate_worst_case_data(size)) for size in self.sizes]
            y_values_2 = [measure_steps(algo2, generate_worst_case_data(size)) for size in self.sizes]

    ax.plot( *args: x_values, y_values_1, label=algo1, marker="o")
    ax.plot( *args: x_values, y_values_2, label=algo2, marker="x")
    ax.set_title(f"Comparison: {algo1} vs {algo2}")
```

Functionality

- **Input Parameters:**
  - The method derives its input from UI elements such as dropdown menus (algo1_combo and algo2_combo), mode variables (csv_var, mode_var), and pre-set configurations (data, data_type).
- **Modes of Operation:**
  - **compare_two Mode:** The method focuses on comparing two algorithms (algo1 and algo2) based on their performance across varying data sizes.

Implementation Details

1. **Data Source Selection:**
   - If use_csv is True and valid data is available (self.data), the x-values are determined by the size of the CSV data, incremented by a step of 5.
   - Otherwise, the data sizes (self.sizes) are generated based on a predefined range, step size, and array size values.
2. **Data Type Handling:**
   - Depending on the data_type, the method generates specific datasets:
     - **Random Data:** Calls generate_test_data(size) to simulate random inputs.
     - **Best Case Data:** Calls generate_best_case_data(size) for inputs that yield optimal algorithm performance.
     - **Worst Case Data:** Calls generate_worst_case_data(size) for inputs that maximize algorithm complexity.
   - Performance of each algorithm is measured for the generated data using the measure_steps function.
3. **Plot Construction:**
   - Two series of y-values are calculated, corresponding to the performance of algo1 and algo2.
   - These series are plotted against the x-values (data sizes) with distinct markers (o for algo1, x for algo2) for easy differentiation.
   - The plot is given a title indicating the comparison (e.g., "Comparison: Algorithm1 vs Algorithm2").

Advantages

- **Visual Clarity:** The plotted graph allows users to compare the efficiency and scalability of two algorithms intuitively.

- **Customizable Analysis:** By supporting different data types (random, best-case, worst-case), the method provides comprehensive insights into algorithm behavior.
- **Dynamic Inputs:** The integration of CSV data and user-defined configurations ensures flexibility in input data.

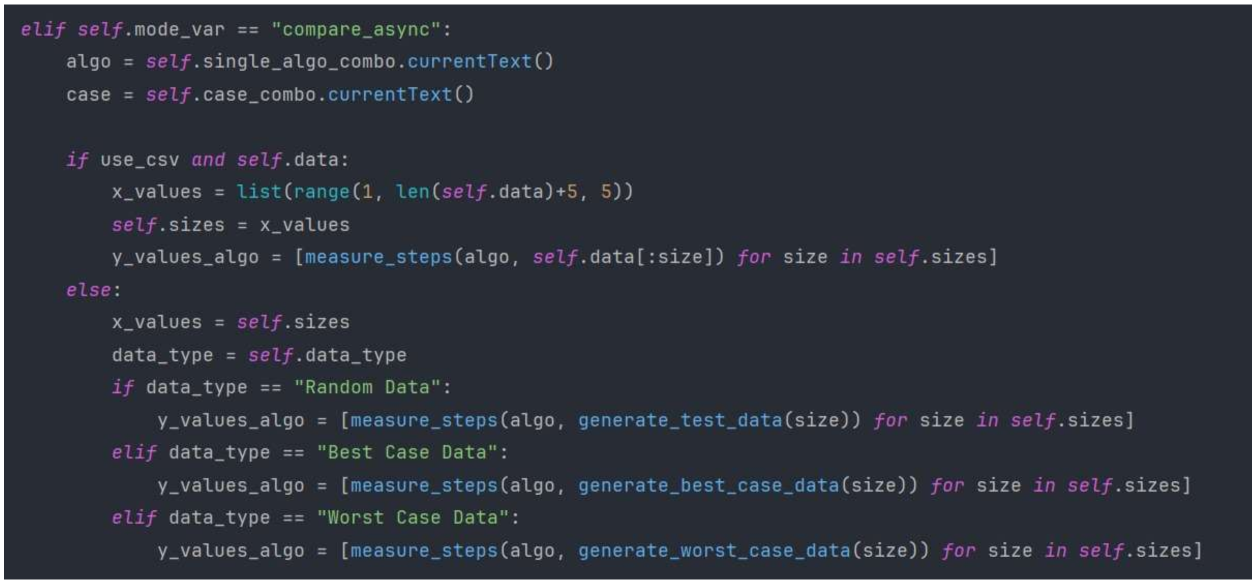
Example Output

The resulting plot would typically display two distinct curves representing the computational steps taken by algo1 and algo2 for varying input sizes, with each data point visually marked for clarity.

5.5 Plot Algorithm & Asymptotic Efficiency

This section of the code focuses on plotting the performance of a single algorithm under varying data scenarios in asynchronous mode (compare_async). The visualization highlights how the algorithm performs across different data sizes and cases.

Code:



```
elif self.mode_var == "compare_async":
    algo = self.single_algo_combo.currentText()
    case = self.case_combo.currentText()

    if use_csv and self.data:
        x_values = list(range(1, len(self.data)+5, 5))
        self.sizes = x_values
        y_values_algo = [measure_steps(algo, self.data[:size]) for size in self.sizes]
    else:
        x_values = self.sizes
        data_type = self.data_type
        if data_type == "Random Data":
            y_values_algo = [measure_steps(algo, generate_test_data(size)) for size in self.sizes]
        elif data_type == "Best Case Data":
            y_values_algo = [measure_steps(algo, generate_best_case_data(size)) for size in self.sizes]
        elif data_type == "Worst Case Data":
            y_values_algo = [measure_steps(algo, generate_worst_case_data(size)) for size in self.sizes]
```

Functionality

- **Input Parameters:**
    - **Algorithm (algo)**: Selected from the single_algo_combo dropdown.

  - **Data Case (case)**: Selected from the case_combo dropdown.
  - **Data Source**: Derived from either a CSV file (self.data) or dynamically generated datasets based on the current data type (self.data_type).
- **Modes of Operation:**
  - **CSV Data Mode:** When use_csv is True and self.data is available, x-values represent indices of the dataset incremented by a step of 5. The y-values are calculated by measuring the performance of the algorithm on progressively larger slices of the dataset.
  - **Generated Data Mode:** If no CSV data is provided, x-values correspond to the predefined sizes (self.sizes). The y-values are computed based on test datasets generated according to the selected data_type.

Implementation Details

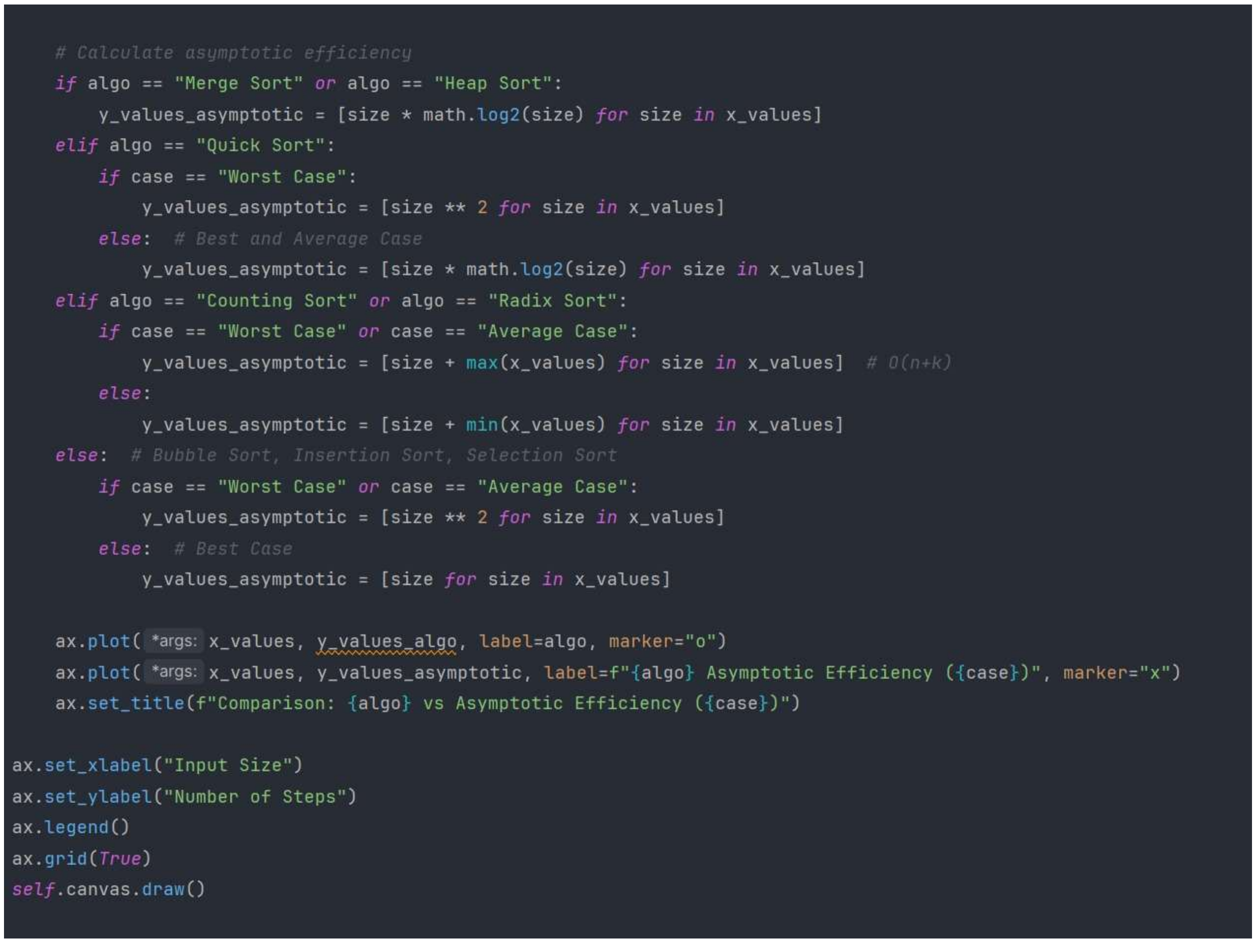
1. **Data Source Determination:**
   - For CSV data, slices of the dataset (self.data[:size]) are used to compute the y-values for the algorithm.
   - For generated data, the method selects between "Random Data," "Best Case Data," or "Worst Case Data" and generates appropriate datasets using helper functions (generate_test_data, generate_best_case_data, generate_worst_case_data).
2. **Performance Measurement:**
   - The measure_steps function evaluates the algorithm's performance for each data size in self.sizes, producing the y-values.
   - This data represents the computational effort required by the algorithm for increasing input sizes.
3. **Plot Data Preparation:**

- x-values represent data sizes, while y-values reflect the algorithm's performance in terms of computational steps.
- These values are prepared for subsequent plotting using the associated ax.plot method outside the provided snippet.

5.6 Asymptotic Efficiency Calculation

This segment of the code evaluates and visualizes the asymptotic efficiency of a selected algorithm in comparison with its actual performance. It provides a theoretical baseline against which the algorithm's real-world behavior is analyzed.

Code:



```
    # Calculate asymptotic efficiency
    if algo == "Merge Sort" or algo == "Heap Sort":
        y_values_asymptotic = [size * math.log2(size) for size in x_values]
    elif algo == "Quick Sort":
        if case == "Worst Case":
            y_values_asymptotic = [size ** 2 for size in x_values]
        else:  # Best and Average Case
            y_values_asymptotic = [size * math.log2(size) for size in x_values]
    elif algo == "Counting Sort" or algo == "Radix Sort":
        if case == "Worst Case" or case == "Average Case":
            y_values_asymptotic = [size + max(x_values) for size in x_values]  # O(n+k)
        else:
            y_values_asymptotic = [size + min(x_values) for size in x_values]
    else:  # Bubble Sort, Insertion Sort, Selection Sort
        if case == "Worst Case" or case == "Average Case":
            y_values_asymptotic = [size ** 2 for size in x_values]
        else:  # Best Case
            y_values_asymptotic = [size for size in x_values]

    ax.plot( *args: x_values, y_values_algo, label=algo, marker="o")
    ax.plot( *args: x_values, y_values_asymptotic, label=f"{algo} Asymptotic Efficiency ({case})", marker="x")
    ax.set_title(f"Comparison: {algo} vs Asymptotic Efficiency ({case})")

ax.set_xlabel("Input Size")
ax.set_ylabel("Number of Steps")
ax.legend()
ax.grid(True)
self.canvas.draw()
```

Functionality

- **Input Parameters:**

- **Algorithm (algo)**: Selected from the single_algo_combo dropdown.
- **Data Case (case)**: Chosen from the case_combo dropdown.
- **Input Sizes (x_values)**: Represent different data sizes for performance evaluation.
- **Actual Performance (y_values_algo)**: Computed from the algorithm's execution steps for varying input sizes.

Implementation Details

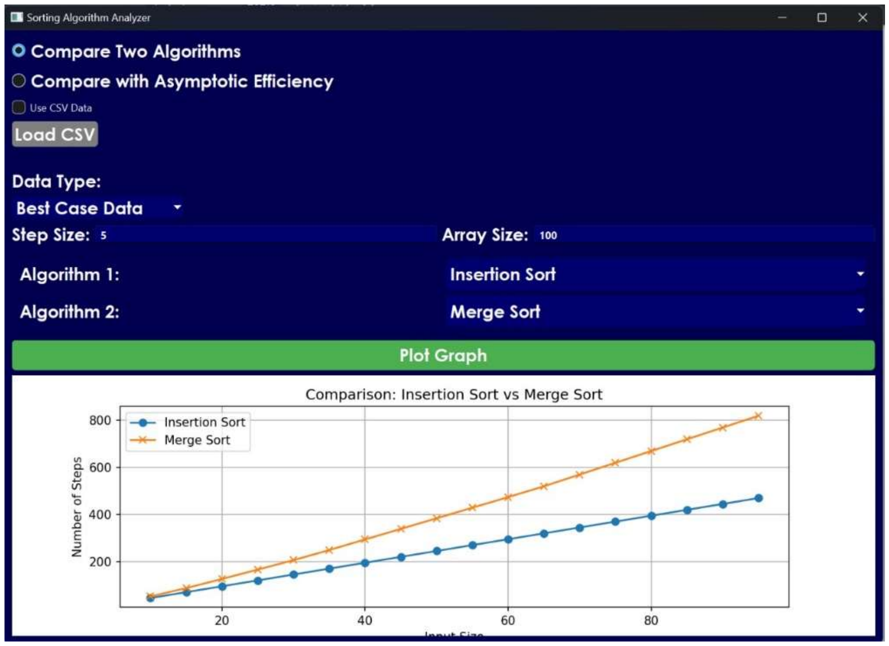
1. **Asymptotic Efficiency Calculation:**
   - For each algorithm, the method calculates theoretical asymptotic complexity values (y_values_asymptotic) based on input sizes (x_values) and the selected data case:
     - **Merge Sort and Heap Sort:** O(nlog⁡n)O(n \log n)O(nlogn)
     - **Quick Sort:** O(n2)O(n^2)O(n2) in the worst case; O(nlog⁡n)O(n \log n)O(nlogn) in average and best cases.
     - **Counting Sort and Radix Sort:** O(n+k)O(n + k)O(n+k), where kkk is the maximum or minimum value in the input data.
     - **Bubble Sort, Insertion Sort, Selection Sort:** O(n2)O(n^2)O(n2) for worst and average cases; O(n)O(n)O(n) for the best case.
   - Mathematical calculations leverage Python's math.log2 function for logarithmic operations.
2. **Plotting Logic:**
   - **Actual Performance Plot:** Displays the algorithm's measured performance (y_values_algo) for various input sizes with circular markers (o).

- **Asymptotic Efficiency Plot:** Visualizes the theoretical efficiency (y_values_asymptotic) using cross markers (x) and an appropriate label indicating the algorithm and data case.
- **Graph Elements:**
  - Title: Includes both the algorithm name and the specific data case being analyzed.
  - Axes Labels: Clearly mark the x-axis as "Input Size" and the y-axis as "Number of Steps."
  - Legend: Differentiates between the actual performance and the asymptotic efficiency curves.
  - Grid: Enhances readability and visual alignment of data points.

3. **Canvas Update:**
   - The self.canvas.draw() method refreshes the plot, ensuring the latest data is displayed in the user interface.

6. Test Cases: -

6.1 Best Case: Insertion VS Merge Sort


Sorting Algorithm Analyzer

[✓ Compare Two Algorithms]

Compare with Asymptotic Efficiency

Use CSV Data

Load CSV

Data Type: Best Case Data

Step Size: 5

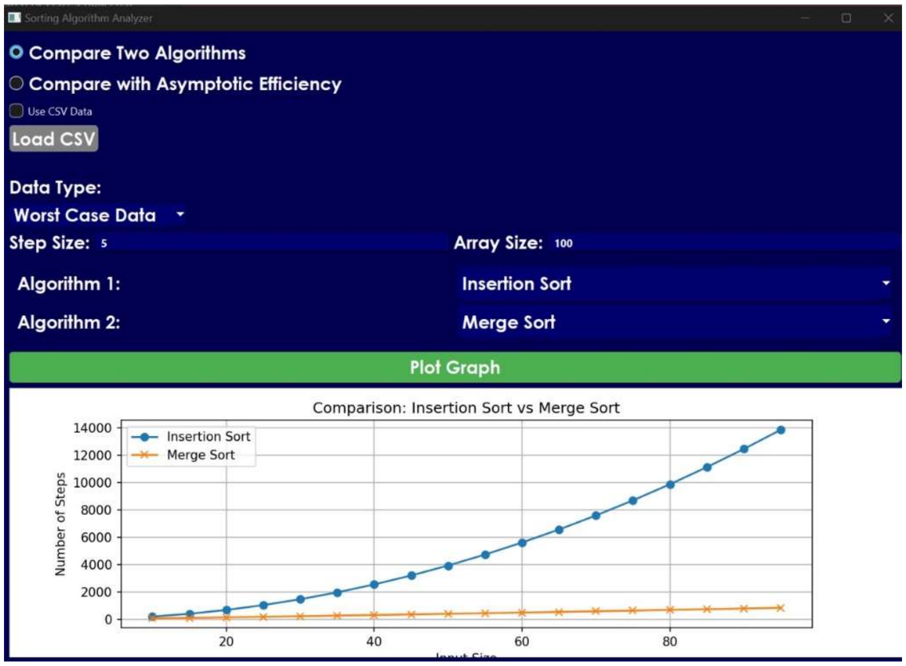
Array Size: 100

Algorithm 1: Insertion Sort

Algorithm 2: Merge Sort

Plot Graph

6.2 Worst Case: Insertion VS Merge Sort



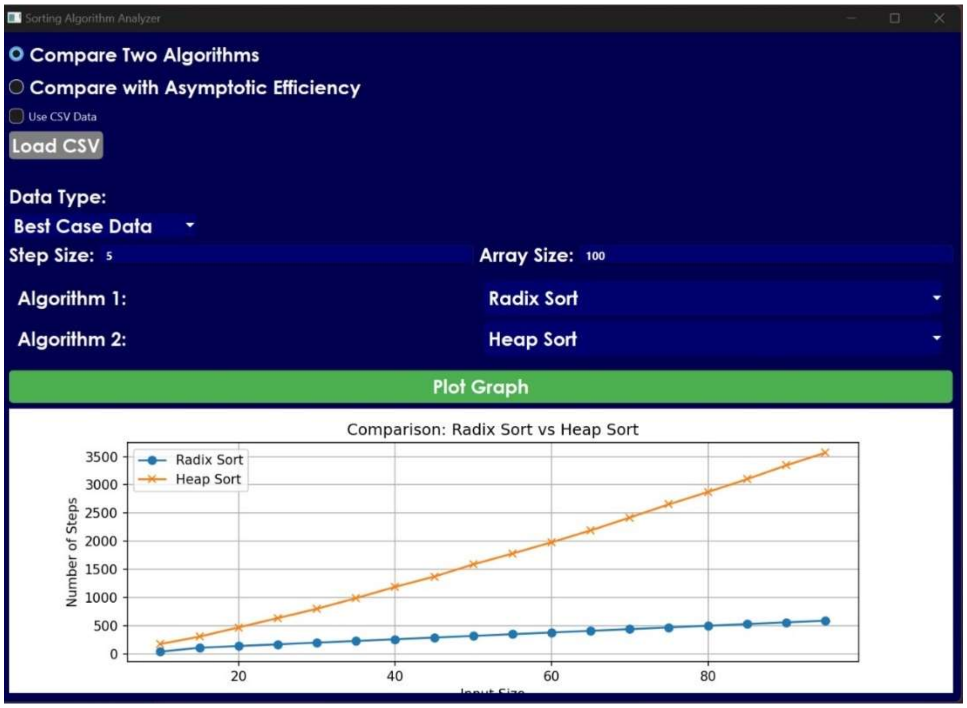
[✓ Compare Two Algorithms]
Compare with Asymptotic Efficiency
Use CSV Data

Data Type:
Worst Case Data

Step Size: 5
Array Size: 100

Algorithm 1: Insertion Sort
Algorithm 2: Merge Sort

6.3 Best Case: Radix VS Heap Sort



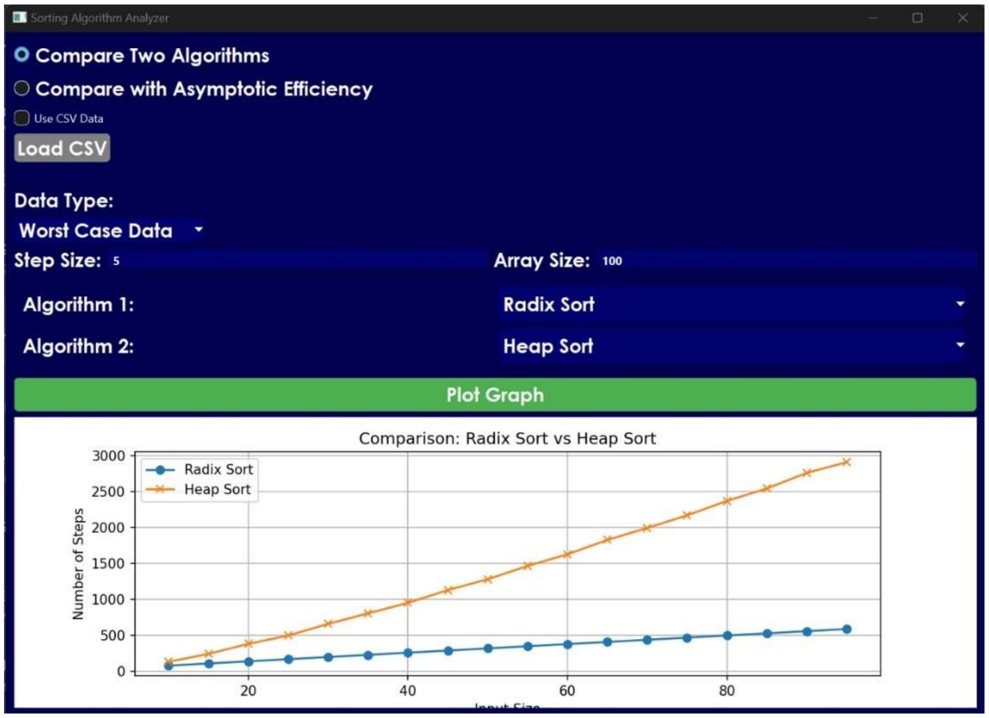
[✓ Compare Two Algorithms]
Compare with Asymptotic Efficiency
Use CSV Data

Data Type:
Best Case Data

Step Size: 5
Array Size: 100

Algorithm 1: Radix Sort
Algorithm 2: Heap Sort

6.4 Worst Case: Radix VS Heap Sort


Sorting Algorithm Analyzer

Compare Two Algorithms [✓]

Compare with Asymptotic Efficiency

Use CSV Data

Load CSV

Data Type: Worst Case Data

Step Size: 5

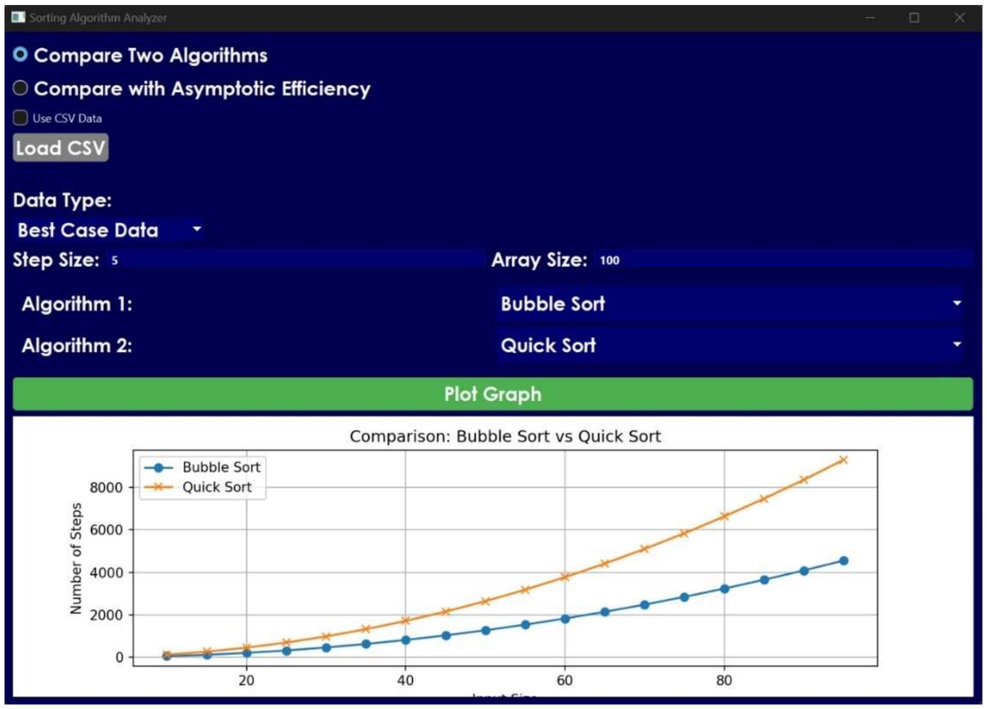
Array Size: 100

Algorithm 1: Radix Sort

Algorithm 2: Heap Sort

Plot Graph

6.5 Best Case: Bubble VS Quick Sort


Sorting Algorithm Analyzer

Compare Two Algorithms [✓]

Compare with Asymptotic Efficiency

Use CSV Data

Load CSV

Data Type: Best Case Data

Step Size: 5

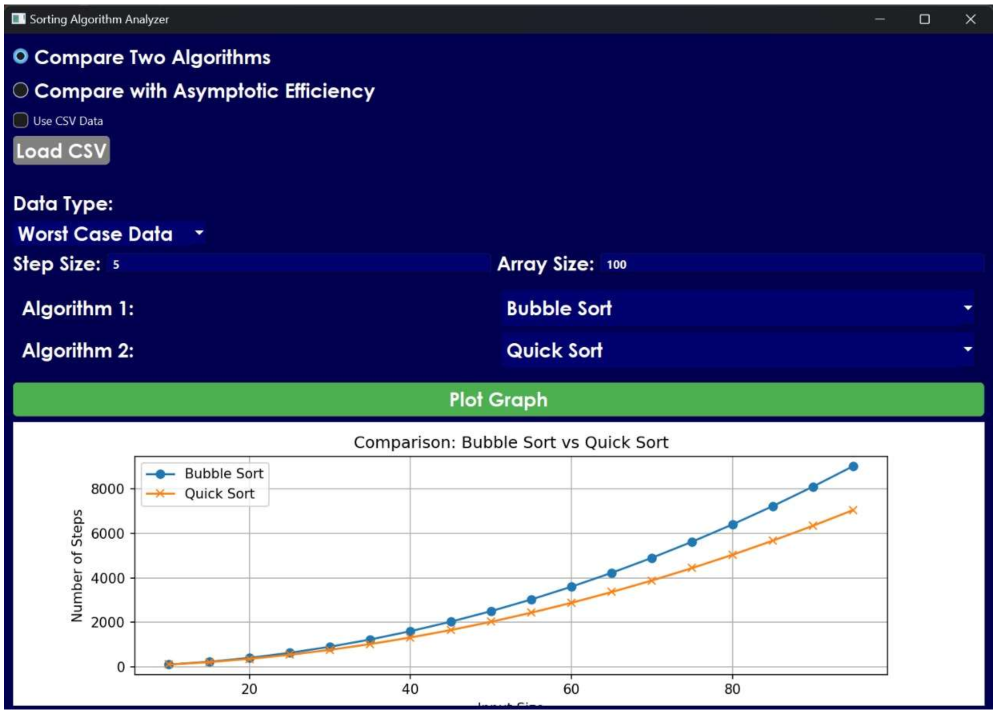
Array Size: 100

Algorithm 1: Bubble Sort

Algorithm 2: Quick Sort

Plot Graph

6.6 Best Case: Bubble VS Quick Sort



[✓ Compare Two Algorithms]

Compare with Asymptotic Efficiency

Use CSV Data

Data Type: Worst Case Data

Step Size: 5

Array Size: 100

Algorithm 1: Bubble Sort

Algorithm 2: Quick Sort

Conclusion:

In this project, we successfully developed an application to compare the efficiency of various sorting algorithms with their respective asymptotic behaviors. The application provides a robust framework for analyzing the performance of sorting algorithms by measuring the number of steps taken to sort test data.

The application allows users to:

- Compare the performance of two algorithms.
- Compare an algorithm's actual performance against its theoretical asymptotic efficiency.

This project achieves the outlined objectives by integrating input validation, flexible data generation, and efficient plotting mechanisms. It serves as a valuable tool for understanding the practical and theoretical aspects of algorithm efficiency, bridging the gap between computational theory and implementation.