**Project 1 (Sorting Algorithms)**

**Implement and compare the following sorting Algorithm**

**Sorting Algorithms:**

Sorting techniques encompass various methods used to organize data within an array or list, arranging them either in ascending or descending order. The optimization of sorting procedures holds significance in enhancing the overall efficiency of complementary algorithms like search and merge methodologies.

This document serves as a comprehensive exploration and analysis of several sorting algorithms, including **Merge Sort, Heap Sort, Quick Sort(all three pivots-1st Element, Last Element, Random Element), Quick Sort using 3 medians, Insertion Sort, Selection Sort, and Bubble Sort**. The practical implementation of these algorithms was executed through the Java programming language in a file named DAAProject.java, housing all the mentioned methodologies. Users are granted the flexibility to control the input methods —either manual entry or random generation, compare multiple algorithms against given data, and provide the input. In CLI , manual input can be facilitated by selecting Option 1, while random input generation can be prompted via Option 2.

The report is supplemented with graphical and tabular illustrations, depicting the relationship between input data size and the time taken for sorting, alongside insights into time complexity, space complexity, and pseudo-code representations. Additionally, attachments containing these visual representations and code details are enclosed within this report.

Please note: Running the code necessitates the Java Development Kit. Compilation can be achieved through:-  
GUI- ' java -jar DAAProject.jar ‘.

CLI- 'javac DAAProject.java,' followed by execution via 'java DAAProject’.

**Algorithms:**

1. **Merge Sort:**

Merge Sort operates as a Divide and Conquer algorithm. It systematically disassembles a list into multiple subsets until each subset comprises a singular element. It subsequently merges these subsets in a manner that systematically constructs a sorted list. The implementation of Merge Sort follows the Top-down approach, ensuring a methodical and efficient sorting process.

**Steps:**

1. The input array undergoes recursive division into halves until it reduces to single elements.
2. When each subset comprises a single element, the sub-arrays are merged using a two-way merge sort approach, resulting in a sorted output array.

**Pseudo-code:**

callMergeSort(arr, left, right):

if left > right

return

mid = (left+right)/2

mergeSortDivide(arr, left, mid)

mergeSortDivide(arr, mid+1, right)

mergeSort(arr, left, mid, right)

**Functions:**

* **mergeSortDivide**: Divides the input array into two halves and recursively calls itself for sorting.
* **mergeSort**: Handles the merging process of the two sorted halves.

**Data Structure:** Arrays

***Comparison of input size and time taken to run the code in tabular form and graphical form:***

A table with numbers and letters

Description automatically generated

**Analysis:** The Merge Sort algorithm performs 'n' comparisons and for each comparison, it entails approximately 'log(n)' comparisons, culminating in an overall time complexity of 'n(logn)'. This average case scenario delineates the performance expectation for Merge Sort. While the best-case scenario assumes a pre-sorted array and the worst-case scenario involves an array arranged in descending order, the time complexity consistently remains 'O(nlog(n))' across all cases.

* **Best-case Complexity of Merge Sort:** *O*(*n*log*n*)
* **Worst-case Complexity of Merge Sort:** *O*(*n*log*n*)
* **Average-case Complexity of Merge Sort:** *O*(*n*log*n*)

The algorithm's efficiency is reflected as *O*(*n*log*n*) regardless of the array's initial arrangement, signifying its consistent performance across different scenarios.

1. **Heap Sort:**

Heap Sort is a comparison-based sorting method reliant on the Binary Heap data structure. The algorithm functions by conceptualizing the array elements as a unique type of complete binary tree known as a 'heap' . In this process, the conversion of the given array into a 'heap' structure is achieved through the method of 'Heapify' , which transforms the array into a heap-form arrangement.

**Steps:**

1. Transform the given input array into a heap structure using the heapify method.
2. Once the array is in heap form, sequentially remove each element from the visualized heap tree and place it at the end of the array. After each removal, re-establish the heap structure by utilizing the heapify method.

**Pseudo-code:**

callHeapSort(A as array):

n = elements\_in(A)

for i = floor(n/2) to 1

Heapify(A, i, n)

for i = n to 1

Heapswaps(A[1], A[i])

n = n - 1

Heapify(A, 1)

Heapify(A as array, i as int, n as int):

left = 2i

right = 2i + 1

if (left <= n) and (A[left] > A[i])

max = left

else

max = i

if (right <= n) and (A[right] > A[max])

max = right

if (max != i)

Heapswaps(A[i], A[max])

Heapify(A, max)

**Functions:**

* **callHeapSort**: An intermediary function managing the initial call to the heapify method and performing the deletion of the root element.
* **Heapify**: This function's purpose is to convert the array into a heap structure.
* **Heapswaps**: Utilized for swapping elements within the input array based on their provided indexes.

**Data Structure:** Arrays, visualization of a binary tree, Heap structure

A table with numbers and a green text

Description automatically generated***Comparison of input size and time taken to run the code in tabular form and graphical form:***

**Analysis:** The **callHeapSort** function involves two distinct loops. One of these loops iterates 'n' times, while the other calls the **Heapify** function, which involves approximately 'log(n)' comparisons. Consequently, the overall time complexity for the Heap Sort algorithm in the average case scenario is 'nlog(n)'. Interestingly, the time complexity remains unaffected by the initial order of the data, whether in an ascending or descending sequence, as the algorithm still executes 'n' loops with the **Heapify** operation.

* **Best-case Complexity of Heap Sort:** *O*(*n*log*n*)
* **Worst-case Complexity of Heap Sort:** *O*(*n*log*n*)
* **Average-case Complexity of Heap Sort:** *O*(*n*log*n*)

The efficiency of the Heap Sort algorithm remains consistent at *O*(*n*log*n*) across various scenarios, reflecting its stability regardless of the initial array arrangement.

1. **Quick Sort:**

Quick Sort functions as a Divide and Conquer algorithm. It selects an element as the pivot and organizes the given array around this pivotal element. The pivotal function within QuickSort is the partition function. The objective is, given an array and an element 'x' within the array as the pivot, to place 'x' in its correct sorted position within the array. All smaller elements (those less than 'x') are arranged before 'x', and all greater elements (those greater than 'x') are positioned after 'x'. This entire process is aimed to be completed within a linear time frame.

***Steps:***

1. Select a pivot from the lowest index, highest index, or any random index within the input

array.

2. Position all elements less than the pivot to its left and elements greater than the pivot to its   
 right, based on the pivot's placement in the input array.

3. Continue this process for all the sub-arrays, recursively applying the same methodology.

***Pseudo-code:***

quickSort(arr[], low, high):

if (low < high):

pi = partition(arr, low, high)

quickSort(arr, low, pi - 1) # Before pi

quickSort(arr, pi + 1, high) # After pi

**Functions:**

* **quicksortElements**: Serves as an intermediary function between the main function and the Quick Sort algorithm.
* **quicksortPartitionInputArray**: Assists in determining the pivot position within the input array, organizing all elements smaller than the pivot to its left.

**Data Structure:** Arrays

***Comparison of input size and time taken to run the code in tabular form and graphical form:***

**Quick Sort Pivot Element as 1st Element**

A table with numbers and a few digits

Description automatically generated with medium confidence

**Quick Sort Pivot Element as Last Element**

A table with numbers and text

Description automatically generatedA table with numbers and text

Description automatically generated**Analysis**: The worst-case time complexity for Quick Sort can reach O(n2) when the array is already sorted in ascending or descending order, or when the pivot is the highest or lowest number in the array. Achieving balanced partitioning on both sides is essential to attain an O(nlogn) time complexity. This balance can be ensured by choosing the median as the pivot, a feature exhibited in the 3 Median Quick Sort algorithm.

**Quick Sort Pivot Element as Random Element**

* Best-case Complexity of Quick Sort: O(nlogn)
* Worst-case Complexity of Quick Sort: O(n2)
* Average-case Complexity of Quick Sort: O(nlogn)

1. **Quick Sort Three Median:**

Quick Sort Three Median is also a Divide and Conquer algorithm. In this variant, the pivot is selected as the median among the leftmost, center, and rightmost elements in the array. By using the median as the pivot, the aim is to address the drawbacks observed in the traditional Quick Sort algorithm.

**Steps:**

1. Choose the pivot as the median among the lowest index, highest index, and center from the input array.
2. Position all elements less than the pivot to its left and elements greater than the pivot to its right based on the pivot's position in the input array.
3. Continue the same process for all the sub-arrays.

**Pseudo-code:**

QuickSortMedian(arr[], low, high):

if (low < high):

median = MedianOfThree (arr, low, high)

position = PartitionMedian (arr, low, high, median)

QuickSortMedian (arr, low, position - 1) # Before position

QuickSortMedian (arr, position + 1, high) # After position

**Functions:**

* **callQuickMedianSort**: An intermediate function coordinating the main function and the Quick Sort algorithm.
* **PartitionMedian**: Determines the pivot's position in the input array and organizes elements less than the pivot to its left and elements greater than the pivot to its right.
* **QuickSortMedian**: Divides the array at the index of partition.
* **SwapMedian**: Facilitates the swapping of elements within the input array based on their provided indexes.
* **MedianOfThree**: Determines the median among the low, high, and center indexes of the array and sorts these three elements in ascending order.

**Data Structure:** Arrays

***Comparison of input size and time taken to run the code in tabular form and graphical form:***

A table with numbers and a few black text

Description automatically generated

***Analysis:***

The time complexity for the Median of Three Quick Sort is similar to the traditional Quick Sort but is expected to have more consistent performance across different cases:

* **Average Case Time Complexity:** O(nlog(n))
* **Best Case Time Complexity:** O(nlog(n))
* **Worst Case Time Complexity:** O(nlog(n))

1. **Insertion Sort:** Insertion sort iterates through an array, comparing each element with the already sorted portion, placing it in the correct position. It repeatedly selects an element, shifts larger elements, and inserts the element into the sorted sequence, creating a fully sorted array.

**Steps:**

1. Starting a loop from the second element (index 1) to the end of the array and select current element as ‘min\_val’.
2. Start an inner loop from the current position back to the start of the array.
3. While the element before 'min\_val' is greater, shift larger elements forward and place ‘min\_val’ in the correct position in sorted sequence. This continues until all elements are sorted in ascending order.

**Pseudo Code:**

callInsertionSort(arrayToSort):

for i in range(1, len(arrayToSort)):

min\_val = arrayToSort[i]

j = i

while j > 0 and arrayToSort[j - 1] > min\_val:

arrayToSort[j] = arrayToSort[j - 1]

j -= 1

arrayToSort[j] = min\_val

**Functions:**

* **callInsertionSort**: Implements the Insertion Sort algorithm. It sorts an array (**arrayToSort**) by iteratively selecting an element and placing it in its correct position within the array, gradually building a sorted sequence. The function accomplishes this by iterating through the array and shifting elements as needed to create an ascending order.

**Data Structure:** Arrays

***Comparison of input size and time taken to run the code in tabular form and graphical form:***

A table with numbers and numbers

Description automatically generated

***Analysis:***

The time complexity of Insertion Sort is influenced by the number of comparisons and shifts needed to place elements in their correct positions within the sorted sequence. Insertion Sort's performance degrades with the increased number of elements and the level of disorder within the initial array:-

1. **Best Case:** When the array is already sorted, Insertion Sort checks each element and realizes they are in the correct order, resulting in a linear time complexity
   * Time Complexity: O(n)
2. **Average Case:** In an average scenario, Insertion Sort involves nested loops to compare and shift elements, resulting in a quadratic time complexity
   * Time Complexity: O(n2)
3. **Worst Case:** When the array is in reverse order, Insertion Sort requires maximum comparisons and shifting operations for each element, resulting in a quadratic time complexity due to its nested loops
   * Time Complexity: O(n2)
4. **Selection Sort:**

Selection sort sorts an array by repeatedly selecting the smallest element and swapping it with the current element, gradually building a sorted sequence. It traverses the array, finding the minimum element and placing it in the correct position, iteratively reducing the unsorted portion.

**Steps:**

1. Traverse the array, setting the current element as the minimum value and its index as the starting index.
2. Iterate from the current position to the end and find the smallest element by comparison. Update the minimum value and its index if a smaller element is found.
3. Swap the current element with the minimum value found. Progress through the array, placing the smallest elements in the sorted portion.

**Pseudo-code:**

callSelectionSort(Array):

for j = 0 to length of Array

minValue = Array[j]

index = j

for i = j to length of Array

if Array[i] < minValue

index = i

minValue = Array[i]

if index is not equal to j

swap(Array[j], Array[index])

**Functions:**

* **callSelectionSort**: Implements the Selection Sort algorithm. It sorts an array (**arrayToSort**) by iteratively finding the minimum element in the unsorted portion and placing it at the beginning of the sorted section. This process continues, gradually building a fully sorted array by selecting the smallest element and swapping it with the current element.

**Data Structure:** Arrays

A screenshot of a table

Description automatically generated***Comparison of input size and time taken to run the code in tabular form and graphical form:***

***Analysis:***

The time complexity of Selection Sort remains the same across all cases:

* + **Best Case Time Complexity**: O(n2)
  + **Average Case Time Complexity**: O(n2)
  + **Worst Case Time Complexity**: O(n2)

1. **Bubble Sort:**

Bubble Sort, a simple sorting algorithm, iterates through the list, comparing adjacent elements and swapping them if they are in the wrong order. It repeats this process until the entire list is sorted, gradually 'bubbling' the largest elements to the end.

**Steps:**

1. Move through the array from the left, comparing adjacent elements, and position the larger element on the right side.
2. By following this approach, the largest element is initially shifted to the far right.
3. Repeat this process iteratively to identify and position the second largest element and continue until the entire dataset is arranged in ascending order.

**Pseudo-code:**

callBubbleSort (array):

pass\_val = 0

while pass\_val < length of array

for i from 0 to length of array - 1

if array[i + 1] < array[i]

temp = array[i]

array[i] = array[i + 1]

array[i + 1] = temp

pass\_val = pass\_val + 1

**Functions:**

* **callBubbleSort**: Employs nested loops to traverse the array, compare adjacent elements, and swap them if they are in the wrong order. This process is repeated until the entire array is sorted in ascending order. The number of passes through the array is controlled by the **pass\_val** variable.

**Data Structure:** Arrays

***Comparison of input size and time taken to run the code in tabular form and graphical form:***

A table with numbers and numbers

Description automatically generated

***Analysis:*** Bubble sort's worst-case and average complexities make it less efficient compared to other   
 sorting algorithms in typical scenarios.

* **Best-Case Complexity:** O(n) , when the list is already sorted
* **Average Complexity:** O(n2) , in a typical scenario where the array is unordered, and multiple passes and swaps are needed
* **Worst-Case Complexity:** O(n2), occurs when the array is sorted in reverse order, and each element needs to be compared and swapped in every pass

**Input against all the sorting algorithms with analysis :-**

A screenshot of a computer

Description automatically generated