**Algo Speedometer Code Documentation.**

**CSE 1325 Sec 005**

**Author Information**

* **Author:** Kenil Patel
* **ID:** 1001866497

**Overview**

The AlgoSpeedometer Java program offers a comprehensive platform to evaluate and compare the runtime performances of various sorting algorithms. Users can select from a range of sorting methods, including Selection Sort, Merge Sort, Bubble Sort, Quick Sort, and Radix Sort. The program enables users to either use default datasets or input their own data, facilitating a detailed exploration of sorting algorithm efficiency.

**Main Class: Algo-Speedometer**

**Methods**

**createAndWriteSortedFile(int[] array, String fileName)**

* **Description:** This method takes in the array that was created by using the file from user and it overwrites with sorted elements.
* **Parameters:**
  + **array**: The array to be written to the file.
  + **fileName**: The name of the file to be created.

**radixSort(int[] arr)**

* **Description:** Implements the Radix Sort algorithm, which efficiently handles integer sorting by distributing elements into buckets based on individual digits.
* **Parameters:**
  + **arr**: The array to be sorted using Radix Sort.

**countingSort(int[] arr, int exp)**

* **Description:** Executes the counting sort algorithm for a specified digit place during the Radix Sort process.
* **Parameters:**
  + **arr**: The array to be sorted.
  + **exp**: The current exponent value.

**getMax(int[] arr)**

* **Description:** Finds and returns the maximum value in the array.
* **Parameters:**
  + **arr**: The array to find the maximum value from.

**bubbleSort(int[] arr)**

* **Description:** Implements the Bubble Sort algorithm, a simple sorting method that repeatedly compares and swaps adjacent elements.
* **Parameters:**
  + **arr**: The array to be sorted using Bubble Sort.

**readFile(String fileName)**

* **Description:** Reads integer data from a file and returns an array.
* **Parameters:**
  + **fileName**: The name of the file to read data from.
* **Returns:** An integer array containing the data read from the file.

**quickSort(int[] array, int low, int high)**

* **Description:** Implements the Quick Sort algorithm, a divide-and-conquer approach that efficiently sorts an array by selecting a pivot element.
* **Parameters:**
  + **array**: The array to be sorted using Quick Sort.
  + **low**: The starting index of the array or sub-array.
  + **high**: The ending index of the array or sub-array.

**partition(int[] array, int low, int high)**

* **Description:** Partitions the array for Quick Sort.
* **Parameters:**
  + **array**: The array to be partitioned.
  + **low**: The starting index of the array or sub-array.
  + **high**: The ending index of the array or sub-array.
* **Returns:** The pivot index.

**swap(int[] array, int i, int j)**

* **Description:** Swaps two elements in the array.
* **Parameters:**
  + **array**: The array in which elements will be swapped.
  + **i**: The index of the first element.
  + **j**: The index of the second element.

**mergeSort(int[] array)**

* **Description:** Implements the Merge Sort algorithm, a highly efficient sorting technique that uses a divide-and-conquer strategy.
* **Parameters:**
  + **array**: The array to be sorted using Merge Sort.

**mergeSort(int[] array, int[] auxiliaryArray, int low, int high)**

* **Description:** Recursive helper function for Merge Sort.
* **Parameters:**
  + **array**: The array to be sorted.
  + **auxiliaryArray**: An auxiliary array used in the merging process.
  + **low**: The starting index of the array or sub-array.
  + **high**: The ending index of the array or sub-array.

**merge(int[] array, int[] auxiliaryArray, int low, int middle, int high)**

* **Description:** Merges two halves of an array in the Merge Sort algorithm.
* **Parameters:**
  + **array**: The array to be sorted.
  + **auxiliaryArray**: An auxiliary array used in the merging process.
  + **low**: The starting index of the array or sub-array.
  + **middle**: The middle index of the array or sub-array.
  + **high**: The ending index of the array or sub-array.

**selectionSort(int[] default\_array)**

* **Description:** Implements the Selection Sort algorithm, which iteratively selects the minimum element and places it in the correct position.
* **Parameters:**
  + **default\_array**: The array to be sorted using Selection Sort.

**print\_10(int[] default\_array)**

* **Description:** Prints the first 10 elements of the array.
* **Parameters:**
  + **default\_array**: The array to be printed.

**create\_array(int size)**

* **Description:** Creates an array of specified sizes by taking input from the user.
* **Parameters:**
  + **size**: The size of the array to be created.
* **Returns:** The array created with user input.

**print\_info(int value)**

* **Description:** Prints information about the runtime characteristics of each sorting algorithm based on the user's choice.
* **Parameters:**
  + **value**: The user's choice corresponding to a sorting algorithm.

**main(String[] args)**

* **Description:** The main function that orchestrates the program. It provides a user-friendly menu for selecting a sorting algorithm and input data source, measures and displays sorting time, and offers insights into the runtime characteristics of the chosen algorithm. To calculate time taken by each sorting algorithm I started the clock first and on the second line I pass the arguments of sorting function and when the function returns, I stop the clock. And to get final time in minutes it is (end time – start time)/6000) this gives us time in minutes. And if we do the same thing for algorithms that do not take long time to sort, I used the dividend to be 1000 to get time in seconds. This code overwrites the file that is provided by user with sorted elements. It is recommended to use a copy. Please be aware that some algorithms like Bubble Sort takes longer time to sort.

**Bigg O time Complexity graphs:**

