Author: Ali Yağız Canıgüroğlu

ID: 1008789881

Section: CMPE223_02

Lecturer: Asst. Prof. Ulaş Güleç

Assignment: 2

1. **Experiment Explanation**: The goal was to analyze sorting algorithms by measuring the elapsed time when applied to arrays of varying sizes and orderings.
2. **Challenges**:
   A. Selecting an appropriate array size for clear visualization; larger arrays (e.g., 200k elements) led to misleading graphs.
   B. External factors, such as RAM usage by other applications and battery charge, significantly impacted the results. Additionally, an electrical short during the experiment caused a drop in computing power, which was only noticed during data transfer to Excel.
   C. Data visualization and transformation: Thanks to my background on Data Science and AI, I used Python (via Kaggle) for data manipulation, collection, and visualization.
3. **Methodology**:
   Helper Methods:
   - **generateArray(*int size*, *String type*):** Generates arrays with specified sizes and types (ascending, descending, or random) to account for best, worst, and average case scenarios in sorting algorithms.
   - **callSortMethod(*int methodNumber*, *int[] array, long id*):** Uses a switch case structure to call the appropriate sorting method.
   - **testSorts(*int[] array*, *String arrayName*, *long id*):** Measures elapsed time and prints results.
4. **Folders**:
   - **Screenshots**: Captured from Eclipse IDE's console.
   - **Excel Tables**: Contain data for array sizes (10k, 20k, 30k, 50k, 100k) with 3 repetitions for each, including error analysis and a "Combined" table.
   - **Plots**: Generated using Python libraries (Pandas, Matplotlib) from the "Combined" table, separated by array types to analyze algorithm behavior across different array orders.

NOTE: All of the experiment data is in the zip file provided.

5. **Analysis Based on the Graphs:**

A. Sort1: Likely Quick Sort

- **Performance Trends**:
  - For **ascending arrays**, Sort1 consistently performs well, with times like 1 for 10k and 11.67 for 100k.
  - For **descending arrays**, performance get worse significantly, rising from 1.33 for 10k to 12.33 for 100k.
  - For **random arrays**, Sort1 is still efficient, maintaining times like 11.33 for 10k and 50 for 100k.
- **Analysis**:
  - These results align with Quick Sort's behavior: it performs well with randomly distributed arrays due to effective pivot choices, but poorly on already sorted or descending arrays if the pivot selection was not optimal. The worst case scenario did not happen thanks to optimal pivot selection.

B. Sort2: Likely Merge Sort

- **Performance Trends**:
  - For **ascending, descending, and random arrays**, the performance is stable and consistent across different input types. Since algorithm splits the array into 2 part recursively, without any external affect like pivot value.
  - Example times include 6 for ascending, 5 for descending, and 4 for random arrays at 10k size.
  - The time grows steadily with input size, e.g., for random arrays: 4 (10k), 7 (20k), and 45 (100k).
- **Analysis**:
  - This **stable behavior** is characteristic of Merge Sort, which consistently achieves O(n*log(n)) regardless of input distribution.
  - **Example**: The consistent performance across all types (e.g., 65.33 for ascending vs. 60.67 for descending and 45 arrays at 100k) highlights Merge Sort's insensitivity to initial array order.

C. Sort3: Likely Bubble Sort

- **Performance Trends**:
  - Sort3 is notably **slower than others**, with massive execution times like 192.33 (10k) to 15682.33 (100k) for descending arrays.
  - Performance for random arrays is catastrophic, e.g., 344.67 (10k) and 46534.67 (100k).
- **Analysis**:
  - The quadratic time complexity $O(n^2)$ of Bubble Sort matches these observations, where small datasets are manageable, but performance degrades exponentially with larger inputs.
  - **Example**: The extreme inefficiency for random arrays (46534.67 for 100k) is a hallmark of Bubble Sort's inability to handle unsorted inputs efficiently.

D. Sort4: Likely Selection Sort

- **Performance Trends**:
  - Sort4 **displays similar behavior across all array types**, with times increasing linearly as input size grows.
  - Examples include: 177 (10k) to 17660.33 (100k) for ascending arrays and 181 (10k) to 17802.33 (100k) for descending arrays.
  - Performance is similarly poor for random arrays: 183,67 (10k) to 17956.67 (100k).

- **Analysis**:
  - This consistency, of the array's initial order, aligns with Selection Sort's $O(n^2)$ complexity, which always examines each element pair.
  - **Example**: The times for descending arrays (17802.33 for 100k) are almost identical to those for ascending arrays (17660.33 for 100k), also for random arrays (17956 for 100k). The results showing Selection Sort's lack of dependency on array order.

E. Sort5: Likely Insertion Sort

- **Performance Trends**:
  - Sort5 excels on **ascending arrays**, achieving near-linear times such as 0.33 (10k) and 1.67 (100k).
  - However, it struggles significantly with **descending** (118.33 at 10k, 11769.33 at 100k) and **random arrays** (78 at 10k, 12928.33 at 100k).

- **Analysis**:
  - Insertion Sort operates at O(n) for already sorted arrays but performs at $O(n^2)$ for reversed or random inputs.
  - **Example**: For descending arrays, the time grows steeply from 118.33 (10k) to 11769.33 (100k), highlighting the algorithm's inefficiency with unsorted data.
  - Note: The worst case scenarios of Insertion and Bubble sort are same, descending order of an array. However, insertion sort is slightly better than for descending order. Reason behind that fact is, bubble sort applies swapping operation for every iteration.