Anexo A: Comparative Performance Analysis of Sorting Algorithms

Listing 1: Comparative Performance Analysis of Sorting Algorithms

```cpp

/*
 * Comparative Performance Analysis of Sorting Algorithms
 *
 * Author: Rafael Passos Domingues
 *     RA: 2023.1.08.036
 *
 * The code at hand performs a performance analysis to compare non-recursive sorting methods
 * and understand their differences. To do so, it builds vectors of varying dimensions, filled
 * with non-repeated random numbers, and sorts them in three ways: in ascending, random and
 * descending order.
 *
 * During sorting, the uses of each vector are counted, allowing a comparison between the methods.
 * The comparison is performed between the three implemented methods, using vector sizes ranging
 * from 100 to 10'000 units at regular intervals of 100 units.
 *
 * The code generates an output file called "sort_usages.csv", containing the data obtained.
 * These values can be used to create graphs that make up the report "comparison_sort_algorithmis.pdf",
 * which is part of this repository.
*/

#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

// Function to perform Bubble Sort
int bubbleSort(int array[], int length) {
  int bubbleUsage = 0; // Counter for Bubble Sort usage
  int i, j;
  for (i = 0; i < length - 1; i++) {
    bubbleUsage += 2; // Increment counter for Bubble Sort usage
    for (j = 0; j < length - i - 1; j++) {
      if (array[j] > array[j + 1]) {
        int tmp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = tmp;
      }
      bubbleUsage += 4;
    }
  }
  return bubbleUsage;
}

// Function to perform Insertion Sort
int insertionSort(int array[], int length) {
  int insertionUsage = 0; // Counter for Insertion Sort usage
  int i, j;
  for (i = 1; i < length; i++) {
    int handle = array[i];
    insertionUsage += 2; // Increment counter for Insertion Sort usage
    for (j = i - 1; j >= 0 && array[j] > handle; j--) {
      array[j + 1] = array[j];
      insertionUsage += 3;
    }
    array[j + 1] = handle;
  }
  return insertionUsage;
}

// Function to perform Selection Sort
int selectionSort(int array[], int length) {
```

```
  int selectionUsage = 0; // Counter for Selection Sort usage
  int i, j;
  for (i = 0; i < length - 1; i++) {
    int minIndex = i;
    selectionUsage += 2; // Increment counter for Selection Sort usage
    for (j = i + 1; j < length; j++) {
      if (array[j] < array[minIndex]) {
        minIndex = j;
      }
    }
    int swap = array[i];
    array[i] = array[minIndex];
    array[minIndex] = swap;
    selectionUsage += 4;
  }
  return selectionUsage;
}

// Function to generate an ordered array and shuffle it
void randomArrayGenerator(int start, int length, int randomArray[]) {
    for (int i = 0; i < length; i++) {
        randomArray[i] = i;
    }

    // shuffle array
    for (int i = length - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = randomArray[i];
        randomArray[i] = randomArray[j];
        randomArray[j] = temp;
    }

    for (int i = 0; i < length; i++) {
        randomArray[i] += start;
    }
}

// Function to generate increasing array
void increasingArrayGenerator(int array[], int start, int length) {
  for (int i = 0; i < length; i++) {
    array[i] = start + i;
  }
}

// Function to generate decreasing array
void decreasingArrayGenerator(int array[], int start, int length) {
  for (int i = 0; i < length; i++) {
    array[i] = start + length - i - 1;
  }
}

/*
 * This function copy the elements of one array to another ensures that each sorting
 * algorithm operates on a separate copy of the original matrix, avoiding interference
 * between algorithm runs.
*/
void copyArray(int source[], int destination[], int length) {
  for (int i = 0; i < length; i++) {
    destination[i] = source[i];
  }
}

// Function to save results to a CSV file
void saveResultsToFile(const string& filename, const int sizes[], const int randomBubbleUsages[],
                       const int increasingBubbleUsages[], const int decreasingBubbleUsages[],
                       const int randomInsertionUsages[], const int increasingInsertionUsages[],
                       const int decreasingInsertionUsages[], const int randomSelectionUsages[],
```

```cpp
                        const int increasingSelectionUsages[], const int decreasingSelectionUsages[],
                        int size) {
  ofstream file(filename);
  if (!file.is_open()) {
    cout << "Failed to create the file." << endl;
    return;
  }

  // Write header to the file
  file << "Array Size" << "," <<
    "Random Bubble Sort Usage" << "," << "Increasing Bubble Sort Usage" << "," << "Decreasing Bubble Sort Usage" << "," <<
    "Random Insertion Sort Usage" << "," << "Increasing Insertion Sort Usage" << "," << "Decreasing Insertion Sort Usage" << "," <<
    "Random Selection Sort Usage" << "," << "Increasing Selection Sort Usage" << "," << "Decreasing Selection Sort Usage" << endl;

  // Write data to the file
  for (int i = 0; i < size; i++) {
    file << sizes[i] << "," <<
      randomBubbleUsages[i] << "," << increasingBubbleUsages[i] << "," << decreasingBubbleUsages[i] << "," <<
      randomInsertionUsages[i] << "," << increasingInsertionUsages[i] << "," << decreasingInsertionUsages[i] << "," <<
      randomSelectionUsages[i] << "," << increasingSelectionUsages[i] << "," << decreasingSelectionUsages[i] << endl;
  }

  file.close();
}

int main(void) {
  const int start = 100;
  const int end = 10000;
  const int step = 100;
  const int numSizes = (end - start) / step + 1;

  int sizes[numSizes];

  int randomBubbleUsages[numSizes];
  int increasingBubbleUsages[numSizes];
  int decreasingBubbleUsages[numSizes];

  int randomInsertionUsages[numSizes];
  int increasingInsertionUsages[numSizes];
  int decreasingInsertionUsages[numSizes];

  int randomSelectionUsages[numSizes];
  int increasingSelectionUsages[numSizes];
  int decreasingSelectionUsages[numSizes];

  int currentSize = start;
  for (int i = 0; i < numSizes; i++) {
    sizes[i] = currentSize;
    currentSize += step;
  }

  for (int i = 0; i < numSizes; i++) {
    int length = sizes[i];

    // Random Array
    int randomArray[length];
    randomArrayGenerator(start, length, randomArray);

    int sortArray[length];

    // Bubble Sort
    copyArray(randomArray, sortArray, length);
    randomBubbleUsages[i] = bubbleSort(sortArray, length);
```

```
    // Insertion Sort
    copyArray(randomArray, sortArray, length);
    randomInsertionUsages[i] = insertionSort(sortArray, length);

    // Selection Sort
    copyArray(randomArray, sortArray, length);
    randomSelectionUsages[i] = selectionSort(sortArray, length);

    // Increasing Array
    increasingArrayGenerator(sortArray, start, length);
    increasingBubbleUsages[i] = bubbleSort(sortArray, length);
    copyArray(randomArray, sortArray, length);
    increasingInsertionUsages[i] = insertionSort(sortArray, length);
    copyArray(randomArray, sortArray, length);
    increasingSelectionUsages[i] = selectionSort(sortArray, length);

    // Decreasing Array
    decreasingArrayGenerator(sortArray, start, length);
    decreasingBubbleUsages[i] = bubbleSort(sortArray, length);
    copyArray(randomArray, sortArray, length);
    decreasingInsertionUsages[i] = insertionSort(sortArray, length);
    copyArray(randomArray, sortArray, length);
    decreasingSelectionUsages[i] = selectionSort(sortArray, length);
  }

  // Save results to a file
  saveResultsToFile("sort_usages.csv", sizes,
    randomBubbleUsages, increasingBubbleUsages, decreasingBubbleUsages,
    randomInsertionUsages, increasingInsertionUsages, decreasingInsertionUsages,
    randomSelectionUsages, increasingSelectionUsages, decreasingSelectionUsages,
    numSizes);

  cout << "File 'sort_usages.csv' created successfully." << endl;

  return 0;
}
```

Anexo B: Kendall's tau-b correlation coefficient

Listing 2: Kendall's tau-b correlation coefficient

```

import numpy as np
import matplotlib.pyplot as plt

# Function to compute Kendall's tau-b correlation coefficient
def kendalls_tau_b(a, b):
    concordantPairs = 0
    discordantPairs = 0
    tiedPairsA = 0
    tiedPairsB = 0

    for i in range(len(a)):
        for j in range(i + 1, len(a)):
            a1, a2 = a[i], a[j]
            b1, b2 = b[i], b[j]

            if a1 == a2 or b1 == b2:
                continue

            if (a1 < a2 and b1 < b2) or (a1 > a2 and b1 > b2):
                concordantPairs += 1
            elif (a1 < a2 and b1 > b2) or (a1 > a2 and b1 < b2):
                discordantPairs += 1

            if a1 == a2 and b1 != b2:
                tiedPairsA += 1

            if b1 == b2 and a1 != a2:
                tiedPairsB += 1

    numerator = concordantPairs - discordantPairs
    denominator = np.sqrt((concordantPairs + discordantPairs + tiedPairsA) *
                          (concordantPairs + discordantPairs + tiedPairsB))
    return numerator / denominator

def main():
    with open("random_array.txt", "r") as inputFile:
        randomArrays = [list(map(int, line.strip().split())) for line in inputFile]

    numVectors = len(randomArrays)

    # Calculate the average Kendall's tau-b coefficient for elements within a vector
    totalIntraRandomness = 0.0
    totalComparisonsIntra = 0

    for i in range(numVectors):
        for j in range(i + 1, numVectors):
            correlation = kendalls_tau_b(randomArrays[i], randomArrays[j])
            totalIntraRandomness += correlation
            totalComparisonsIntra += 1

    averageIntraRandomness = totalIntraRandomness / totalComparisonsIntra

    # Calculate the average Kendall's tau-b coefficient for elements between different vectors
    totalInterRandomness = 0.0
    totalComparisonsInter = 0

    for i in range(1, numVectors):
        correlation = kendalls_tau_b(randomArrays[0], randomArrays[i])
        totalInterRandomness += correlation
        totalComparisonsInter += 1

    averageInterRandomness = totalInterRandomness / totalComparisonsInter

    # Ensure the results are within the range [-1, 1]
    averageIntraRandomness = max(-1.0, min(1.0, averageIntraRandomness))
```

```
    averageInterRandomness = max(-1.0, min(1.0, averageInterRandomness))

    print("Average Kendall's tau-b coefficient for elements within a vector:",
        averageIntraRandomness)
    print("Average Kendall's tau-b coefficient for elements between different vectors:",
        averageInterRandomness)

if __name__ == "__main__":
    main()
```

Anexo C: Best Method

Listing 3: Best Method

```python

import csv
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def normalize_data(data):
    max_value = max(data)
    min_value = min(data)
    return [(value - min_value) / (max_value - min_value) for value in data]

def main():
    sizes = []
    randomBubbleUsages = []
    increasingBubbleUsages = []
    decreasingBubbleUsages = []
    randomInsertionUsages = []
    increasingInsertionUsages = []
    decreasingInsertionUsages = []
    randomSelectionUsages = []
    increasingSelectionUsages = []
    decreasingSelectionUsages = []

    # Read the CSV File
    data = pd.read_csv("sort_usages.csv")

    # Extract the data of columns
    sizes = data["Array␣Size"]
    randomBubbleUsages = data["Random␣Bubble␣Sort␣Usage"]
    increasingBubbleUsages = data["Increasing␣Bubble␣Sort␣Usage"]
    decreasingBubbleUsages = data["Decreasing␣Bubble␣Sort␣Usage"]
    randomInsertionUsages = data["Random␣Insertion␣Sort␣Usage"]
    increasingInsertionUsages = data["Increasing␣Insertion␣Sort␣Usage"]
    decreasingInsertionUsages = data["Decreasing␣Insertion␣Sort␣Usage"]
    randomSelectionUsages = data["Random␣Selection␣Sort␣Usage"]
    increasingSelectionUsages = data["Increasing␣Selection␣Sort␣Usage"]
    decreasingSelectionUsages = data["Decreasing␣Selection␣Sort␣Usage"]

    # Normalize the data
    sizes = normalize_data(sizes)
    randomBubbleUsages = normalize_data(randomBubbleUsages)
    increasingBubbleUsages = normalize_data(increasingBubbleUsages)
    decreasingBubbleUsages = normalize_data(decreasingBubbleUsages)
    randomInsertionUsages = normalize_data(randomInsertionUsages)
    increasingInsertionUsages = normalize_data(increasingInsertionUsages)
    decreasingInsertionUsages = normalize_data(decreasingInsertionUsages)
    randomSelectionUsages = normalize_data(randomSelectionUsages)
    increasingSelectionUsages = normalize_data(increasingSelectionUsages)
    decreasingSelectionUsages = normalize_data(decreasingSelectionUsages)

    # Calculate the mean for each method and create a dictionary
    method_means = {
        "Random␣Bubble": sum(randomBubbleUsages) / len(randomBubbleUsages),
        "Increasing␣Bubble": sum(increasingBubbleUsages) / len(increasingBubbleUsages),
        "Decreasing␣Bubble": sum(decreasingBubbleUsages) / len(decreasingBubbleUsages),
        "Random␣Insertion": sum(randomInsertionUsages) / len(randomInsertionUsages),
        "Increasing␣Insertion": sum(increasingInsertionUsages) / len(increasingInsertionUsages)
            ,
        "Decreasing␣Insertion": sum(decreasingInsertionUsages) / len(decreasingInsertionUsages)
            ,
        "Random␣Selection": sum(randomSelectionUsages) / len(randomSelectionUsages),
        "Increasing␣Selection": sum(increasingSelectionUsages) / len(increasingSelectionUsages)
            ,
        "Decreasing␣Selection": sum(decreasingSelectionUsages) / len(decreasingSelectionUsages)
            ,
```

```
    }

    # Sort the method_means dictionary by mean values (ascending order)
    sorted_means = sorted(method_means.items(), key=lambda x: x[1])

    # Unpack the sorted_means into two separate lists
    sorted_methods, sorted_means = zip(*sorted_means)

    # Define colors for each method
    colors = {"Random Bubble": "blue", "Increasing Bubble": "blue", "Decreasing Bubble": "blue",
              "Random Insertion": "green", "Increasing Insertion": "green", "Decreasing Insertion": "green",
              "Random Selection": "red", "Increasing Selection": "red", "Decreasing Selection": "red"}

    # Create a pie chart showing the percentage of usage for each method
    plt.figure(figsize=(8, 8))

    # Define the explode values to highlight the method of lowest usage
    explode = [0.1 if method == sorted_methods[0] else 0 for method in sorted_methods]

    plt.pie(sorted_means, labels=sorted_methods, colors=[colors[method] for method in sorted_methods],
            autopct="%1.4f%%", startangle=140, explode=explode)

    plt.title("Percentage of Usage for Each Sorting Algorithm")
    plt.axis("equal")

    plt.show()

    # Calculate the mean normalized usage for each size
    mean_usage_insertion = (np.array(randomInsertionUsages) + np.array(increasingInsertionUsages) + np.array(decreasingInsertionUsages)) / 3

    # Create a scatter plot with regression line for the mean normalized usage (insertion method)
    sns.set_theme(style="whitegrid")
    plt.figure(figsize=(10, 6))
    plt.scatter(sizes, randomBubbleUsages, label="Random Bubble", s=30, color=colors["Random Bubble"], alpha=0.1)
    plt.scatter(sizes, increasingBubbleUsages, label="Increasing Bubble", s=30, color=colors["Increasing Bubble"], alpha=0.1)
    plt.scatter(sizes, decreasingBubbleUsages, label="Decreasing Bubble", s=30, color=colors["Decreasing Bubble"], alpha=0.1)
    plt.scatter(sizes, randomSelectionUsages, label="Random Selection", s=30, color=colors["Random Selection"], alpha=0.1)
    plt.scatter(sizes, increasingSelectionUsages, label="Increasing Selection", s=30, color=colors["Increasing Selection"], alpha=0.1)
    plt.scatter(sizes, decreasingSelectionUsages, label="Decreasing Selection", s=30, color=colors["Decreasing Selection"], alpha=0.1)

    plt.plot(sizes, mean_usage_insertion, label="Mean Normalized Usage (Insertion)", color="green", linewidth=2)

    plt.xlabel("Normalized Array Size")
    plt.ylabel("Normalized Usage")
    plt.title("Normalized Usage for Each Sorting Algorithm")
    plt.legend(loc="upper left")

    # Adjust y-axis limits for better visualization
    plt.ylim(-0.1, 1.1)

    plt.show()

# Run the main function
if __name__ == "__main__":
    main()
```