Análise Comparativa de Desempenho de Algoritmos de Ordenação

Rafael Passos Domingues

I. Introdução

A ordenação numa aplicação computacional é o processo de organizar dados numa determinada ordem. Atualmente, vivemos uma era de grandes volumes de dados e, para isso, a ordenação é essencial, especialmente se tratando de processamento de dados em tempo real [Friedman, 1977].

A ordenação melhora a eficiência da pesquisa de dados específicos dados no computador. Há muitas técnicas sofisticadas para ordenação: Nesta análise, serão investigados os algoritmos de ordenação *Bubble Sort*, *Selection Sort* e *Insertion Sort* aplicados a três distindos arranjos de vetores: em ordem crescente, decrescente e aleatório.

Foram realizados testes para analisar o desempenho de duas implementações distintas para a função *randomArrayGenerator*: Na primeira abordagem, os vetores foram gerados aleatoriamente, elemento a elemento, garantindo que não houvesse repetição entre os elementos. Porém, essa implementação demandou um tempo de execução excessivamente longo para vetores com mais de 100 elementos, o que impossibilitou a obtenção de resultados.

Por outro lado, a segunda abordagem consistiu em gerar um vetor inicialmente ordenado e, em seguida, embaralhar aleatoriamente as posições dos elementos. Essa implementação se mostrou muito mais eficiente para a análise realizada. Essa abordagem foi testada estatisticamente aplicando o método de *coeficientes Kendall's tau b* [Xu, W.; 2013] que apresentou uma anti-correlação média[1] de -0.0034 entre elementos de cada vetor e 0.027 entre os vetores, variando o tamanho do vetor de 100 a 10000, em incrementos de 100 unidades, contidos num intervalo de 0 a 1000 de magnitude, valor ótimo com relação ao transiente de aleatoriedade.

O desempenho desses algoritmos será avaliado com base no número de iterações necessários em cada um dos métodos. No algoritmo foram implementados contadores que acumulam a contagem de cada troca de endereçamento dos elementos, variando o tamanho dos vetores de 100 a 10000, a partir de um incremento de 200 unidades de tamanho.

II. Referencial Teórico

Existem várias métricas que podem ser consideradas para avaliar o desempenho de um algoritmo de ordenação, como o tempo de execução, o consumo de memória e o número de operações realizadas. Também existem vários tipos de algoritmos de ordenação. Cada métrica e algoritmo possui vantagens e desvantagens em relação ao desempenho, dependendo das características dos dados a serem ordenados [Mannila, 1985]; [Chauhan, 2020]; [Harris, 2009].

A seguir, são abordados os métodos a serem explorados neste trabalho.

- **Bubble Sort:** O *Bubble Sort* é um algoritmo que compara repetidamente pares adjacentes de elementos e os troca se estiverem na ordem errada. Esse processo é repetido várias vezes até que a lista esteja totalmente ordenada. O nome *"Bubble"* vem do fato de que os elementos maiores *"borbulham"* para o final da lista durante as comparações e trocas. [Al-Kharabsheh, 2013]
- **Insertion Sort:** O Insertion Sort constrói a lista ordenada um elemento por vez. Ele divide a lista em uma parte ordenada e uma parte não ordenada. Em cada iteração, um elemento da parte não ordenada é selecionado e inserido na posição correta da parte ordenada. [Prajapati, 2017].
- **Selection Sort:** O Selection Sort seleciona repetidamente o menor elemento da lista não ordenada e o coloca no início da lista ordenada. Ele divide a lista em uma parte ordenada e uma parte não ordenada, assim como o Insertion Sort. Em cada iteração, o elemento mínimo da parte não ordenada é identificado e trocado com o primeiro elemento da parte não ordenada [Sedgewick,2011].

III. Material Utilizado

Para realizar esta análise, foi utilizado um dos computadores do laboratório do BCC-UNIFAL com sistema operacional Linux Ubuntu 20.04 LTS. O código foi implementado em C++ e compilado através do ambiente *NetBeans 13*. Por simplicidade, os gráficos foram plotados utilizando as bibliotecas *pandas* e *matplotlib* do Python 3.9 a partir da IDE *VSCode*.

IV. Métodos Implementados

O código *main.cc* em questão implementa os seguintes métodos:

- Inclusão das bibliotecas necessárias: `iostream`, `fstream`, para criar o arquivo de saída `ctime`, para gerar os vetores aleatórios.
- Definição das funções de ordenação: `bubbleSort`, `insertionSort` e `selectionSort`.
- Definição da função `randomArrayGenerator` para gerar um array aleatório a partir de uma sequência ordenada embaralhada aleatoriamente.
- Definição da função `copyArray` para reconstruir os vetores originais e manipulá-los livremente nas chamadas de função, gerando seus distintos arranjos necessários para a análise.

[1] O coeficiente de *anti-correlação* perfeita = -1

- Definição da função `saveResultsToFile` para salvar os resultados em um arquivo CSV.
- Implementação da função principal (`main`) que realiza as seguintes ações:
  1) Definição de constantes para determinar o tamanho inicial, tamanho final e incremento. *start*, *length* e *step*.
  2) Declaração dos arrays e variáveis necessários para armazenar os resultados: foi criado um vetor *sizes[length]* para armazenar os distintos tamanhos de vetor.
  3) Execução dos algoritmos de ordenação para diferentes tamanhos de arrays (aleatório, crescente e decrescente).
  4) Armazenamento do número de utilizações realizadas em cada algoritmo de ordenação nas variáveis *Usage* para os distintos arranjos de vetores.
  5) Chamada da função `saveResultsToFile` para salvar os resultados em um arquivo *CSV* chamado "sort_usages.csv".

V. Resultados Obtidos

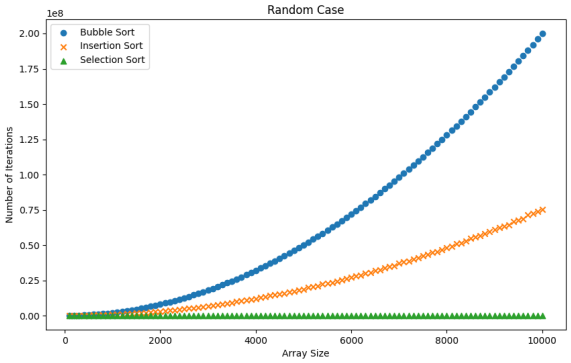

Fig. 1. Número de utilizações dos métodos de ordenação para o caso de vetores aleatórios.

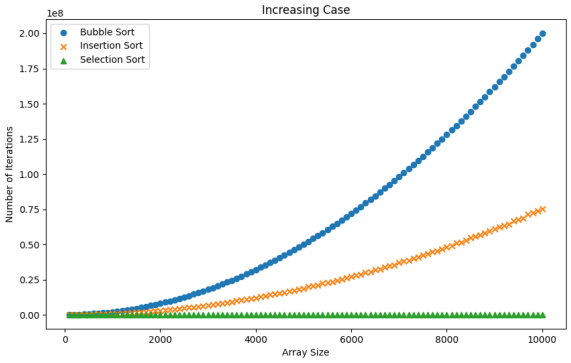

Fig. 2. Número de utilizações dos métodos de ordenação para o caso de vetores crescentes.

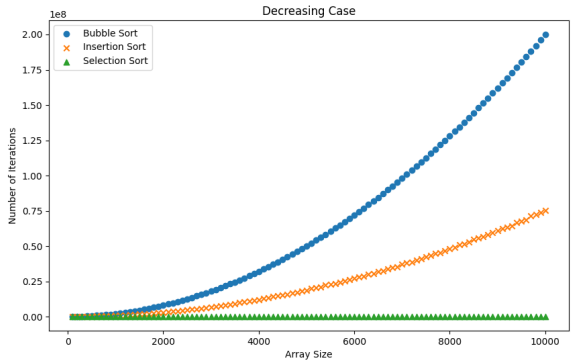

Fig. 3. Número de utilizações dos métodos de ordenação para o caso de vetores decrescentes.

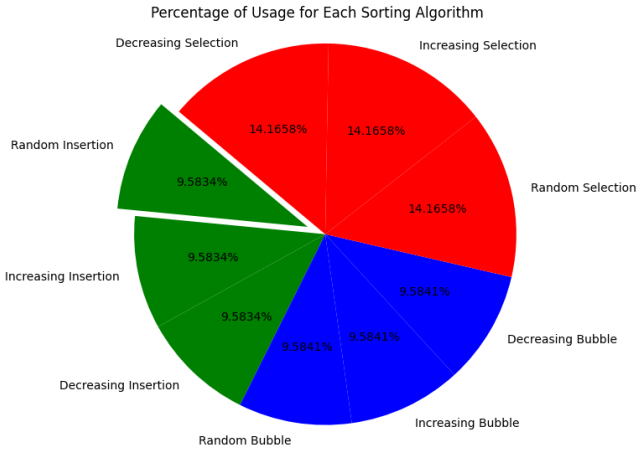

Fig. 4. Os dados de contagens de uso foram normalizados para garantir uma comparação justa entre os métodos de modo que foi possível plotar o percentual de uso de cada método em um gráfico de pizza, destacando o método menos utilizado.

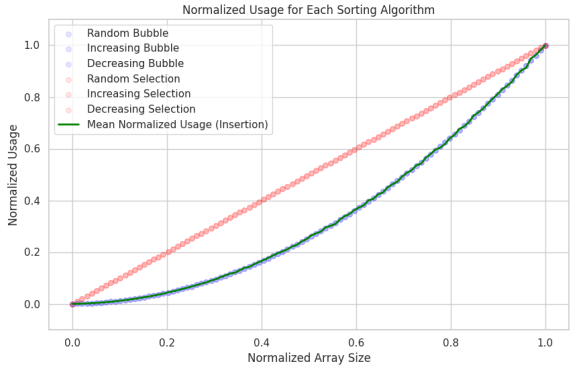

Fig. 5. O gráfico mostra a tendência das médias normalizadas para todos os casos, onde a linha verde é a curva ótima, representando o comportamento do algoritmo *Insertion Sort*.

VI. Conclusão

Com base nos resultados obtidos, foi possível concluir que a escolha do método de ordenação adequado depende diretamente do arranjo do vetor. Os resultados indicam que os algoritmos *Bubble Sort* e *Insertion Sort* se mostraram mais performáticos, com *Insertion Sort* ganhando na quarta casa decimal dentre as médias percentuais , sendo então o melhor método para os três distintos arranjos de vetor.

VII. Referências Bibliográficas

1) Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3), 209-226.
2) Mannila, H. (1985). Measures of presortedness and optimal sorting algorithms. IEEE transactions on computers, 100(4), 318-325.
3) Chauhan, Y., & Duggal, A. (2020). Different sorting algorithms comparison based upon the time complexity. International Journal Of Research And Analytical Reviews, (3), 114-121.
4) Xu, W., Hou, Y., Hung, Y. S., & Zou, Y. (2013). A comparative analysis of Spearman's rho and Kendall's tau in normal and contaminated normal models. Signal Processing, 93(1), 261-276.
5) Satish, N., Harris, M., & Garland, M. (2009, May). Designing efficient sorting algorithms for manycore GPUs. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-10). IEEE.
6) Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I., & Zanoon, N. I. (2013). Review on sorting algorithms a comparative study. International Journal of Computer Science and Security (IJCSS), 7(3), 120-126.
7) Prajapati, P., Bhatt, N., & Bhatt, N. (2017). Performance comparison of different sorting algorithms. vol. VI, no. Vi, 39-41.
8) Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley Professional.

Anexo A: Comparative Performance Analysis of Sorting Algorithms

Listing 1: Comparative Performance Analysis of Sorting Algorithms

```

/*
 * 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
                           [],
```

```
                        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);
```

```
194
195     // Insertion Sort
196     copyArray(randomArray, sortArray, length);
197     randomInsertionUsages[i] = insertionSort(sortArray, length);
198
199     // Selection Sort
200     copyArray(randomArray, sortArray, length);
201     randomSelectionUsages[i] = selectionSort(sortArray, length);
202
203     // Increasing Array
204     increasingArrayGenerator(sortArray, start, length);
205     increasingBubbleUsages[i] = bubbleSort(sortArray, length);
206     copyArray(randomArray, sortArray, length);
207     increasingInsertionUsages[i] = insertionSort(sortArray, length);
208     copyArray(randomArray, sortArray, length);
209     increasingSelectionUsages[i] = selectionSort(sortArray, length);
210
211     // Decreasing Array
212     decreasingArrayGenerator(sortArray, start, length);
213     decreasingBubbleUsages[i] = bubbleSort(sortArray, length);
214     copyArray(randomArray, sortArray, length);
215     decreasingInsertionUsages[i] = insertionSort(sortArray, length);
216     copyArray(randomArray, sortArray, length);
217     decreasingSelectionUsages[i] = selectionSort(sortArray, length);
218   }
219
220   // Save results to a file
221   saveResultsToFile("sort_usages.csv", sizes,
222     randomBubbleUsages, increasingBubbleUsages, decreasingBubbleUsages,
223     randomInsertionUsages, increasingInsertionUsages, decreasingInsertionUsages,
224     randomSelectionUsages, increasingSelectionUsages, decreasingSelectionUsages,
225     numSizes);
226
227   cout << "File␣'sort_usages.csv'␣created␣successfully." << endl;
228
229   return 0;
230 }
```

Anexo B: Kendall's tau-b correlation coefficient

Listing 2: Kendall's tau-b correlation coefficient

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Function to compute Kendall's tau-b correlation coefficient
6  def kendalls_tau_b(a, b):
7      concordantPairs = 0
8      discordantPairs = 0
9      tiedPairsA = 0
10     tiedPairsB = 0
11
12     for i in range(len(a)):
13         for j in range(i + 1, len(a)):
14             a1, a2 = a[i], a[j]
15             b1, b2 = b[i], b[j]
16
17             if a1 == a2 or b1 == b2:
18                 continue
19
20             if (a1 < a2 and b1 < b2) or (a1 > a2 and b1 > b2):
21                 concordantPairs += 1
22             elif (a1 < a2 and b1 > b2) or (a1 > a2 and b1 < b2):
23                 discordantPairs += 1
24
25             if a1 == a2 and b1 != b2:
26                 tiedPairsA += 1
27
28             if b1 == b2 and a1 != a2:
29                 tiedPairsB += 1
30
31     numerator = concordantPairs - discordantPairs
32     denominator = np.sqrt((concordantPairs + discordantPairs + tiedPairsA) *
33                           (concordantPairs + discordantPairs + tiedPairsB))
34     return numerator / denominator
35
36 def main():
37     with open("random_array.txt", "r") as inputFile:
38         randomArrays = [list(map(int, line.strip().split())) for line in inputFile]
39
40     numVectors = len(randomArrays)
41
42     # Calculate the average Kendall's tau-b coefficient for elements within a vector
43     totalIntraRandomness = 0.0
44     totalComparisonsIntra = 0
45
46     for i in range(numVectors):
47         for j in range(i + 1, numVectors):
48             correlation = kendalls_tau_b(randomArrays[i], randomArrays[j])
49             totalIntraRandomness += correlation
50             totalComparisonsIntra += 1
51
52     averageIntraRandomness = totalIntraRandomness / totalComparisonsIntra
53
54     # Calculate the average Kendall's tau-b coefficient for elements between different vectors
55     totalInterRandomness = 0.0
56     totalComparisonsInter = 0
57
58     for i in range(1, numVectors):
59         correlation = kendalls_tau_b(randomArrays[0], randomArrays[i])
60         totalInterRandomness += correlation
61         totalComparisonsInter += 1
62
63     averageInterRandomness = totalInterRandomness / totalComparisonsInter
64
65     # Ensure the results are within the range [-1, 1]
66     averageIntraRandomness = max(-1.0, min(1.0, averageIntraRandomness))
```

```
67      averageInterRandomness = max(-1.0, min(1.0, averageInterRandomness))
68
69      print("Average␣Kendall's␣tau-b␣coefficient␣for␣elements␣within␣a␣vector:",
            averageIntraRandomness)
70      print("Average␣Kendall's␣tau-b␣coefficient␣for␣elements␣between␣different␣vectors:",
            averageInterRandomness)
71
72  if __name__ == "__main__":
73      main()
```

Anexo C: Best Method

Listing 3: Best Method

```

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()
```