

# Anexo A: Comparative Performance Analysis of Sorting Algorithms

Listing 1: Comparative Performance Analysis of Sorting Algorithms

```
1
2 /*
3  * Comparative Performance Analysis of Sorting Algorithms
4  *
5  * Author: Rafael Passos Domingues
6  *      RA: 2023.1.08.036
7  *
8  * The code at hand performs a performance analysis to compare non-recursive sorting methods
9  * and understand their differences. To do so, it builds vectors of varying dimensions, filled
10 * with non-repeated random numbers, and sorts them in three ways: in ascending, random and
11 * descending order.
12 *
13 * During sorting, the uses of each vector are counted, allowing a comparison between the
14 * methods.
15 * The comparison is performed between the three implemented methods, using vector sizes
16 * ranging
17 * from 100 to 10'000 units at regular intervals of 100 units.
18 *
19 * The code generates an output file called "sort_usages.csv", containing the data obtained.
20 * These values can be used to create graphs that make up the report "
21 * comparison_sort_algorithmis.pdf",
22 * which is part of this repository.
23 */
24
25 #include <iostream>
26 #include <fstream>
27 #include <ctime>
28
29 using namespace std;
30
31 // Function to perform Bubble Sort
32 int bubbleSort(int array[], int length) {
33     int bubbleUsage = 0; // Counter for Bubble Sort usage
34     int i, j;
35     for (i = 0; i < length - 1; i++) {
36         bubbleUsage += 2; // Increment counter for Bubble Sort usage
37         for (j = 0; j < length - i - 1; j++) {
38             if (array[j] > array[j + 1]) {
39                 int tmp = array[j];
40                 array[j] = array[j + 1];
41                 array[j + 1] = tmp;
42             }
43             bubbleUsage += 4;
44         }
45     }
46     return bubbleUsage;
47 }
48
49 // Function to perform Insertion Sort
50 int insertionSort(int array[], int length) {
51     int insertionUsage = 0; // Counter for Insertion Sort usage
52     int i, j;
53     for (i = 1; i < length; i++) {
54         int handle = array[i];
55         insertionUsage += 2; // Increment counter for Insertion Sort usage
56         for (j = i - 1; j >= 0 && array[j] > handle; j--) {
57             array[j + 1] = array[j];
58             insertionUsage += 3;
59         }
60         array[j + 1] = handle;
61     }
62     return insertionUsage;
63 }
64
65 // Function to perform Selection Sort
66 int selectionSort(int array[], int length) {
```

```

64     int selectionUsage = 0; // Counter for Selection Sort usage
65     int i, j;
66     for (i = 0; i < length - 1; i++) {
67         int minIndex = i;
68         selectionUsage += 2; // Increment counter for Selection Sort usage
69         for (j = i + 1; j < length; j++) {
70             if (array[j] < array[minIndex]) {
71                 minIndex = j;
72             }
73         }
74         int swap = array[i];
75         array[i] = array[minIndex];
76         array[minIndex] = swap;
77         selectionUsage += 4;
78     }
79     return selectionUsage;
80 }
81
82 // Function to generate an ordered array and shuffle it
83 void randomArrayGenerator(int start, int length, int randomArray[]) {
84     for (int i = 0; i < length; i++) {
85         randomArray[i] = i;
86     }
87
88     // shuffle array
89     for (int i = length - 1; i > 0; i--) {
90         int j = rand() % (i + 1);
91         int temp = randomArray[i];
92         randomArray[i] = randomArray[j];
93         randomArray[j] = temp;
94     }
95
96     for (int i = 0; i < length; i++) {
97         randomArray[i] += start;
98     }
99 }
100
101 // Function to generate increasing array
102 void increasingArrayGenerator(int array[], int start, int length) {
103     for (int i = 0; i < length; i++) {
104         array[i] = start + i;
105     }
106 }
107
108 // Function to generate decreasing array
109 void decreasingArrayGenerator(int array[], int start, int length) {
110     for (int i = 0; i < length; i++) {
111         array[i] = start + length - i - 1;
112     }
113 }
114
115 /*
116  * This function copy the elements of one array to another ensures that each sorting
117  * algorithm operates on a separate copy of the original matrix, avoiding interference
118  * between algorithm runs.
119  */
120 void copyArray(int source[], int destination[], int length) {
121     for (int i = 0; i < length; i++) {
122         destination[i] = source[i];
123     }
124 }
125
126 // Function to save results to a CSV file
127 void saveResultsToFile(const string& filename, const int sizes[], const int randomBubbleUsages
    [],
128                        const int increasingBubbleUsages[], const int decreasingBubbleUsages[],
129                        const int randomInsertionUsages[], const int increasingInsertionUsages
    [],
130                        const int decreasingInsertionUsages[], const int randomSelectionUsages
    [],

```

```

131         const int increasingSelectionUsages[], const int
              decreasingSelectionUsages[],
132         int size) {
133     ofstream file(filename);
134     if (!file.is_open()) {
135         cout << "Failed to create the file." << endl;
136         return;
137     }
138
139     // Write header to the file
140     file << "Array Size" << "," <<
141         "Random Bubble Sort Usage" << "," << "Increasing Bubble Sort Usage" << "," << "Decreasing
              Bubble Sort Usage" << "," <<
142         "Random Insertion Sort Usage" << "," << "Increasing Insertion Sort Usage" << "," << "
              Decreasing Insertion Sort Usage" << "," <<
143         "Random Selection Sort Usage" << "," << "Increasing Selection Sort Usage" << "," << "
              Decreasing Selection Sort Usage" << endl;
144
145     // Write data to the file
146     for (int i = 0; i < size; i++) {
147         file << sizes[i] << "," <<
148             randomBubbleUsages[i] << "," << increasingBubbleUsages[i] << "," <<
              decreasingBubbleUsages[i] << "," <<
149             randomInsertionUsages[i] << "," << increasingInsertionUsages[i] << "," <<
              decreasingInsertionUsages[i] << "," <<
150             randomSelectionUsages[i] << "," << increasingSelectionUsages[i] << "," <<
              decreasingSelectionUsages[i] << endl;
151     }
152
153     file.close();
154 }
155
156 int main(void) {
157     const int start = 100;
158     const int end = 10000;
159     const int step = 100;
160     const int numSizes = (end - start) / step + 1;
161
162     int sizes[numSizes];
163
164     int randomBubbleUsages[numSizes];
165     int increasingBubbleUsages[numSizes];
166     int decreasingBubbleUsages[numSizes];
167
168     int randomInsertionUsages[numSizes];
169     int increasingInsertionUsages[numSizes];
170     int decreasingInsertionUsages[numSizes];
171
172     int randomSelectionUsages[numSizes];
173     int increasingSelectionUsages[numSizes];
174     int decreasingSelectionUsages[numSizes];
175
176     int currentSize = start;
177     for (int i = 0; i < numSizes; i++) {
178         sizes[i] = currentSize;
179         currentSize += step;
180     }
181
182     for (int i = 0; i < numSizes; i++) {
183         int length = sizes[i];
184
185         // Random Array
186         int randomArray[length];
187         randomArrayGenerator(start, length, randomArray);
188
189         int sortArray[length];
190
191         // Bubble Sort
192         copyArray(randomArray, sortArray, length);
193         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

```
1
2 import csv
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 import numpy as np
7
8 def normalize_data(data):
9     max_value = max(data)
10    min_value = min(data)
11    return [(value - min_value) / (max_value - min_value) for value in data]
12
13 def main():
14     sizes = []
15     randomBubbleUsages = []
16     increasingBubbleUsages = []
17     decreasingBubbleUsages = []
18     randomInsertionUsages = []
19     increasingInsertionUsages = []
20     decreasingInsertionUsages = []
21     randomSelectionUsages = []
22     increasingSelectionUsages = []
23     decreasingSelectionUsages = []
24
25     # Read the CSV File
26     data = pd.read_csv("sort_usages.csv")
27
28     # Extract the data of columns
29     sizes = data["Array_Size"]
30     randomBubbleUsages = data["Random_Bubble_Sort_Usage"]
31     increasingBubbleUsages = data["Increasing_Bubble_Sort_Usage"]
32     decreasingBubbleUsages = data["Decreasing_Bubble_Sort_Usage"]
33     randomInsertionUsages = data["Random_Insertion_Sort_Usage"]
34     increasingInsertionUsages = data["Increasing_Insertion_Sort_Usage"]
35     decreasingInsertionUsages = data["Decreasing_Insertion_Sort_Usage"]
36     randomSelectionUsages = data["Random_Selection_Sort_Usage"]
37     increasingSelectionUsages = data["Increasing_Selection_Sort_Usage"]
38     decreasingSelectionUsages = data["Decreasing_Selection_Sort_Usage"]
39
40     # Normalize the data
41     sizes = normalize_data(sizes)
42     randomBubbleUsages = normalize_data(randomBubbleUsages)
43     increasingBubbleUsages = normalize_data(increasingBubbleUsages)
44     decreasingBubbleUsages = normalize_data(decreasingBubbleUsages)
45     randomInsertionUsages = normalize_data(randomInsertionUsages)
46     increasingInsertionUsages = normalize_data(increasingInsertionUsages)
47     decreasingInsertionUsages = normalize_data(decreasingInsertionUsages)
48     randomSelectionUsages = normalize_data(randomSelectionUsages)
49     increasingSelectionUsages = normalize_data(increasingSelectionUsages)
50     decreasingSelectionUsages = normalize_data(decreasingSelectionUsages)
51
52     # Calculate the mean for each method and create a dictionary
53     method_means = {
54         "Random_Bubble": sum(randomBubbleUsages) / len(randomBubbleUsages),
55         "Increasing_Bubble": sum(increasingBubbleUsages) / len(increasingBubbleUsages),
56         "Decreasing_Bubble": sum(decreasingBubbleUsages) / len(decreasingBubbleUsages),
57         "Random_Insertion": sum(randomInsertionUsages) / len(randomInsertionUsages),
58         "Increasing_Insertion": sum(increasingInsertionUsages) / len(increasingInsertionUsages)
59         ,
60         "Decreasing_Insertion": sum(decreasingInsertionUsages) / len(decreasingInsertionUsages)
61         ,
62         "Random_Selection": sum(randomSelectionUsages) / len(randomSelectionUsages),
63         "Increasing_Selection": sum(increasingSelectionUsages) / len(increasingSelectionUsages)
64         ,
65         "Decreasing_Selection": sum(decreasingSelectionUsages) / len(decreasingSelectionUsages)
66         ,
67     }
```

```

63 }
64
65 # Sort the method_means dictionary by mean values (ascending order)
66 sorted_means = sorted(method_means.items(), key=lambda x: x[1])
67
68 # Unpack the sorted_means into two separate lists
69 sorted_methods, sorted_means = zip(*sorted_means)
70
71 # Define colors for each method
72 colors = {"Random_Bubble": "blue", "Increasing_Bubble": "blue", "Decreasing_Bubble": "blue"
73           ,
74           "Random_Insertion": "green", "Increasing_Insertion": "green", "Decreasing_
75           Insertion": "green",
76           "Random_Selection": "red", "Increasing_Selection": "red", "Decreasing_Selection":
77           "red"}
78
79 # Create a pie chart showing the percentage of usage for each method
80 plt.figure(figsize=(8, 8))
81
82 # Define the explode values to highlight the method of lowest usage
83 explode = [0.1 if method == sorted_methods[0] else 0 for method in sorted_methods]
84
85 plt.pie(sorted_means, labels=sorted_methods, colors=[colors[method] for method in
86 sorted_methods],
87         autopct="%1.4f%%", startangle=140, explode=explode)
88
89 plt.title("Percentage_of_Usage_for_Each_Sorting_Algorithm")
90 plt.axis("equal")
91
92 plt.show()
93
94 # Calculate the mean normalized usage for each size
95 mean_usage_insertion = (np.array(randomInsertionUsages) + np.array(
96     increasingInsertionUsages) + np.array(decreasingInsertionUsages)) / 3
97
98 # Create a scatter plot with regression line for the mean normalized usage (insertion
99     method)
100 sns.set_theme(style="whitegrid")
101 plt.figure(figsize=(10, 6))
102 plt.scatter(sizes, randomBubbleUsages, label="Random_Bubble", s=30, color=colors["Random_
103     Bubble"], alpha=0.1)
104 plt.scatter(sizes, increasingBubbleUsages, label="Increasing_Bubble", s=30, color=colors["
105     Increasing_Bubble"], alpha=0.1)
106 plt.scatter(sizes, decreasingBubbleUsages, label="Decreasing_Bubble", s=30, color=colors["
107     Decreasing_Bubble"], alpha=0.1)
108 plt.scatter(sizes, randomSelectionUsages, label="Random_Selection", s=30, color=colors["
109     Random_Selection"], alpha=0.1)
110 plt.scatter(sizes, increasingSelectionUsages, label="Increasing_Selection", s=30, color=
111     colors["Increasing_Selection"], alpha=0.1)
112 plt.scatter(sizes, decreasingSelectionUsages, label="Decreasing_Selection", s=30, color=
113     colors["Decreasing_Selection"], alpha=0.1)
114
115 plt.plot(sizes, mean_usage_insertion, label="Mean_Normalized_Usage_(Insertion)", color="
116     green", linewidth=2)
117
118 plt.xlabel("Normalized_Array_Size")
119 plt.ylabel("Normalized_Usage")
120 plt.title("Normalized_Usage_for_Each_Sorting_Algorithm")
121 plt.legend(loc="upper_left")
122
123 # Adjust y-axis limits for better visualization
124 plt.ylim(-0.1, 1.1)
125
126 plt.show()
127
128 # Run the main function
129 if __name__ == "__main__":
130     main()

```