ISSN: 2230-9926

Available online at http://www.journalijdr.com

IJDR

International Journal of Development Research
Vol. 09, Issue, 06, pp. 28405-28415, June, 2019

RESEARCH ARTICLE

OPEN ACCESS

DEVELOPMENT OF SOFTWARE FOR QUALITATIVE AND COMPARATIVE PERFORMANCE ANALYSIS BETWEEN SOME DATA ORDINATION ALGORITHMS

[1,2]Ricardo Silva Parente, [1,2]Ítalo Rodrigo Soares Silva, [2]Davi Samuel Dias Maia, [1,2]Paulo Oliveira Siqueira, [1,*]Jorge de Almeida Brito Júnior, [1]Manoel Henrique Reis Nascimento, [1]Alyson de Jesus dos Santos and Héber Pinheiro Martins[2]

[1]Research Department, Institute of Technology and Education Galileo of the Amazon (ITEGAM), Manaus, Amazonas, Brazil

[2]Student, Paulista University (UNIP), Manaus, Amazonas, Brazil.

ARTICLE INFO

Article History:

Received 28th March, 2019
Received in revised form
09th April, 2019
Accepted 14th May, 2019
Published online 30th June, 2019

Key Words:

Data ordering,
Computational complexity,
Algorithms, Qualitative analysis.

ABSTRACT

Ordination can be obtained by common means such as the logical reasoning of a human being, and also by computational means using algorithms to sort the desired data in a predefined order, the present research aims to show the results of the ordering of integers through algorithms, such as Insertionsort, Combsort, Quicksort and Mergesort, through an application developed in the Java programming language, where a performance comparison of each algorithm will be performed, showing not only its performance measured in milliseconds but also aspects fundamental as their operation and the ordering logic employed by them, the results show comparisons of different views according to the complexity worked in each scenario.

Citation: Ricardo Silva Parente, Ítalo Rodrigo Soares Silva, Davi Samuel Dias Maia, Paulo Oliveira Siqueira, Jorge de Almeida Brito Júnior, Manoel Henrique Reis Nascimento, Alyson de Jesus dos Santos and Héber Pinheiro Martins. 2019. "Development of software for qualitative and comparative performance analysis between some data ordination algorithms", *International Journal of Development Research*, 09, (05), 28405-28415.

INTRODUCTION

Data ordering consists of a set of methods and techniques for ordering a complete or partial sequence of data or information. Ordination can be obtained by common means such as the logical reasoning of a human being, and also by computational means using algorithms to sort the desired data in a predefined order. There is a great diversity of sorting algorithms, each using a specific logic, and in some cases sorting algorithms have similar sort ordering techniques, but their performance often varies from one technique to another. This variation of performance measured in time, can impact in obtaining a specific result, because in many cases an algorithm that takes more time to make the ordering, has a high probability of being inappropriate for a certain task. Therefore, one must choose a method of ordering appropriate to the type of work, and order it in the expected time. Measuring the time that each algorithm takes to order a same set of elements, it is a task of paramount importance in the computer industry, because depending on the system in which it is applied, its performance will make a difference. In this research we will present the ordering algorithms: Insertionsort, Combsort, Quicksort and Mergesort, showing not only their performance measured in milliseconds, but also fundamental aspects such as their operation and the ordering logic employed by them, capturing the times through a application developed in the Java programming language, where integer type vectors will be ordered. In addition, the algorithms Insertion sort and Combsort are considered simple ordering algorithms, usually used in the ordering of small-sized vectors, are algorithms of easy understanding and application. Their complexity on average is Θ (n²), and in some situations they reach Θ (n) in the best cases.

***Corresponding author: Jorge de Almeida Brito Júnior,**
Research Department, Institute of Technology and Education Galileo of the Amazon (ITEGAM), Manaus, Amazonas, Brazil.

Quicksort and Mergesort are considered sophisticated or efficient sorting methods, because they have a short ordering time. They have a somewhat greater complexity in detail, but their greatest asset is due to a much smaller number of comparisons. They are usually used to order a higher amount of data, with their complexities on average Θ (n log n). Analyzing and comparing sorting algorithms: Insertionsort, Combsort, Quicksort and Mergesort in a Java programming language in order to report performance results using mass data masses is the objective of this article, and it is necessary to analyze the ordering methods as well as performance in the application developed for this purpose.

MATERIALS AND METHODS

Ordination of data: The ordering of data consists of putting elements be they information, data and others in a predefined order, complementing Laureano says that "ordering is the process of arranging a set of similar information in a growing or decreasing order. Specifically, given an ordered list i of n elements, then: i1 <= i <= ... <= In [Laureano, 2012]". According to Da Silva Nascimento and Mozzaquatro: "There are several ways to implement a sorting algorithm, but there is one, but how much need each use. Each algorithm solves a common problem that is the ordering, but as each one behaves in a different way we must understand how each one works so that we know which one to use to solve a certain problem. Their use allows us to solve a problem dynamically, that is, after implementing an algorithm to order a vector in ascending order, for example, it must be functional to any vector regardless of the quantity of values or the way in which these values are arranged in the initial situation in which they are in the vector [Da Silva Nascimento, 2016] ".

When ordering something we have input data (which are still out of order), and output data (data already ordered), in the middle of this process methods and procedures are used in order to obtain an ordered sequence. As in the example quoted below from Cormem et al. [2002]:

- Input: A sequence of n numbers (a1, a2, ..., an).
- Output: A permutation (reordering) (a'1, a'2, ..., a'n) of the input sequence, such that a'1 ≤ a'2 ≤ ... ≤ a'n
- According to Da Silva Nascimento and Mozza Quatro, sorting methods are classified into two main groups: internal and external sorting [Da Silva Nascimento, 2016].
- Internal Ordering: These are methods that do not require a secondary memory for the process; ordering is done in the main memory of the computer.
- External Sort: When the file to be sorted does not fit into main memory and therefore has to be stored on tape or disk.
- The main difference between the two groups, according to Oliveira, is that in the internal ordering method any record can be accessed directly, whereas in the external method it is necessary to do access in blocks [Oliveira, 2002].

Ordination Algorithms: In Laureano's line of thinking the ordering algorithms follow a programming logic, they are able to order a set of elements that appear outside a specific order type - in other words, the elements of the set when they go through the method of ordering the algorithm are placed in a complete or partial order, the numerical and lexicographic orders being the most used [Laureano, 2012]. According to Viana in data ordering algorithms, there are simple sorting methods and efficient sorting methods. The simple methods are easy to apply and understand, since efficient methods are more sophisticated with a smarter logic, and it aims above all performance [Viana, 2016]. There is a great variety of sorting algorithms, used for several sorts of sorts, a crucial factor to qualify is the time spent for sorting already said by Celes and Rangel [Celes, 2002]. In this academic work four sorting algorithms will be approached, for ordering integer vectors, among the chosen algorithms are: InsertionSort, CombSort, QuickSort and MergeSort.

Insertion Sort: Following the Laureano's thinking, the ordering algorithm InsertionSort is considered as a very simple algorithm, its implementation as well as its understanding are of easy applicability and assimilation [Laureano, 2012]. This type of sorting works very well for small-order sorting solutions, one of the most efficient of its kind, which is Simple Sorting Methods, it uses a technique similar to human reasoning, so that,"The insertion algorithm works the same way many people order cards in a card game such as poker. One of the characteristics of this algorithm is the smallest number of exchanges and comparisons if the list is ordered (partially) [Laureano, 2012]". Where the player with a set of cards in his hand receives a new card, it is up to him to compare this new card to insert the card next to the others in the correct position, as he receives new cards, the player must make new comparisons to insert the cards. new cards in the hand of cards that are already sorted, until no more new cards are added for inclusion. The performance of InsertionSort is worthy of a simple sorting method, but it can be more effective than BubbleSort and SelectionSort, which are in the same category according to tests performed by the authors themselves.

Comb Sort

According to Burke CombSort is a simple ordering algorithm, based on the principle of exchanges, it was initially developed by Wlodzimierz Dobosiewicz in 1980. However this sort of ordering was forgotten for a long time, until in April 1991 it was remembered again by Stephen Laccy and Richard Box in an article published in Byte magazine [Burke, 2014]. Following the Burke's writing, the CombSort algorithm is proposed to be an improvement to the BubbleSort sorting algorithm, since it helps to eliminate the slowness at the end of arrays that are very common [Burke, 2014]. In many vectors there is often a low-order disordered element at the end of the vector, this brings a large time bottleneck in ordering using simpler ordering methods. The same author continues to point out that using this sort technique the vector is scanned in an increasing manner, choosing two elements for comparison, which are separated by a space (jump), this space is called GAP, if the value on the right is less than the value on the left, these elements change their positions, this is done repeatedly until the sequence is ordered [Burke, 2014]. BubbleSort also presents this distance between the elements chosen for comparison, however it only equals 1, already in Combsort this distance can be much greater, thus improving the time to order the vector, it also uses a shrink factor that is a constant equivalent to 1.24 used to make the ordering computations of this algorithm, Burke concludes in his paper [Burke, 2014].

Quick Sort: The Quick Sort method makes use of the division and conquest strategy, as its own name says it is a fast and very efficient sort algorithm says Laureano [Laureano, 2012]. This algorithm consumes a time proportional to Θ (n log n) in average and proportional to Θ (n^2) in the worst case, says Cormem [2002]. According to Laureano:

"The algorithm, published by Professor C.A.R. Hoare in 1962, is based on the simple idea of dividing a vector (o-its list to be ordered) into two sub-vectors, so that all elements of the first vector are smaller or equal to all elements of the second vector. Once the division is established, the problem will be solved, since by recursively applying the same technique to each of the sub-vectors, the vector will be ordered by obtaining a sub-vector of only 1 element" [Laureano, 2012]. The process of dividing, conquering and combining QuickSort can be extremely efficient if the central element (pivot) to be chosen represents a median value of the set of elements, if this happens, just after positioning the pivot, there will remain only two sub-vectors to be ordered, both with the number of elements reduced by half, in relation to the original vector, explains Celes and Rangel on the operation of the algorithm QuickSort [Celes, 2002].

Narrative description of the QuickSort Algorithm according to Laureano [Laureano, 2012]:

- Choose any element (a pivot) of the set to be ordered.
- Remove the pivot from the set of elements, and partition the remaining elements of the set into 2 distinct sequences, so that one has a subset directly to the pivot and one to the left of the pivot.
- The left subset should have elements smaller or equal to the pivot, while the rightmost subset will contain elements greater than or equal to the pivot.

Finally, the algorithm is applied again in the subsets formed.

Merge Sort

Continuing with Laureano, he notes that just like QuickSort Mergesort is a sorting algorithm that makes use of division and conquest, this method divides the input vector into two halves, then the divisions occur until all the elements to be ordered are separated from each other, in the sequence the elements (subdivisions) are ordered by recursion, and are gradually joined until the vector is completely ordered, that is the conquest, soon after we have the junction of all sub problems solved to form the vector ordered [Laureano, 2012]. The authors Vargas and Garcia explain that because of the constant divisions of the vector for the ordering by recursion, the luck Merge has a use of memory considerably high, being classified as little efficient in some circumstances. On average, its ordering time is Θ (n log n) and in the worst case we also have Θ (n log n), each part of the MergeSort order has a specific time [Vargas, 2004]. According to Laureano, the Narrative description of the Mergesort Algorithm is [Laureano, 2012]:

- Divide the vector into small subsequences, where first the vector of size n will be divided into two parts, these parts will be divided again into two other parts, this is done until the elements of the vector are all separated;
- 2. In this step conquest happens, where recursively there is the classification of the parts previously divided, so that they are ordered;

In the last step, the union of the ordered sub-vectors occurs, which is again ordered for this junction, until it forms the ordered final vector.

Complexity algorithms: According to Alves et al., The interval arithmetic developed by Moore aims to control errors of the results of the numerical comparison through the manipulation and operations with intervals, in a way analogous to the work done by algorithms that model iterative techniques of loops where the level of processing defines the context of complexity [Alves, 2018]. According to Piqueira this concept of computational complexity comes from the time of the Turing Machine where the operations of the head and the tape are defined by a table of instructions {I1, I2, ..., In} called the action table [Piqueira, 2016]. The forms of interactions are present in several computational systems and their study is necessary since the performance in information response depends on a good planning of the construction of a software or product. The concept of computational complexity, understood as the number of operations required for the execution of a program, that is, for the execution of a set of algorithms according to Desurvire [2009]. The model shown in the figure above allows to understand how the turing machine works, it is observed that there is a sequential iteration that depending on the size of the tape there will be a great loss in performance in reading and writing since at that time was the ideal model , computational architectures are now based on the Von Newman model, yet the complexity becomes persistent, programming techniques become unfeasible when alarming results are obtained mathematically.

For White and Fuchigami: "The traditional programming problem in flowshop production system occurs in a set of n tasks that must be processed, in the same sequence, in m machines. When the order of processing on all machines is the same, we have the flowshop production environment permutacional, in which the number of possible schedules for n tasks is n! [Branco, 2017] ". In the model elaborated by Branco and Fuchigami is exemplified a Flowshop with four machines and four tasks. The problem is to get a task sequence that optimizes a given measure of performance. With this, the concept of measures of computational complexity arises, being a set of related complexity problems taking as factors the type of problem, the computation model and the resources.

According to Goldreich: Limiting the computation time above by some concrete function f (n) often produces complexity classes that depend on the model of the chosen machine. For example, the language {xx | x is any binary sequence} can be solved in linear time on a multi-tap Turing machine, but necessarily requires quadratic time in the single-tape Turing machine model. If we allow variations in polynomial time running, the Cobham-Edmonds thesis states that "the complexities of time in any two reasonable and general computational models are polynomially related."

According to Pinheiro et al. "To think under the lens of Complexity Theory is to respect the various dimensions of the phenomenon studied, it is to oppose competing and antagonistic conceptions aiming at complementarity through a movement that associates them." Vilela comments on the concept of complexity classes so that A class P plays an important role in complexity theory because it is invariant in all computing models that are polynomially equivalent to the deterministic Turing machine of a single tape and corresponds approximately to the class of problems that are solved realistically in a computer [Vilela, 2016]. Thus complexity theory deals with the adversities of three-level computing commonly known as measures of complexity: best case, worst case, and average case. These concepts will be approached in the course of the methodology according to the use of the ordering algorithms.

Classes of ordination algorithms: The software was developed in Java language using Netbeans IDE 8.2. 5 classes were used, the first called Insertion Sort, contains methods such as ordering the vector, and that returns the vector already ordered in a String, while one method orders, the other just runs through the vector in order to get its values and adds it in a String. The second class called Comb Sort also has two methods that do the same as the first class mentioned above, the difference being that while the first uses the sorting algorithm known as Insertion Sort the second uses the CombSort sorting technique to sort the vectors that will be passed as an argument. The third class named MergeSort, which also gets the name of its respective sorting algorithm, sorts the numbers that are in the vectors initially disordered, has three methods, due to its greater complexity to sort the data. The fourth class is Quick Sort, has five methods because it is the fastest technique used in this work and also the most complex compared to others, the methods that this class has the most are only to order the vectors, four methods are used that combined serve to sort the data that will be passed through a vector as an argument. The fifth class is the class APS that extends the class JFrame, is the class that will shape the software, ie the final view of the application where the results are shown, with the inclusion of buttons, tables and text areas where the vectors are shown.

Generation of data for ordination: Initially, four vectors of different sizes are created in the constructor of the class that inherits JFrame, where they are randomly filled within a loop of repetition, numbers are drawn in a range of 0 to 100,000 thousand using an already existing method in the Java language of the Randon Class, identified by nextInt (), this process is repeated for the four vectors. The process of populating the vectors is triggered by a click on the "show" button, which is disabled after the click, this button basically serves to add to the vectors the random values, the unordered vectors will appear just above the button, since the method that returns the unordered vector is called and played inside a component called JTextPane, this is done for the four vectors, the "sort" buttons start disabled and are enabled after the button that shows the unordered vectors is deactivated. Four replicate loops known as for were used to fill in the vectors, one for each vector.

Data ordination process: For each ordering algorithm, a class of its own was created, in order to leave the code following the patterns of the orientation to objects, all these classes have methods to order the vectors, knowing that these own methods to order the vectors return a set of characters (String), whose value is the time spent for ordination. So the sort method is called when the "sort" button is clicked (the button is disabled after the click), the time spent for sorting is saved in a variable to be used later, then another method is called this takes the vector already ordered and returns a String to be displayed next to the "sort" button, this process is done for each vector and all the algorithms of sorting of the software.

Comparative Performance of the Ordination Algorithms

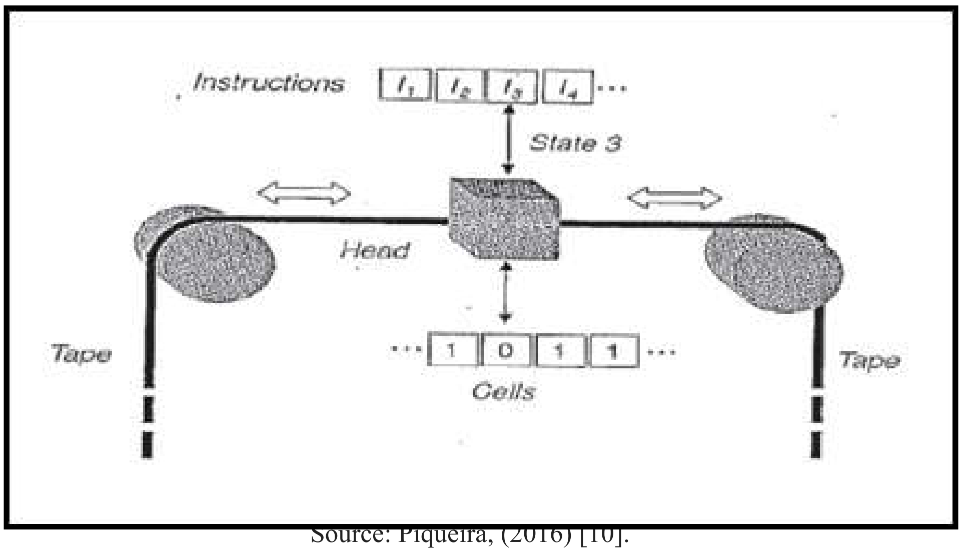
A table shows the performance results of each algorithm and its respective vectors. In the first column is placed the size of the vectors, in a way to identify the vectors that are being related to each time, in the other columns is placed the variables that will receive the ordering time of each vector and algorithm. The whole process is triggered when the "Show Results" button is clicked, a detail that can not be left out is that the button to show algorithm performance results is enabled after all sort buttons are clicked. Once the button is clicked the method to display the table is called and executed.

RESULTS AND DISCUSSION

Based on the results obtained in the software developed by the group, it was possible to perform an analysis of the performance of each algorithm, and all the vectors were filled in a totally random way, where such results were generated using a computer with the following specifications:

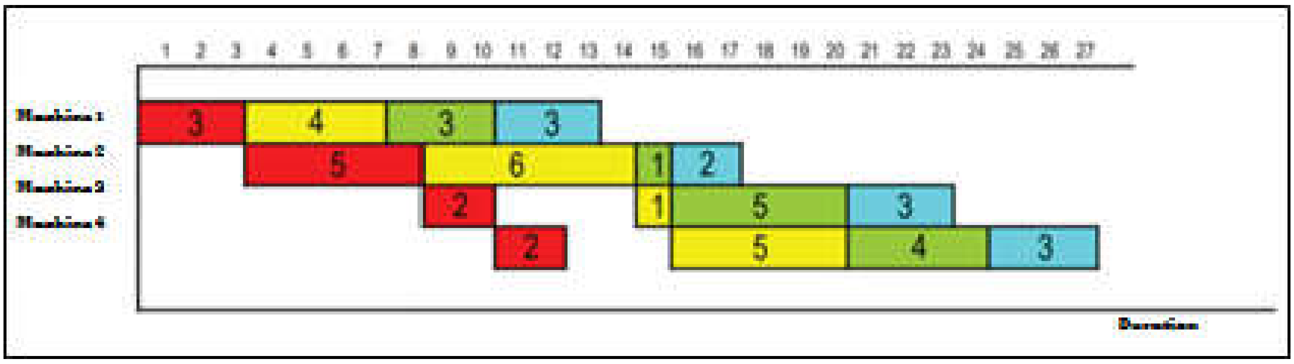
- Processor - Intel (R) Core (TM) i3-5005U CPU 2.00 GHz.
- Memory RAM - 4.00 GB.
- Operating System - Windows 10 64-bit.

Insertion Sort had the worst performance of the four algorithms used for all vector sizes, the least effective being performance, because it is a simple ordering algorithm, the result found was already expected by the group. Despite the slowness in data organization this method becomes useful when the few data to be ordered due to its easy implementation. Merge Sort did not perform well in the tests, due to the use of a division and recursion method, making the comparisons made a lot and consequently the performance of the algorithm was not good for vectors organized in a totally random way, being in third place in the ranking ranking in terms of performance of the algorithms used, earning only from insertion Sort. The Comb Sort, even though it was considered a simple algorithm, obtained a very encouraging result for the vectors used, being the second best in the analysis of the group. This had an approximate organization of 2.7 times slower than the fastest of the algorithms evaluated by the group, if considered the vector of size 40000 (forty thousand), comparing the time related to the vector of 10,000 (ten thousand) positions, the comb Sort further improves its performance to only about 1.49 times slower. And to the surprise of the authors of this work, the algorithm in ordering the vector of size equal to 20,000 (twenty thousand) indexes was faster than quick Sort. The one that had the best performance in all sizes of vectors tested was quick Sort, could not be another, as the name itself says it is very fast, achieved surprising results and was ranked first in the ranking formed, despite the result already be expected by the team, managed to surprise positively. To get a sense of the stark difference between the slower and faster algorithms tested by the group, the insertion Sort was almost 52 (fifty-two) times slower than the faster of the evaluated methods if considering the vector size 40,000 (forty thousand) and approximately 68 (sixty-eight) times slower compared to the vector of size 10,000 (ten thousand). The results were satisfactory and very consistent, a special highlight for comb Sort, which, even though it was a simple sort algorithm, managed to get close enough to the organization time of quick Sort.



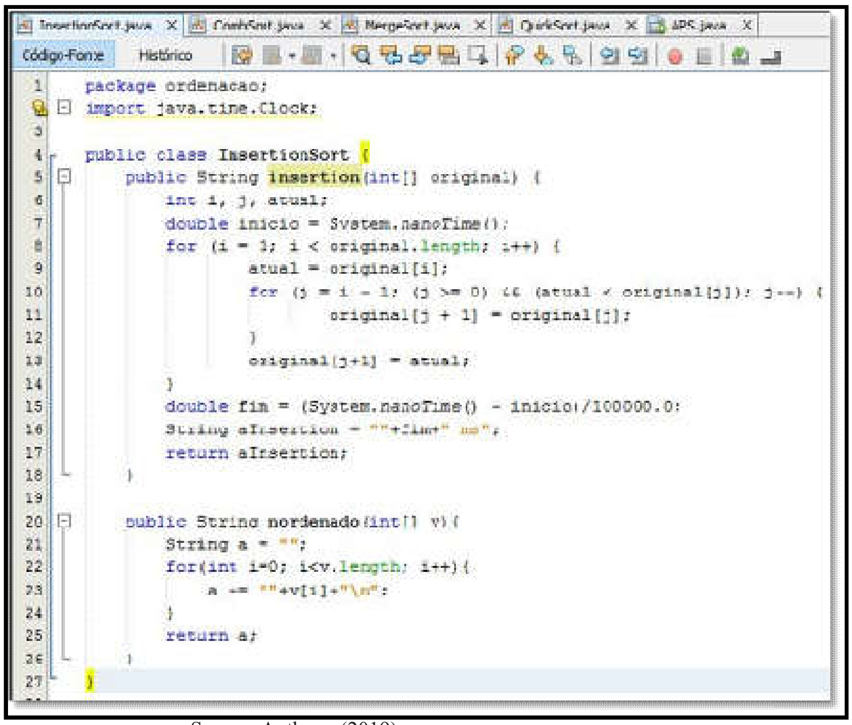
Source: Piqueira, (2016) [10].

Fig. 1. Turing Machine Model



Source: White and Fuchigami, (2017) [12].

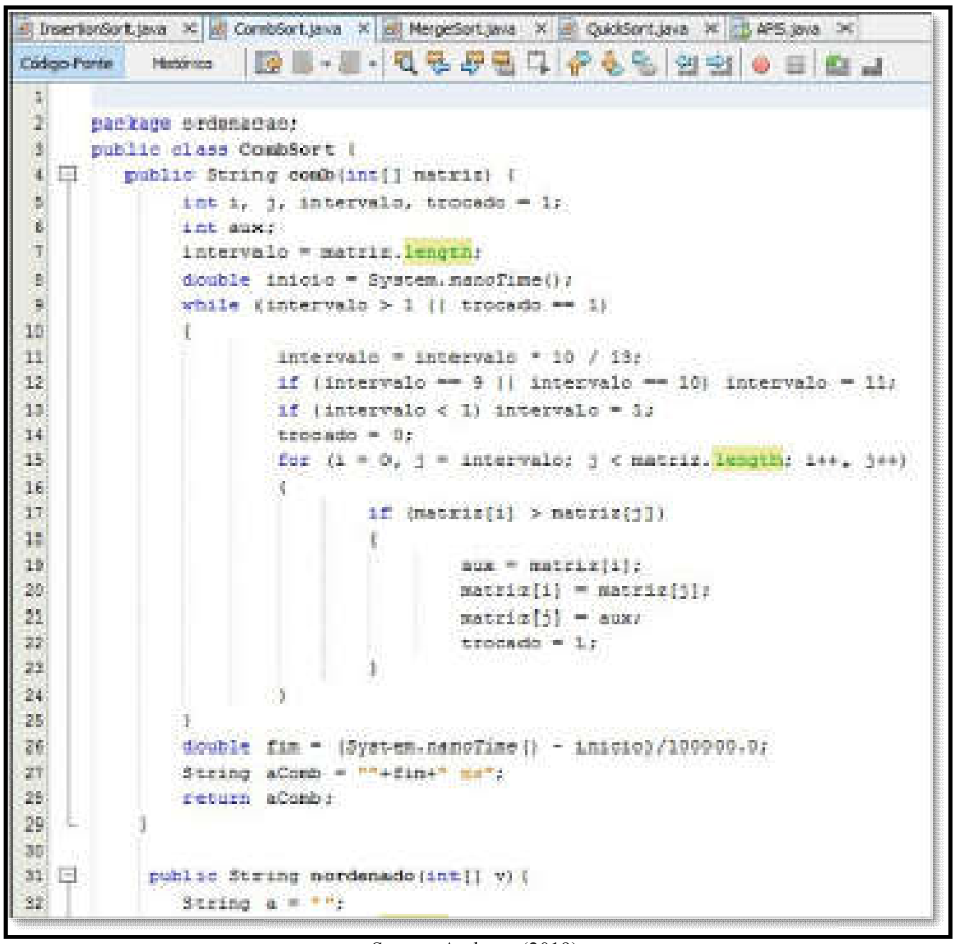
Fig. 2. FlowShop Model



```java
package ordenacao;
import java.time.Clock;

public class InsertionSort {
    public String insertion(int[] original) {
        int i, j, atual;
        double inicio = System.nanoTime();
        for (i = 1; i < original.length; i++) {
                atual = original[i];
                for (j = i - 1; (j >= 0) && (atual < original[j]); j--) {
                        original[j + 1] = original[j];
                }
                original[j+1] = atual;
        }
        double fim = (System.nanoTime() - inicio)/100000.0;
        String aInsertion = ""+fim+" ms";
        return aInsertion;
    }

    public String nordenado(int[] v){
        String a = "";
        for(int i=0; i<v.length; i++){
            a += ""+v[i]+"\n";
        }
        return a;
    }
}
```

Source: Authors, (2019).

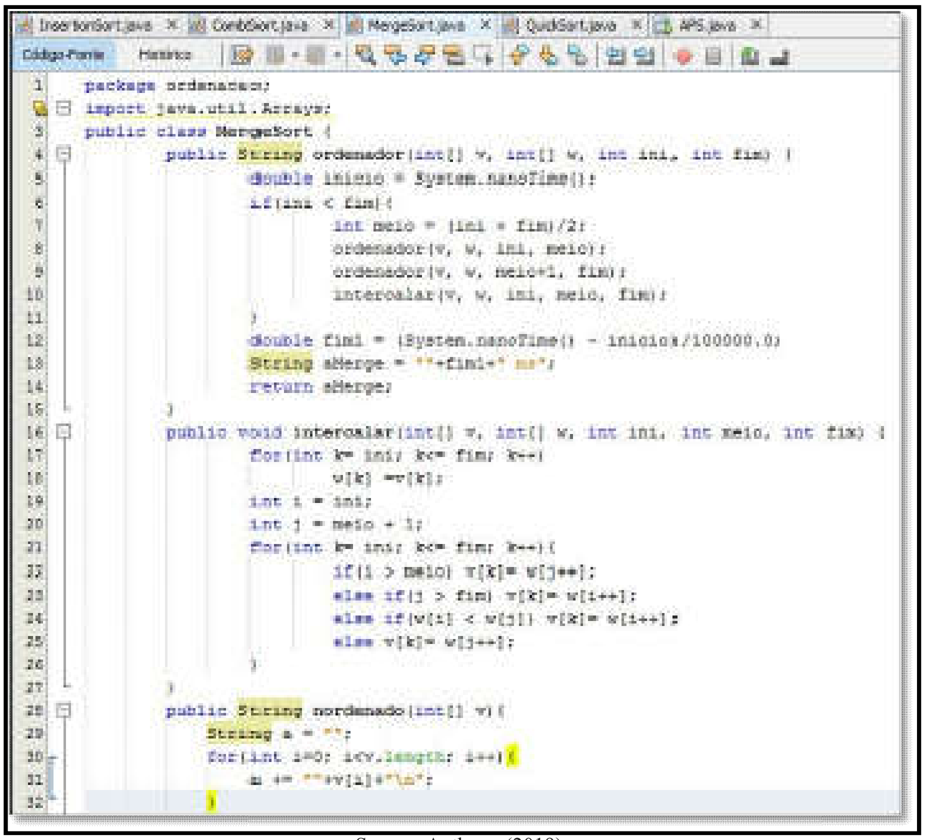
Fig. 3. InsertionSort class



```java

package ordenacao;
public class CombSort {
    public String comb(int[] matriz) {
        int i, j, intervalo, trocado = 1;
        int aux;
        intervalo = matriz.length;
        double inicio = System.nanoTime();
        while (intervalo > 1 || trocado == 1)
        {
                intervalo = intervalo * 10 / 13;
                if (intervalo == 9 || intervalo == 10) intervalo = 11;
                if (intervalo < 1) intervalo = 1;
                trocado = 0;
                for (i = 0, j = intervalo; j < matriz.length; i++, j++)
                {
                        if (matriz[i] > matriz[j])
                        {
                                aux = matriz[i];
                                matriz[i] = matriz[j];
                                matriz[j] = aux;
                                trocado = 1;
                        }
                }
        }
        double fim = (System.nanoTime() - inicio)/100000.0;
        String aComb = ""+fim+" ms";
        return aComb;
    }

    public String nordenado(int[] v) {
        String a = "";
```

Source: Authors, (2019).

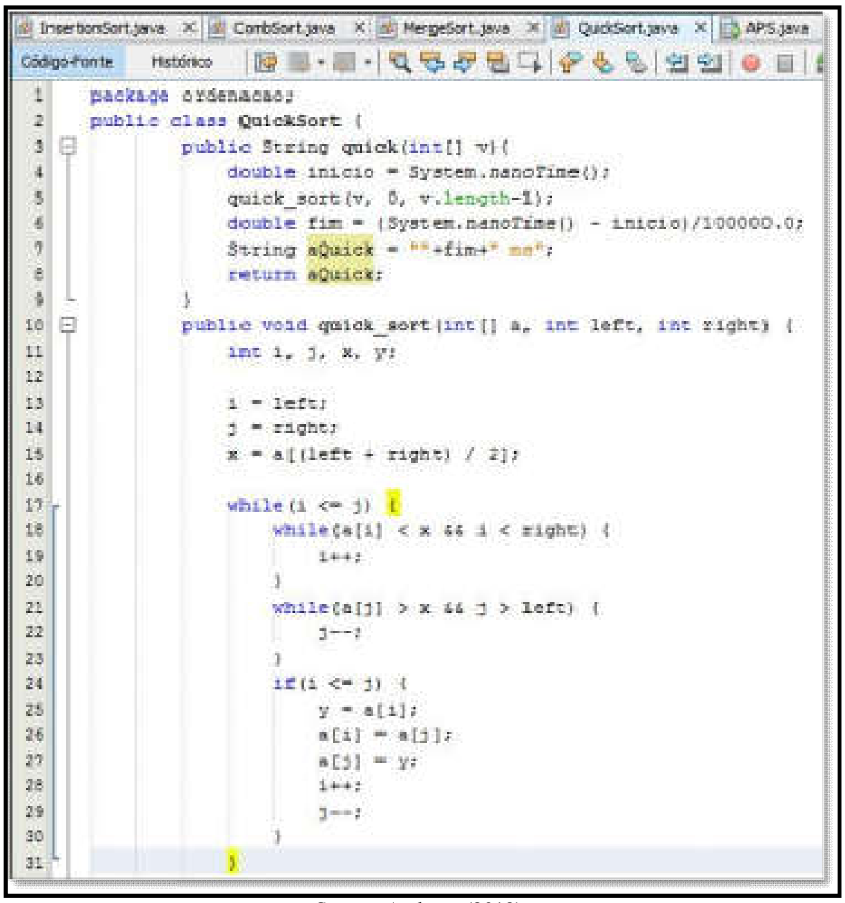
Fig. 4. CombSort Class



```java
package ordenacao;
import java.util.Arrays;
public class MergeSort {
        public String ordenador(int[] v, int[] w, int ini, int fim) {
                double inicio = System.nanoTime();
                if(ini < fim){
                        int meio = (ini + fim)/2;
                        ordenador(v, w, ini, meio);
                        ordenador(v, w, meio+1, fim);
                        intercalar(v, w, ini, meio, fim);
                }
                double fim1 = (System.nanoTime() - inicio)/100000.0);
                String aMerge = ""+fim1+" ms";
                return aMerge;
        }
        public void intercalar(int[] v, int[] w, int ini, int meio, int fim) {
                for(int k= ini; k<= fim; k++)
                        w[k] =v[k];
                int i = ini;
                int j = meio + 1;
                for(int k= ini; k<= fim; k++){
                        if(i > meio) v[k]= w[j++];
                        else if(j > fim) v[k]= w[i++];
                        else if(w[i] < w[j]) v[k]= w[i++];
                        else v[k]= w[j++];
                }
        }
        public String nordenado(int[] v){
            String a = "";
            for(int i=0; i<v.length; i++){
                a += ""+v[i]+"\n";
            }
```

Source: Authors, (2019).

Fig. 5. MergeSort Class

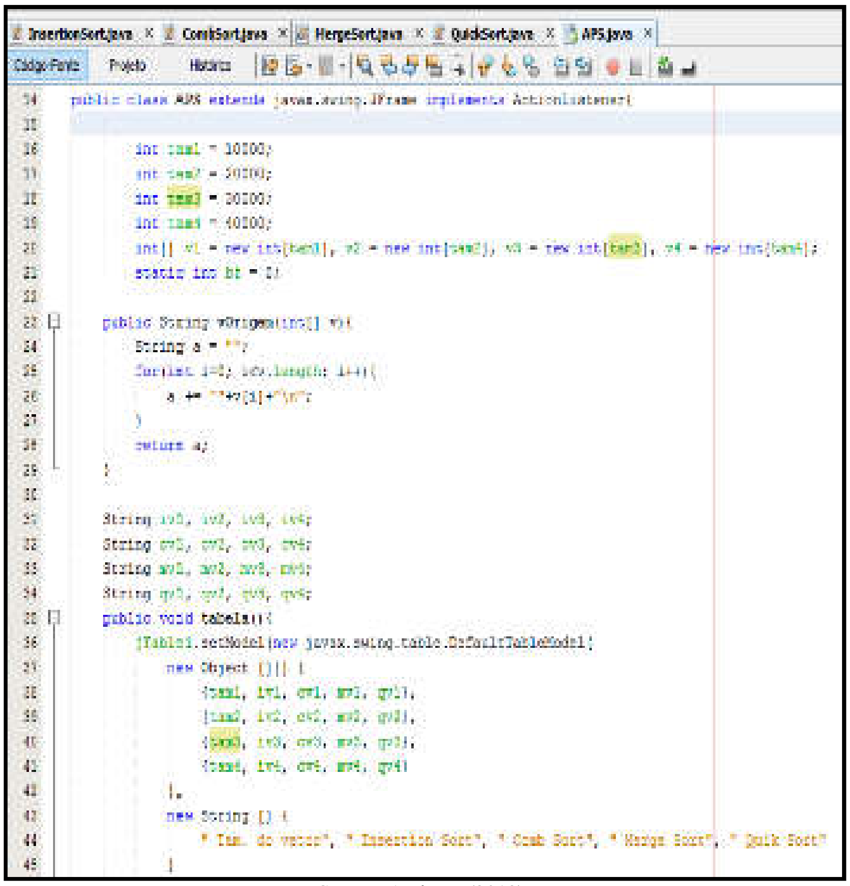

InsertionSort.java × CombSort.java × MergeSort.java × QuickSort.java × APS.java

Código-Fonte Histórico

```java
package ordenacao;
public class QuickSort {
        public String quick(int[] v){
            double inicio = System.nanoTime();
            quick_sort(v, 0, v.length-1);
            double fim = (System.nanoTime() - inicio)/100000.0;
            String aQuick = ""+fim+" ms";
            return aQuick;
        }
        public void quick_sort(int[] a, int left, int right) {
            int i, j, x, y;

            i = left;
            j = right;
            x = a[(left + right) / 2];

            while(i <= j) {
                while(a[i] < x && i < right) {
                    i++;
                }
                while(a[j] > x && j > left) {
                    j--;
                }
                if(i <= j) {
                    y = a[i];
                    a[i] = a[j];
                    a[j] = y;
                    i++;
                    j--;
                }
            }
```

Source: Authors, (2019).

Fig. 6. Quick Sort Class

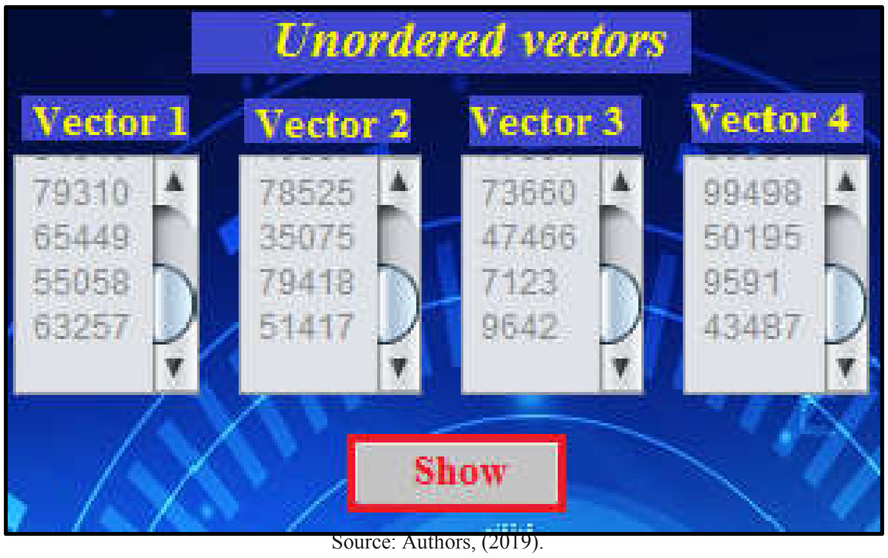

InsertionSort.java × CombSort.java × MergeSort.java × QuickSort.java × APS.java ×

Código-Fonte Projeto Histórico

```java
public class APS extends javax.swing.JFrame implements ActionListener{

        int tam1 = 10000;
        int tam2 = 20000;
        int tam3 = 30000;
        int tam4 = 40000;
        int[] v1 = new int[tam1], v2 = new int[tam2], v3 = new int[tam3], v4 = new int[tam4];
        static int bt = 0;

    public String vOrigem(int[] v){
        String a = "";
        for(int i=0; i<v.length; i++){
            a += ""+v[i]+"\n";
        }
        return a;
    }

    String iv1, iv2, iv3, iv4;
    String cv1, cv2, cv3, cv4;
    String mv1, mv2, mv3, mv4;
    String qv1, qv2, qv3, qv4;
    public void tabela(){
        jTable1.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {
                {tam1, iv1, cv1, mv1, qv1},
                {tam2, iv2, cv2, mv2, qv2},
                {tam3, iv3, cv3, mv3, qv3},
                {tam4, iv4, cv4, mv4, qv4}
            },
            new String [] {
                " Tam. do vetor", " Insertion Sort", " Comb Sort", " Merge Sort", " Quik Sort"
            }
```

Source: Authors, (2019).

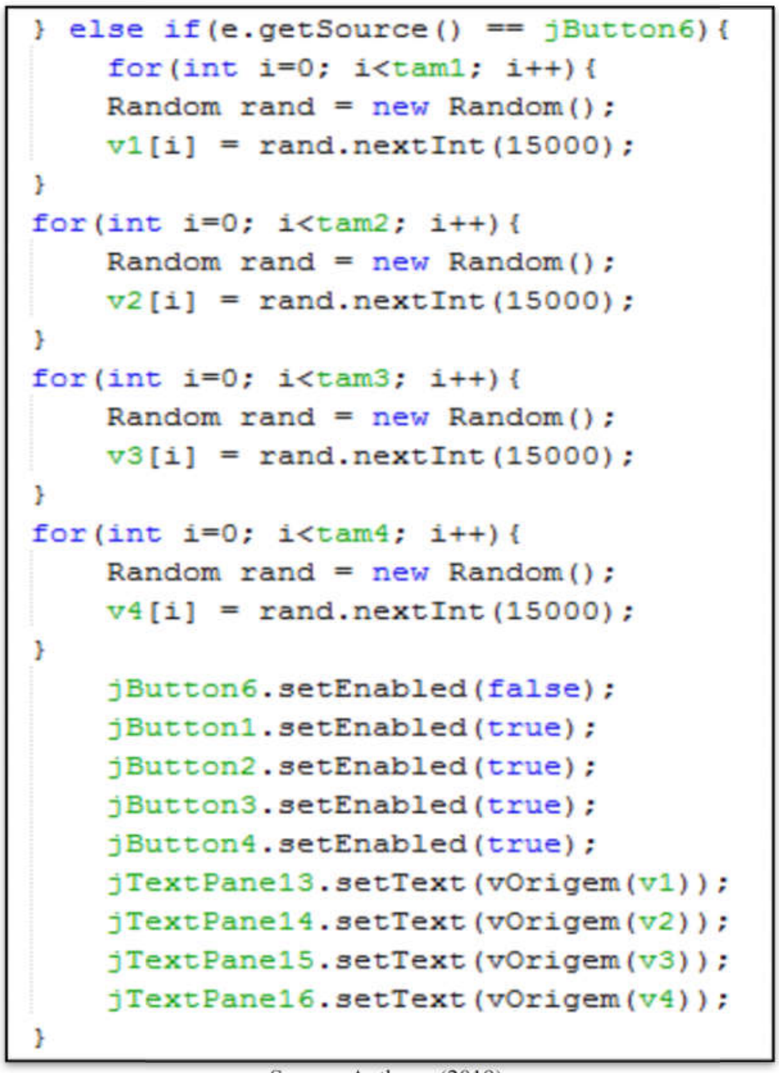
Fig. 7. APS Class



Vector 1	Vector 2	Vector 3	Vector 4
79310	78525	73660	99498
65449	35075	47466	50195
55058	79418	7123	9591
63257	51417	9642	43487

Source: Authors, (2019).

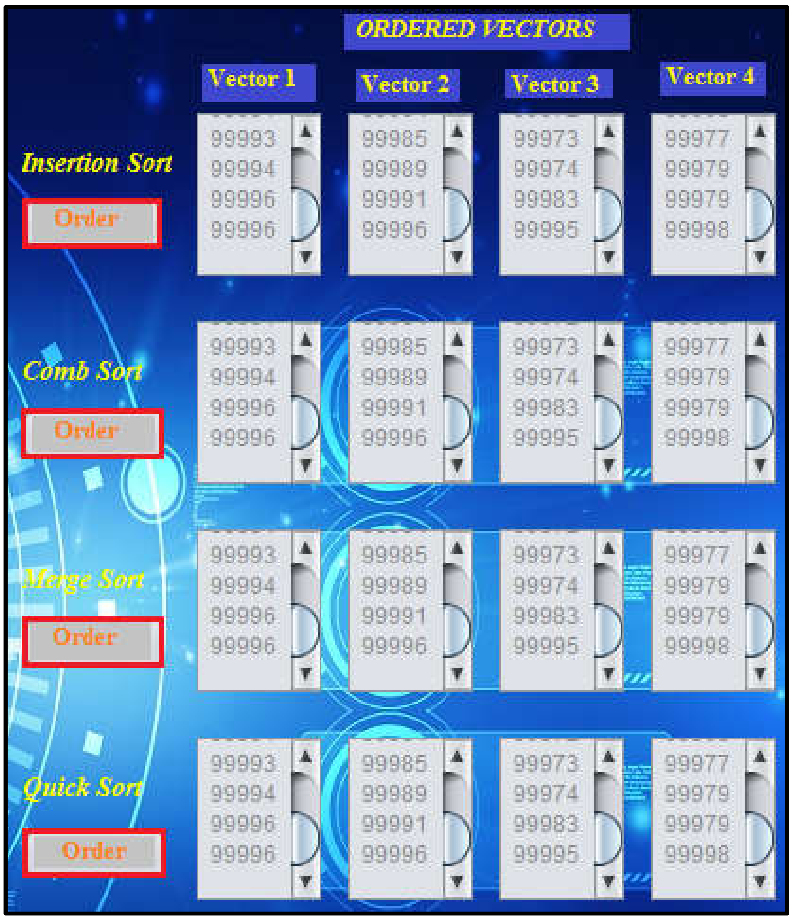
Fig. 8. Unordered vectors



```
} else if(e.getSource() == jButton6){
    for(int i=0; i<tam1; i++){
    Random rand = new Random();
    v1[i] = rand.nextInt(15000);
}
for(int i=0; i<tam2; i++){
    Random rand = new Random();
    v2[i] = rand.nextInt(15000);
}
for(int i=0; i<tam3; i++){
    Random rand = new Random();
    v3[i] = rand.nextInt(15000);
}
for(int i=0; i<tam4; i++){
    Random rand = new Random();
    v4[i] = rand.nextInt(15000);
}
    jButton6.setEnabled(false);
    jButton1.setEnabled(true);
    jButton2.setEnabled(true);
    jButton3.setEnabled(true);
    jButton4.setEnabled(true);
    jTextPane13.setText(vOrigem(v1));
    jTextPane14.setText(vOrigem(v2));
    jTextPane15.setText(vOrigem(v3));
    jTextPane16.setText(vOrigem(v4));
}
```

Source: Authors, (2019).

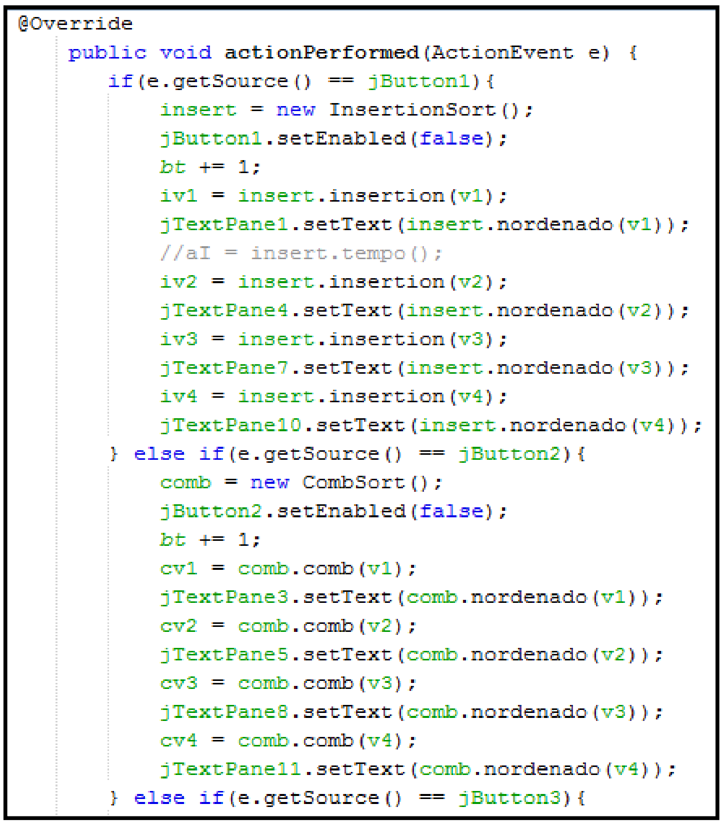
Fig. 9. Filling the vectors



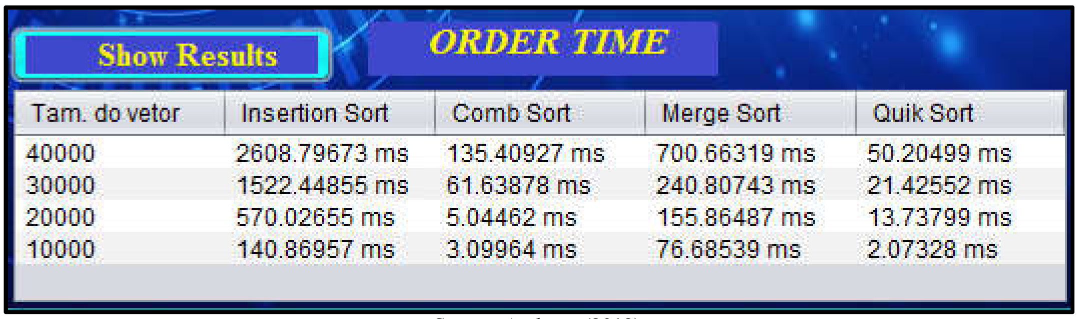
```
99993  99985  99973  99977
99994  99989  99974  99979
99996  99991  99983  99979
99996  99996  99995  99998

99993  99985  99973  99977
99994  99989  99974  99979
99996  99991  99983  99979
99996  99996  99995  99998

99993  99985  99973  99977
99994  99989  99974  99979
99996  99991  99983  99979
99996  99996  99995  99998

99993  99985  99973  99977
99994  99989  99974  99979
99996  99991  99983  99979
99996  99996  99995  99998
```

Source: Authors, (2019).

Fig. 10. Ordered vectors



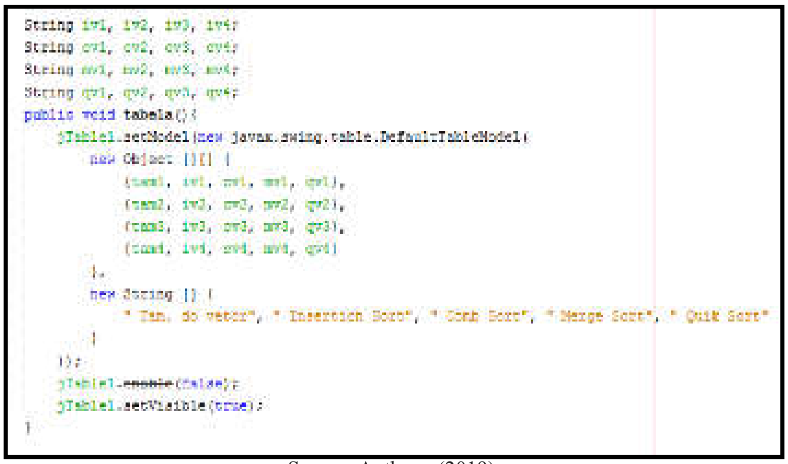
```
@Override
    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == jButton1){
            insert = new InsertionSort();
            jButton1.setEnabled(false);
            bt += 1;
            iv1 = insert.insertion(v1);
            jTextPane1.setText(insert.nordenado(v1));
            //aI = insert.tempo();
            iv2 = insert.insertion(v2);
            jTextPane4.setText(insert.nordenado(v2));
            iv3 = insert.insertion(v3);
            jTextPane7.setText(insert.nordenado(v3));
            iv4 = insert.insertion(v4);
            jTextPane10.setText(insert.nordenado(v4));
        } else if(e.getSource() == jButton2){
            comb = new CombSort();
            jButton2.setEnabled(false);
            bt += 1;
            cv1 = comb.comb(v1);
            jTextPane3.setText(comb.nordenado(v1));
            cv2 = comb.comb(v2);
            jTextPane5.setText(comb.nordenado(v2));
            cv3 = comb.comb(v3);
            jTextPane8.setText(comb.nordenado(v3));
            cv4 = comb.comb(v4);
            jTextPane11.setText(comb.nordenado(v4));
        } else if(e.getSource() == jButton3){
```



Tam. do vetor	Insertion Sort	Comb Sort	Merge Sort	Quik Sort
40000	2608.79673 ms	135.40927 ms	700.66319 ms	50.20499 ms
30000	1522.44855 ms	61.63878 ms	240.80743 ms	21.42552 ms
20000	570.02655 ms	5.04462 ms	155.86487 ms	13.73799 ms
10000	140.86957 ms	3.09964 ms	76.68539 ms	2.07328 ms

Source: Authors, (2019).

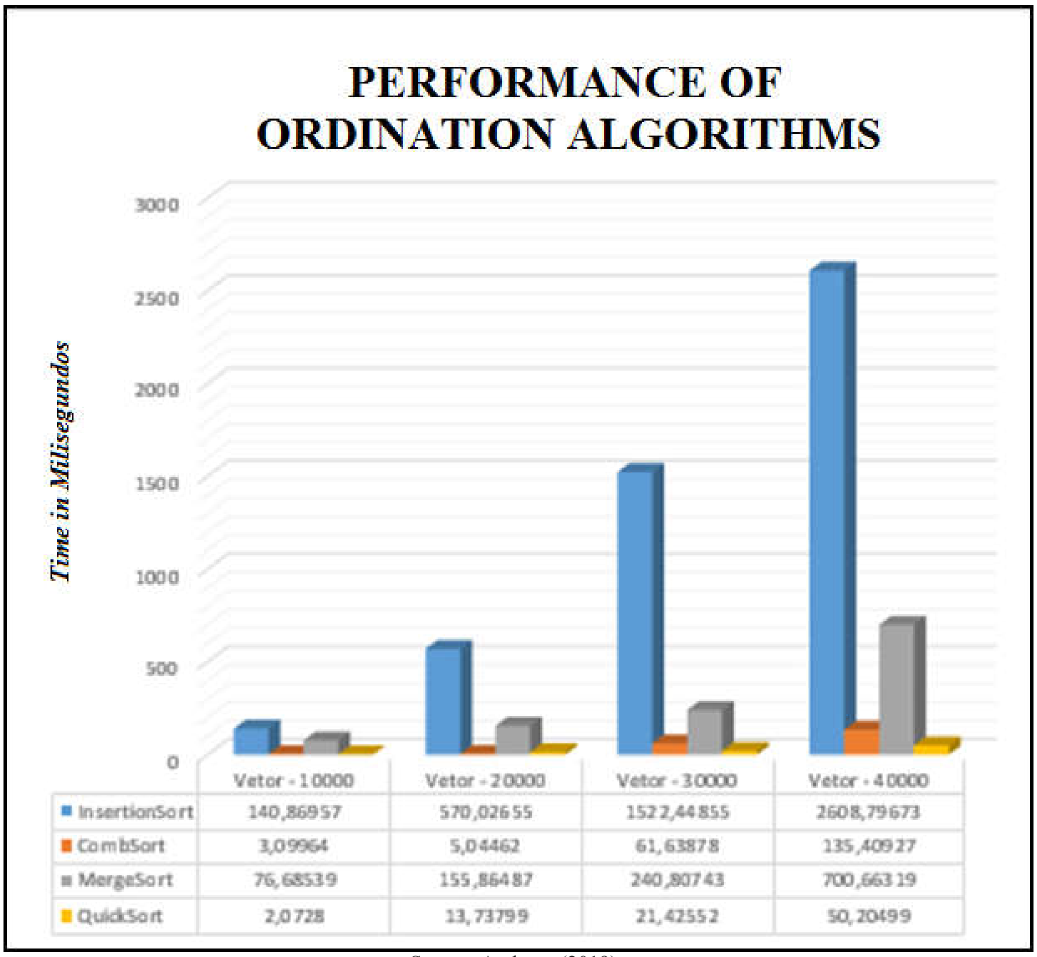
Fig. 12. Table of results



```
String iv1, iv2, iv3, iv4;
String cv1, cv2, cv3, cv4;
String mv1, mv2, mv3, mv4;
String qv1, qv2, qv3, qv4;
public void tabela(){
    jTable1.setModel(new javax.swing.table.DefaultTableModel(
        new Object [][] {
            {tam1, iv1, cv1, mv1, qv1},
            {tam2, iv2, cv2, mv2, qv2},
            {tam3, iv3, cv3, mv3, qv3},
            {tam4, iv4, cv4, mv4, qv4}
        },
        new String [] {
            " Tam. do vetor", " Insertion Sort", " Comb Sort", " Merge Sort", " Quik Sort"
        }
    ));
    jTable1.enable(false);
    jTable1.setVisible(true);
}
```

Source: Authors, (2019).

Fig. 13. Result table method



	Vetor - 10000	Vetor - 20000	Vetor - 30000	Vetor - 40000
■ InsertionSort	140,86957	570,02655	1522,44855	2608,79673
■ CombSort	3,09964	5,04462	61,63878	135,40927
■ MergeSort	76,68539	155,86487	240,80743	700,66319
■ QuickSort	2,0728	13,73799	21,42552	50,20499

Source: Authors, (2019).

Fig. 14. Table of results

Next we have a graph of the four algorithms evaluated, it is important to make explicit that some values were rounded, but this does not change much the graph and the dimension of the obtained times, because the evaluation of the group has already been made in relation to the various tests done with the data, where the time of each test changes, however the percentage variation between the organization of the data by the algorithms varies very little.

Final considerations: With all the results at hand and all analysis done on top of them, it is concluded that the insertionSort sorting algorithm can be used for sorting with small amounts of data, since the time of the ordering process with small quantities is negligible. In turn mergesort did not obtain satisfactory results in our tests, although this is considered a sophisticated algorithm, this is due to the several operations of separating the elements of the vectors, leaving it somewhat slow and consuming a lot of computer memory. The quicksort method is great for ordering very high amounts, since it organizes the data very quickly. It is inevitable to fail to comment on the optimal performance of the combSort algorithm, which is classified as simple but of incredible performance. The software had good and satisfactory results to the point of view of the group, where it brought a comparative of the methods used and easy assimilation by those who see the table generated by the application.

Acknowledgement

The Paulista University - UNIP and the Galileo Institute of Technology and Education of the Amazon - ITEGAM for technical and scientific support and collaboration.

REFERENCES

Alves, Rafael Fogliato et al., 2018. Análise DE Complexidade Computacional DOS Métodos DE Integração Intervalar. Anais do Salão Internacional de Ensino, Pesquisa e Extensão, v. 9, n. 3.

Branco, Fábio José Ceron; Fuchigami, 2017. Helio Yochihiro. Métodos de alto rendimento e baixa complexidade em flowshop. Revista GEPROS, v. 12, n. 4, p. 32.

Burke. 2014. Comb sort algorithm. Programering, página da web 27 novembro.

Celes, W.; Rangel, J. L. 2002. Apostila de Estrutura de Dados. Rio de Janeiro: PUC-RIO - Curso de Engenharia.

Cormem, T. H. et al. 2002. Algoritmos Teoria e Prática, Tradução da 2º Edição Americana. Tradução de Vandenberg D. de Souza. 2º. ed. Rio de Janeiro: Elsevier.

Da Silva Nascimento, Jonathan; Mozzaquatro, 2016. Patricia Mariotto; Antoniazzi, Rodrigo Luiz. Análise e Comparação de Algoritmos Implementados em Java. Simpósio de Pesquisa e Desenvolvimento em Computação, v. 1, n. 1.

Desurvire, E. 2009. Classical and Quantum Information Theory. Cambridge: Cambridge University Press.

Goldreich, Oded. 2008. Computational Complexity: A Conceptual Perspective, Cambridge University Press.

Laureano, M. A. P. 2012. Estrutura de Dados com Algoritmos e C. Curitiba: Brasport.

Oliveira, Á. B. 2002. Métodos de Ordenação Interna. Visual Book, São Paulo, 1st edição.

Pinheiro, Liliane Vieira et al., 2017. O desenvolvimento de coleções em bibliotecas universitárias na perspectiva dos desafios da pós-modernidade: diretrizes sob o olhar da teoria da complexidade e da análise do domínio.

Piqueira, José Roberto Castilho. 2016. Complexidade computacional e medida da informação: caminhos de Turing e Shannon. Estudos Avançados, v. 30, n. 87, p. 339-344.

Vargas, R. B., Garcia, B. B. 2004. Aula 10.2: Dividir e Conquistar - Mergesort. Universidade Federal do Espirito Santo. ISSN INF 02779 Análise de Algoritmos.

Viana, D. 2016. Conheça os principais algoritmos de ordenação. Treina Web Blog, 26 novembro.

Vilela, Bruno Azevedo. 2016. Análise da complexidade de espaço para um algoritmo de K (1)-Validade.

****