FRAMEWORK PARA A IMPLEMENTAÇÃO DE ALGORITMOS DE ORDENAÇÃO

Natasha Girardi Busnardo
Instituto Federal Catarinense
bccnatasha@gmail.com

Resumo. *Este artigo tem como tema o desenvolvimento de um framework Java que realiza a implementação dos algoritmos de ordenação Bubble Sort, Insertion Sort Selection Sort, Merge Sort e Quick Sort.*

Palavras-chave: *Ordenação; Java.*

1 INTRODUÇÃO

Com a manipulação de grandes números de dados, a ordenação se torna um mecanismo eficiente para realizar consultas. De acordo com LAFORE, “Ordenar dados pode também ser um passo preliminar para pesquisá-los”.

Uma pesquisa eficiente proporciona a otimização de tempo

A ordenação, se aplicada de maneira incorreta, pode ocasionar em um tempo de execução muito maior do que realmente necessário.

2 FERRAMENTAS UTILIZADAS NO DESENVOLVIMENTO

2.1 JAVA

Java é uma linguagem de programação orientada a objetos elaborada pela Sam Microsystems em 1985. Se difere de muitas linguagens por possuir seu próprio interpretador que realiza a leitura do código para que o sistema operacional o execute, o JVM. Sua principal característica é a sua multiplataforma, já que possui inúmeras aplicações e vasta quantidade de bibliotecas disponíveis aos desenvolvedores.

2.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

POO, ou Programação Orientada a Objetos, é um paradigma de programação que tem fundamenta-se no conceito de objeto. O objeto é uma entidade que possui características e comportamentos. Se difere da programação estrutural, pois não segue um fluxo sequencial e sim uma estrutura onde todos os objetos interagem entre si.

3 ALGORITMOS DE ORDENAÇÃO

3.1 BUBBLE SORT

São realizadas comparações entre os dados armazenados em um vetor de tamanho n. Cada elemento de posição i será comparado com o elemento de i + 1 e quando a ordenação imposta é encontrada, uma troca de posições de elementos é feita, sendo executado um loop com a quantidade de elementos do vetor e um segundo laço que percorre da primeira à penúltima posição no vetor.

A ordenação pelo método da bolha é mais lenta, pois faz uma comparação entre todos os elementos diversas vezes, porém é conceitualmente a mais simples.

Aplicando a ideia do algoritmo em um vetor de tamanho n, ele realizará $n(n-1) = n^2-n$ comparações.

$$n^2 - n \leq cn^2,\ c = 1,\ n \geq 1.$$

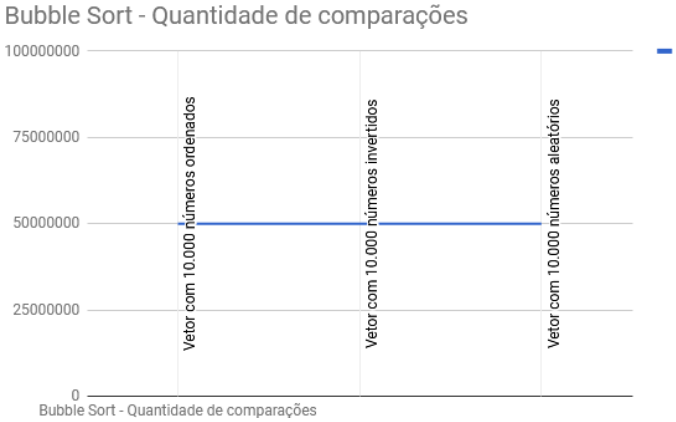
Neste algoritmo, qualquer que seja o vetor de entrada, o algoritmo irá se comportar da mesma maneira, realizando todas as comparações, mesmo que desnecessárias.

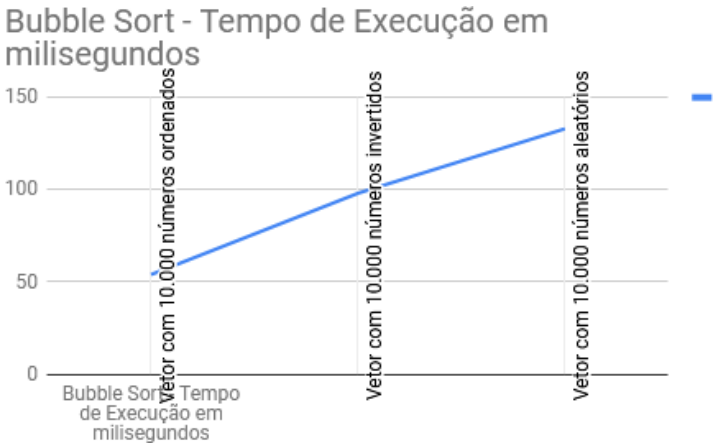
Código-Fonte

```
1   public int[] bubbleSort(int vetor[]) {
2       this.tempototal = sysout.currentTimeMillis();
3           for (int  i=0; i = vetor.length; i >= 1; i--) {
4               for (int j = 1; j < i; j++) {
5                   this.comparacoes++;
6                       if (vetor[j - 1] > vetor[j]) {
7                           int aux = vetor[j];
8                           vetor[j] = vetor[j - 1];
9                           vetor[j - 1] = aux;
10                          this.trocas++;
11                      }
12              }   }
```

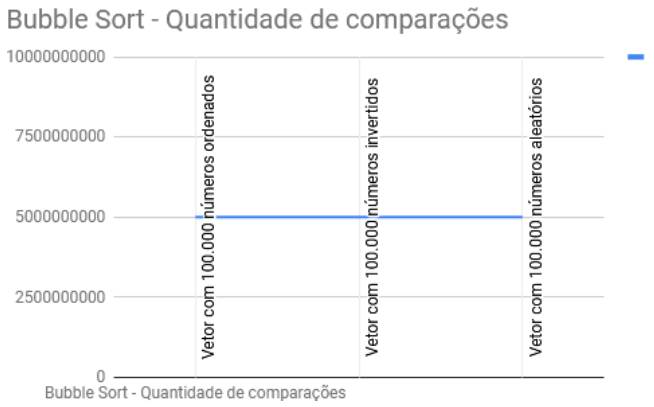
Gráficos de Desempenho

- Em vetores de 10.000 elementos:





- Em vetores de 100.000 elementos:



3.2 INSERTION SORT

São realizadas comparações a partir do segundo elemento do vetor para iniciar comparações, se estendendo até o último elemento. O laço será executado enquanto houverem elementos à esquerda do número eleito para comparações e a posição que diz respeito ao critério de ordenação não for encontrado. O número eleito está na posição i. Os números à esquerda estão nas posições de i-l à 0.

Neste método, o seu pior uso seria no momento em que o vetor de entrada se encontra com elementos com ordem decrescente e se busca a ordenação crescente. Onde *x[j]* é comparado com cada elemento no subvetor ordenado. Onde o número de execução será descoberto através da equação:

$$T\ (n)=((1+n)\ n)/2-1$$

A melhor utilização deste algoritmo se tem quando os valores já estão ordenados. Para cada j=0,1,...,N-2, se obtém a condição x[j]>eleito.

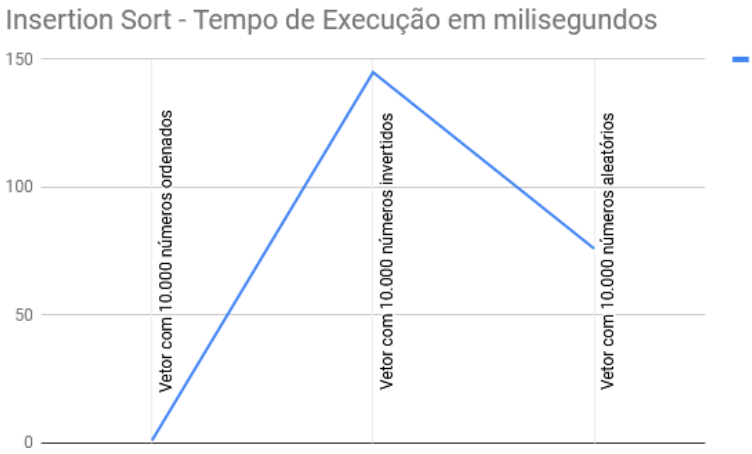
A aplicação do ISP se deu através das duas interfaces geradas, assim, a classe Desenvolvedor não necessitou incitar um método que não lhe teria utilidade.

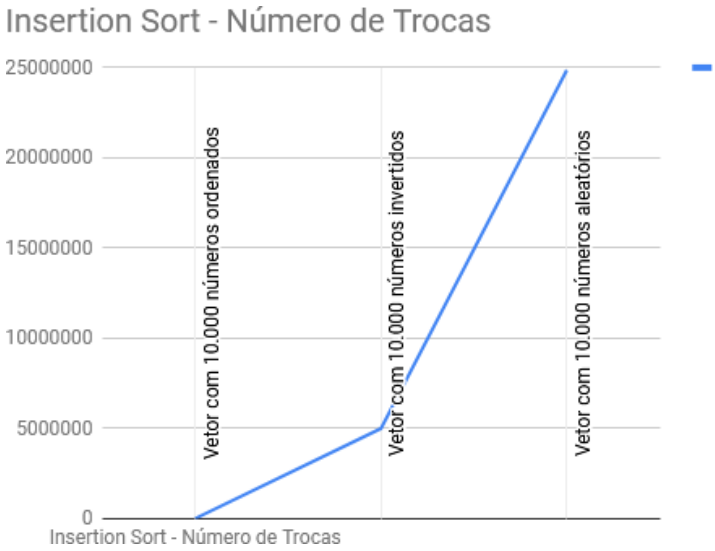
Código-Fonte

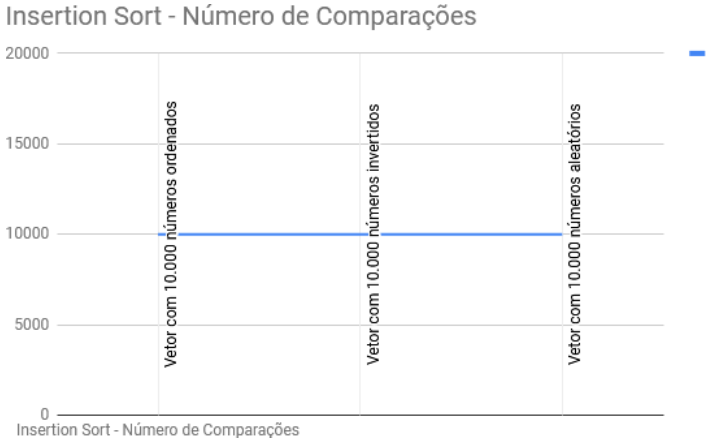
```
1   public int[] insertionSort(int vetor[]) {
2       this.tempototal = sysout.currentTimeMillis();
3           for (int i = 0; i < vetor.length; i >= 1; i--) {
4               int atual = vetor[i];
5               int. j = i -1;
6               this.comparacoes++;
7              while (j >= 0 && vetor[j] >= atual) {
8                  vetor[j + 1] = vetor[j];
9                  j--;
10                this.trocas++;}
11             vetor[j + 1] = atual;
12             this.trocas++;
```

Gráficos de Desempenho

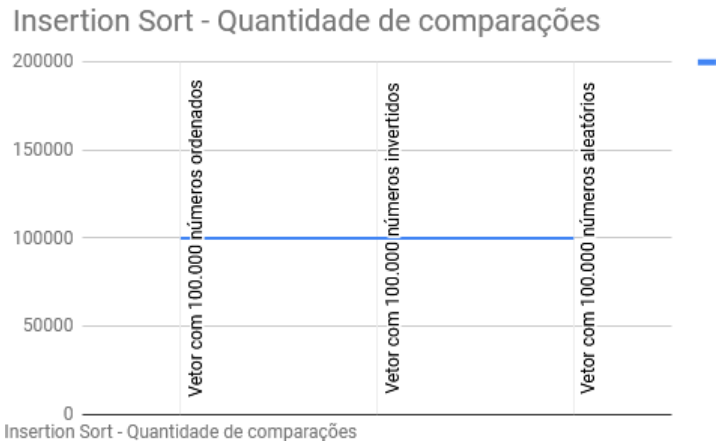
- Em vetores de 10.000 elementos:

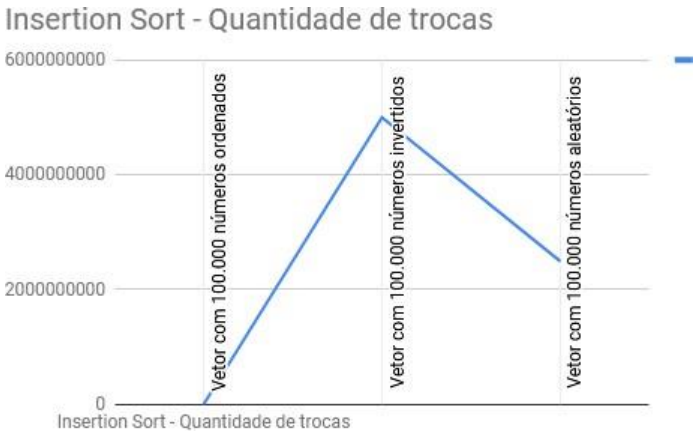


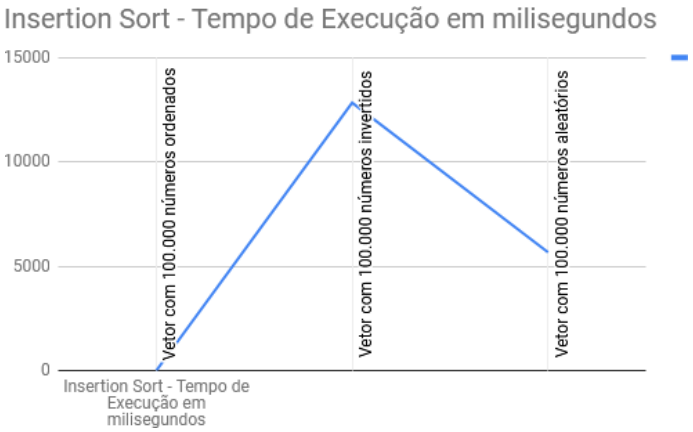




- Em vetores de 100.000 elementos:







3.3 SELECTION SORT

São efetuadas comparações com o menor ou maior valor, número dentre aqueles que estão à direita do selecionado. Quando um número satisfaz as condições da ordenação, ele trocará de posição com o selecionado, deixando à sua esquerda todos os elementos já ordenados. O algoritmo irá possuir um laço com as comparações do primeiro ao penúltimo elemento.

O número eleito, assim como em demais métodos de ordenação, se encontra na posição i. Sendo que à direita de i estão os números i+1 a n-1.

A fórmula para o tempo de execução deste algoritmo se dá por:

$$T(n)=n\,Z\,/\,2-n/2$$

Podemos assim concluir que o tempo de execução, independente do vetor de entrada, irá se comportar da mesma maneira.

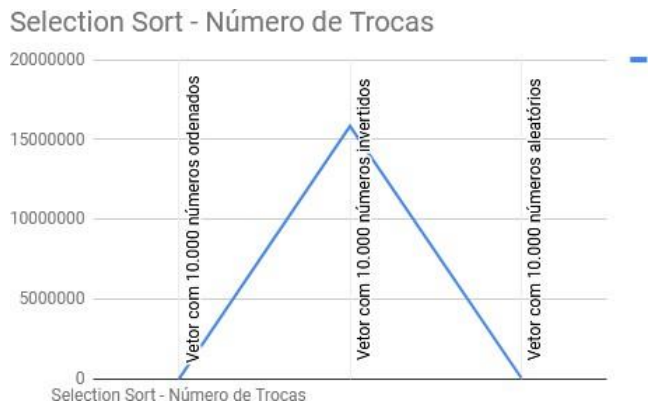
```
public int[] selectionSort(int[] vetor) {
                this.tempototal = System.currentTimeMillis();
```

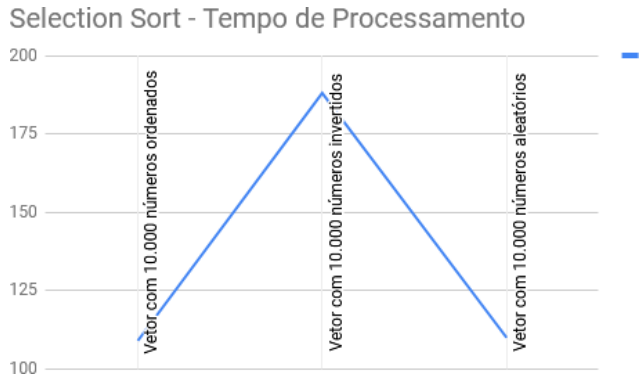
Código-Fonte

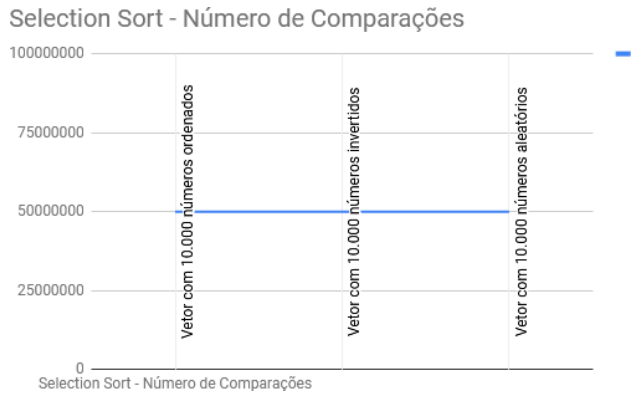
```
1   for (int i = 0; i < vetor.length; i++) {
2           int indiceMinimo = i;
3           for (int j = i + 1; j < vetor.length; j++) {
4             this.comparacoes++;
5               if (vetor[j] < vetor[indiceMinimo]) {
6                   indiceMinimo = j;
7                   this.trocas++;
8               }
9           }
10          int tmp = vetor[indiceMinimo];
11          vetor[indiceMinimo] = vetor[i];
12          vetor[i] = tmp;
13      }
14      this.tempototal = System.currentTimeMillis() - this.tempototal;
15      return vetor;
16  }
```

Gráficos de Desempenho

- Em vetores de 10.000 elementos:

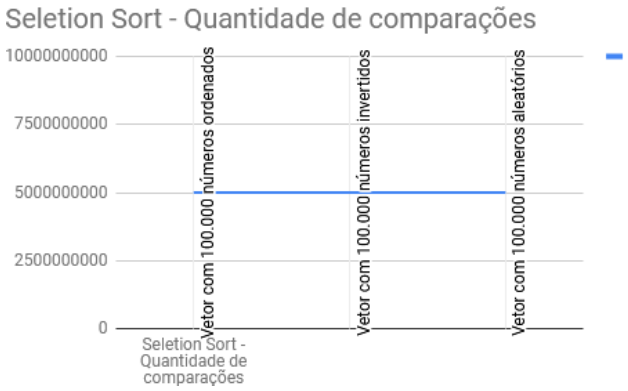


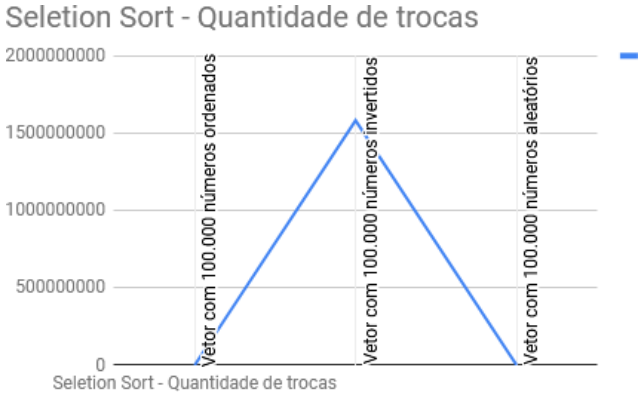

Selection Sort - Tempo de Processamento
200
175
150
125
100
Vetor com 10.000 números ordenados
Vetor com 10.000 números invertidos
Vetor com 10.000 números aleatórios


Selection Sort - Número de Comparações
100000000
75000000
50000000
25000000
0
Vetor com 10.000 números ordenados
Vetor com 10.000 números invertidos
Vetor com 10.000 números aleatórios
Selection Sort - Número de Comparações

2021. Ano 8, Vol. 8. ISSN: 2319-0655

- Em vetores de 100.000 elementos:





3.4 MERGE SORT

O vetor principal é dividido em subvetores com metade do tamanho do vetor inicial através de um método recursivo. Essa segmentação ocorre até que se tenha um vetor com apenas um elemento, onde os mesmos se encontram ordenados e intercalados. No algoritmo, é aplicada a técnica dividir para conquistar, também utilizada no método de busca binária, onde temos os seguintes passos:

- Dividir a sequência de n elementos que irão ser ordenados em duas subse-quênciasde n/2 elementos;
- Ordenar as subsequências recursivamente;
- Intercalar as duas subsequências ordenadas para resolver o problema

principal;Seu tempo de execução pode ser expresso através de:

$$T(n)=2T(n/2)+n$$

Assim como o método Selection Sort, independente da entrada, o algoritmo trabalhará demesmo modo.

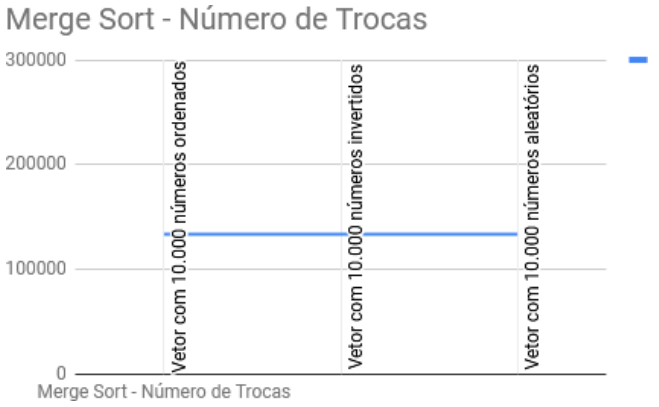
Código-Fonte

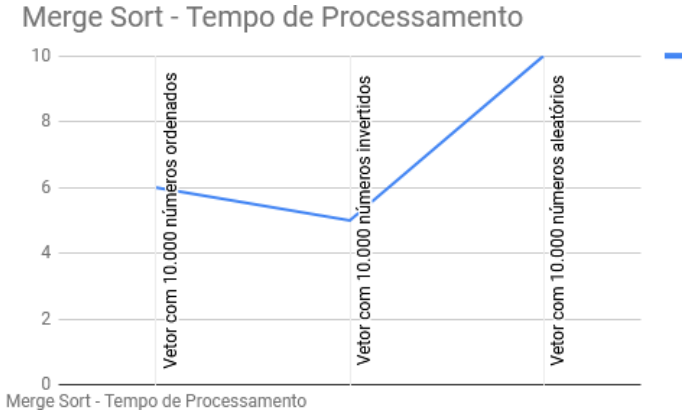
```
1  public int[] sort(int[] array) {
2
3      if (array.length <= 1) {
4
5          return array;
6      }
7      int meio = array.length / 2;
8      int[] dir = array.length % 2 == 0 ? new int[meio] : new int[meio + 1];
9      int[] esq = new int[meio];
10
11     int[] aux = new int[array.length];
12
13     for (int i = 0; i < meio; i++) {
14         esq[i] = array[i];
15     }
16      int auxIndex = 0;
17     for (int i = meio; i < array.length; i++) {
18         dir[auxIndex] = array[i];
19         auxIndex++;
20
```

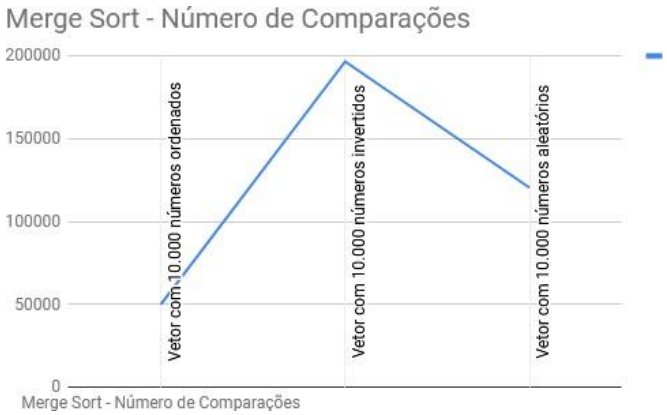
```
21        }
22        esq = sort(esq);
23        dir = sort(dir);
24        aux = mergesort(esq, dir);
25        return aux;
26    }
```

Gráficos de Desempenho

- Em vetores de 10.000 elementos:







- Em vetores de 100.000 elementos:

3.5 QUICK SORT

O vetor é fragmentado em duas partes através de uma recursão. Assim como a ordenação Merge Sort, esta fragmentação ocorre até que o vetor fique com apenas um elemento. Neste algoritmo se obtém a ordenação através do método dividir para conquistar. Esse método é realizado da seguinte maneira:

- Dividir o vetor X[p..r] em dois subvetores X[p..q] e X[q+1..r], onde cada elemento de X[p..q] é menor ou igual a cada elemento de X[q+1..r], q -ou pivô- é o

elemento que se encontra na metade do vetor original, sendo que os menores valores vão à esquerda do pivô e os maiores à direita;

- Ordenar os dois subvetores por chamadas recursivas;

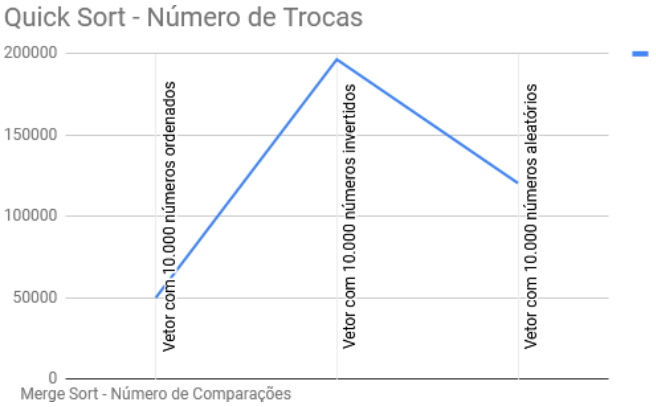
Assim como o método *Merge Sort*, o tempo de processamento está relacionado com o balanceamento do particionamento. O pior caso para o *quick sort* ocorre quando se é desenvolvida com n-1 elementos e outra com somente um elemento.

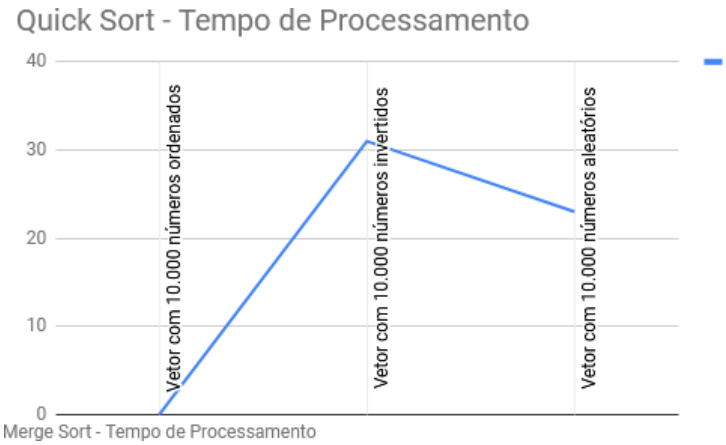
Código-Fonte

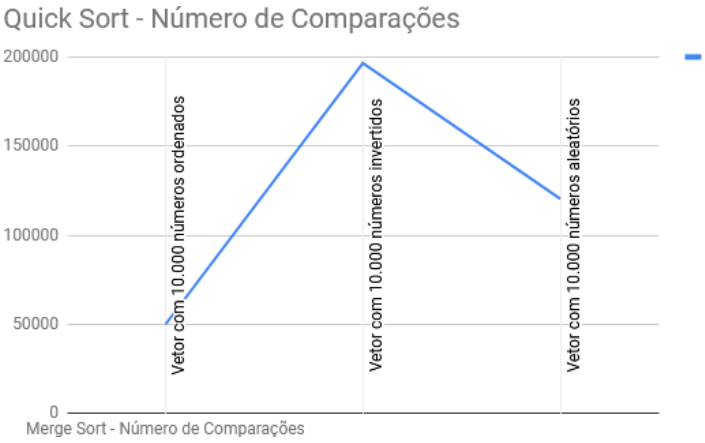
```
1  public int[] quicksort(int vet[], int esq, int dir){
2            int pivo = esq, i,ch,j;
3            for(i=esq+1;i<=dir;i++){
4                j = i;
5                this.comparacoes++;
6                if(vet[j] < vet[pivo]){
7                    ch = vet[j];
8                    while(j > pivo){
9                        vet[j] = vet[j-1];
10                       j--;
11                       this.trocas++;
12                   }
13                   vet[j] = ch;
14                   pivo++;
15               }
           }
           if(pivo-1 >= esq){
             return quicksort(vet,esq,pivo-1);
           }
           if(pivo+1 <= dir){
             return quicksort(vet,pivo+1,dir);
           }
           return vet;
       }
```

Gráficos de Desempenho

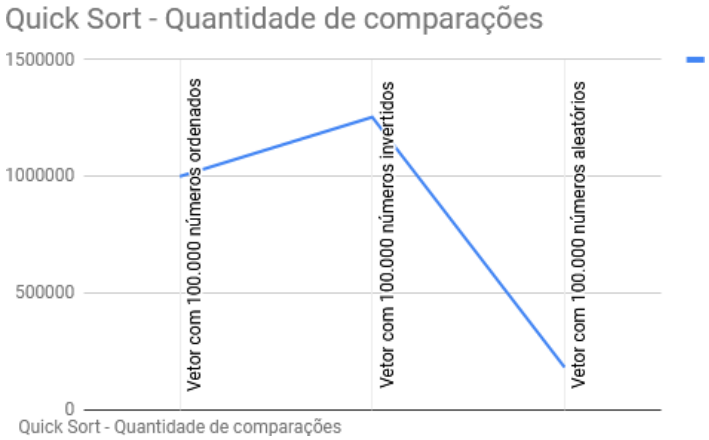
- Em vetores de 10.000 elementos:

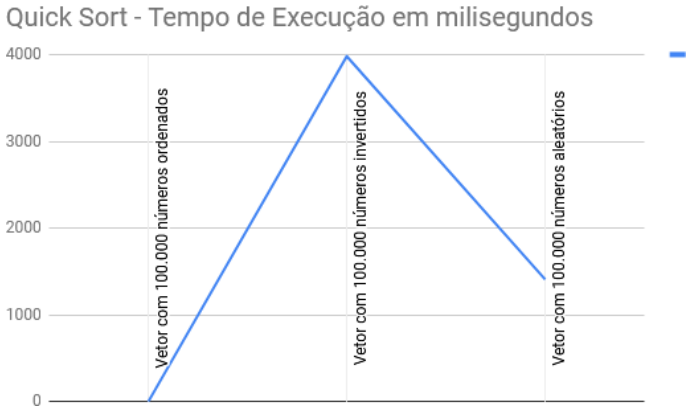






- Em vetores de 100.000 elementos:





4 CONSIDERAÇÕES FINAIS

A ordenação de dados é importante e por muito tempo foi demasiadamente demora. Após duradouras pesquisas na área de manipulação de dados, os métodos para rearranjo de dados se sofisticaram.

Cada método, seja ele simples ou sofisticado, proporciona vantagens e desvantagens. Métodos mais simples, como ordenação por seleção, mesmo que relativamente lentas em comparação à outros métodos, são de fácil compreensão e se destacam em certas circunstâncias.

5 REFERÊNCIAS

ASCENCIO, Ana Fernanda Gomes e ARAÚJO, Graziela Santos de. **Estrutura de dados: Algoritmos, analise da complexidade e implementações em java e C/C++**. São Paulo:Perarson Prentice Halt, 2010.

LAFORE, R. **Estrutura de dados e algoritmos em Java**, 2.ed. Rio de Janeiro: Ciência ModernaLtda, 2004.

MAGALHÃES, P. A.; TIOSSO, F. CÓDIGO LIMPO: padrões e técnicas no desenvolvimento de software. **Revista Interface Tecnológica**, *[S. l.]*, v. 16, n. 1, p. 197-207, 2019. Disponível em: https://revista.fatectq.edu.br/index.php/interfacetecnologica/article/view/597. Acesso em: 24 maio. 2021.