

UNIVERSIDADE PAULISTA – UNIP

**GABRIEL DE ALMEIDA BATISTA
FELIPE DA SILVA BORGES NEVES
CÉSAR MAGNUN OLIVEIRA
JOSÉ VITOR ZANONI DA COSTA**

DESENVOLVIMENTO DE SISTEMA:
ANÁLISE DE PERFORMANCE DE ALGORITMO DE ORDENAÇÃO DE DADOS

**SÃO PAULO
2016**

**GABRIEL DE ALMEIDA BATISTA
FELIPE DA SILVA BORGES NEVES
CÉSAR MAGNUN OLIVEIRA
JOSÉ VITOR ZANONI DA COSTA**

DESENVOLVIMENTO DE SISTEMA:
ANÁLISE DE PERFORMANCE DE ALGORITMO DE ORDENAÇÃO DE DADOS

Atividade prática supervisionada e apresentada ao curso Ciência da Computação, para fins de conhecimento na área.

Orientador: Prof. César Augusto Cardoso Caetano.

**SÃO PAULO
2016**

DEDICATÓRIA

Dedicamos este trabalho primeiramente a Deus por ter nos dado vida até este presente momento e aos nossos pais que nos educou, com força e amor incondicionais.

AGRADECIMENTOS

Agradecemos em primeiro lugar a Deus por ser a base das nossas conquistas.

Aos nossos pais, por acreditar e terem interesse em nossas escolhas, apoiando-nos e esforçando-se junto a nós, para que supríssemos todas elas.

Ao professor César Augusto Cardoso Caetano, pela dedicação em suas orientações prestadas na elaboração deste trabalho, nos incentivando e colaborando no desenvolvimento de nossas ideias.

Ao nosso amigo Clederson Bispo da Cruz, por ter nos auxiliado, desde o princípio, na correção gramatical.

*“O fracasso é simplesmente uma
oportunidade para começar de novo, desta
vez de forma mais inteligente”.*

(Henry Ford)

RESUMO

A ordenação é o processo de organizar um conjunto de informações segundo uma determinada ordem. Se o conjunto estiver ordenado é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo, a pesquisa binária. O objetivo da ordenação na manipulação de dados é facilitar a busca pela informação, dando a possibilidade de uma pesquisa binária. O objetivo deste trabalho é avaliar de forma clara o conceito de ordenação, sua utilização e sua aplicabilidade, por meio de sete algoritmos de ordenação em um software desenvolvido na linguagem Java. O trabalho realizado teve caráter qualitativo. Fundamentado em obras publicadas por estudiosos especialistas, que já apontaram os conceitos, usos, aplicações, desempenho e falhas de certas técnicas de ordenação. A ordenação é um conceito totalmente necessário e importante nos meios de manipulação de dados e informações que utilizamos hoje, já que há um crescente aumento de busca e uma frequente atualização de dados entre os usuários e servidores que necessitam de busca rápida e eficaz.

Palavras-chave: Análise de desempenho; Software; Linguagem Java.

ABSTRACT

Sorting is the process of organizing a set of information in a particular order. If the set is ordered it is possible to use more efficient search algorithms, such as binary search. The purpose of ordering the data manipulation is to facilitate the search for information, giving the possibility of a binary search. The objective of this study is to evaluate clearly the concept of ordination, its use and its applicability through seven sorting algorithms in software developed in Java. The work was qualitative. Based on works published by expert scholars who have pointed out the concepts, uses, applications, performance and fault ordering certain techniques. Ordination is a totally necessary and important concept in data manipulation means and information that we use today, since there is an increasing search and frequent update of data between users and servers that need fast and effective searches.

Keywords: Performance Analysis; Software; Java Language.

LISTA DE ILUSTRAÇÕES

Imagem 01 – Opção de manipulação.	Pag. 38.
Imagem 02 – Obtenção de imagem.	Pag. 39.
Imagem 03 – Solicitação de nome.	Pag. 40.
Imagem 04 – Obtenção de número.	Pag. 41.
Imagem 05 – Método <i>getTxtNumber</i> .	Pag. 41.
Imagem 06 – Método de ação.	Pag. 42.
Imagem 07 – Classe <i>ImagemPadrao</i> .	Pag. 43.
Imagem 08 – Lista de imagem padrão.	Pag. 43.
Imagem 09 – Classe <i>NumeroPadrao</i> .	Pag. 44.
Imagem 10 – Método <i>bubble</i> .	Pag. 45.
Imagem 11 – Método <i>shake</i> .	Pag. 46.
Imagem 12 – Método <i>select</i> .	Pag. 48.
Imagem 13 – Método <i>insert</i> .	Pag. 49.
Imagem 14 – Método <i>recMergeSortInt</i> .	Pag. 50.
Imagem 15 – Método <i>mergeInt</i> .	Pag. 50.
Imagem 16 – Método <i>recQuickSortInt</i> e <i>medianOf3Int</i> .	Pag. 51.
Imagem 17 – Método <i>partitionIntInt</i> e <i>insertionSortInt</i> .	Pag. 52.
Imagem 18 – Método <i>partitionIntInt</i> e <i>insertionSortInt</i> .	Pag. 53.
Imagem 19 – Método <i>heapSortInt</i> .	Pag. 54.
Imagem 20 – Método <i>peneiraInt</i> .	Pag. 55.
Imagem 21 – Lista de imagem antes.	Pag. 56.
Imagem 22 – Lista de imagem após o <i>bubble</i> .	Pag. 56.
Imagem 23 – Lista de imagem após o <i>shake</i> .	Pag. 57.
Imagem 24 – Lista de imagem após o <i>select</i> .	Pag. 57.
Imagem 25 – Lista de imagem após o <i>insert</i> .	Pag. 57.
Imagem 26 – Lista de imagem após o <i>merge</i> .	Pag. 58.
Imagem 27 – Lista de imagem após o <i>quick</i> .	Pag. 58.
Imagem 28 – Lista de imagem após o <i>heap</i> .	Pag. 59.
Imagem 29 – Lista de números antes.	Pag. 59.
Imagem 30 – Lista de números após o <i>bubble</i> .	Pag. 59.
Imagem 31 – Lista de números após o <i>shake</i> .	Pag. 60.
Imagem 32 – Lista de números após o <i>select</i> .	Pag. 60.

Imagem 33 – Lista de números após o <i>insert</i> .	Pag. 61.
Imagem 34 – Lista de números após o <i>merge</i> .	Pag. 61.
Imagem 35 – Lista de números após o <i>quick</i> .	Pag. 62.
Imagem 36 – Lista de números após o <i>heap</i> .	Pag. 64.
Imagem 37 – Gabriel de Almeida Batista.	Pag. 99.
Imagem 38 – Felipe da Silva Borges Neves.	Pag. 100.
Imagem 39 – César Magnun Oliveira.	Pag. 101.
Imagem 40 – José Vitor Zanoni da Costa.	Pag. 102.

LISTA DE TABELAS E GRÁFICOS

Tabela 01 – Interações do <i>bubble</i> .	Pag. 69.
Tabela 02 – Interações do <i>shake</i> .	Pag. 72.
Tabela 03 – Interações do <i>select</i> .	Pag. 73.
Tabela 04 – Interações do <i>insert</i> .	Pag. 75.
Tabela 05 – Interações do <i>merge</i> .	Pag. 77.
Tabela 06 – Interações do <i>quick</i> .	Pag. 78.
Tabela 07 – Interações do <i>heap</i> .	Pag. 80.

Gráfico 01 – Interações do <i>bubble</i> .	Pag. 69.
Gráfico 02 – Interações do <i>shake</i> .	Pag. 71.
Gráfico 03 – Interações do <i>select</i> .	Pag. 73.
Gráfico 04 – Interações do <i>insert</i> .	Pag. 74.
Gráfico 05 – Interações do <i>merge</i> .	Pag. 76.
Gráfico 06 – Interações do <i>quick</i> .	Pag. 78.
Gráfico 07 – Interações do <i>heap</i> .	Pag. 79.
Gráfico 08 – Ordenação simples de cem dados.	Pag. 81.
Gráfico 09 – Ordenação simples de mil dados.	Pag. 81.
Gráfico 10 – Ordenação simples de cinco mil dados.	Pag. 82.
Gráfico 11 – Ordenação simples de dez mil dados.	Pag. 82.
Gráfico 12 – Ordenação simples de cem mil dados.	Pag. 83.
Gráfico 13 – Ordenação simples de quinhentos mil dados.	Pag. 83.
Gráfico 14 – Ordenação simples de um milhão de dados.	Pag. 84.
Gráfico 15 – Ordenação simples de cinco milhões de dados.	Pag. 84.
Gráfico 16 – Ordenação avançada de cem dados.	Pag. 86.
Gráfico 17 – Ordenação avançada de mil dados.	Pag. 87.
Gráfico 18 – Ordenação avançada de cinco mil dados.	Pag. 87.
Gráfico 19 – Ordenação avançada de dez mil dados.	Pag. 98.
Gráfico 20 – Ordenação avançada de cem mil dados.	Pag. 88.
Gráfico 21 – Ordenação avançada de quinhentos mil dados.	Pag. 89.
Gráfico 22 – Ordenação avançada de um milhão de dados.	Pag. 89.
Gráfico 23 – Ordenação avançada de cinco milhões de dados.	Pag. 90.

LISTA DE ABREVIATURA E SIGLAS

ASCII – *American Standard Code for Information Interchange.*

TDA – Tipo Abstrato de Dado.

TDAs – Tipos Abstratos de Dados.

RAM – *Random Access Memory.*

GB – *Giga Byte.*

HD – *Hard Disk Drive.*

TB – *Tera Byte.*

Pro – Profissional.

IDE – *Integrated Development Environment.*

JDK – *Java Developer's Kit.*

JRE – *Java Runtime Environment.*

JSE – *Java Standard Edition.*

API – *Applications Programming Interface.*

GUI – *Graphic User Interface.*

AWT – *Abstract Window Toolkit.*

CLI – *Comand Line Interface.*

IHC – Interação Humano Computador.

JPG – *Joint Photographic Experts Group.*

n – Tamanho do vetor.

Out – Último elemento do vetor.

In – Primeiro elemento do vetor.

SAMS – *Samsung.*

log – Logarítmico.

LISTA DE SÍMBOLOS

® – Marca registrada.

™ – *Trade Mark*.

@ – Arroba.

SUMÁRIO

1	INTRODUÇÃO.....	23
2	ESTRUTURA DE DADOS	27
2.1	Tipos de dados	29
2.1.1	<i>Tipos primitivos.....</i>	29
2.1.2	<i>Tipo de dados estruturais</i>	29
2.1.3	<i>Tipo ponteiro.....</i>	30
2.1.4	<i>Tipos abstratos de dados</i>	30
2.2	Pesquisa de dados	30
2.3	Ordenação de dados	32
3	DESENVOLVIMENTO	33
3.1	Interface Gráfica (GUI).....	33
3.1.1	<i>Janelas (JFrame).....</i>	34
3.1.2	<i>Rótulos (JLabel).....</i>	35
3.1.3	<i>Botões (JButton).....</i>	35
3.1.4	<i>Painéis (JPanel).....</i>	36
3.1.5	<i>Eventos de Ação (Acition Event)</i>	36
3.1.6	<i>Campos de Texto (JTextField).....</i>	37
3.1.7	<i>Barras de Rolagem (JScrollPane)</i>	37
3.1.8	<i>Listas (JList).....</i>	38
3.2	Obtenção de Dados	38
3.2.1	<i>Arquivos.....</i>	39
3.2.2	<i>Números</i>	40
3.3	Geração de Dados	42
3.3.1	<i>Arquivos.....</i>	42
3.3.2	<i>Números</i>	44
3.4	Algoritmos de ordenação	44
3.4.1	<i>Bubble Sort.....</i>	45
3.4.2	<i>Shake Sort.....</i>	46
3.4.3	<i>Select Sort.....</i>	47
3.4.4	<i>Insert Sort.....</i>	48
3.4.5	<i>Merge Sort.....</i>	49
3.4.6	<i>Quick Sort.....</i>	51
3.4.7	<i>Heap Sort</i>	54
3.5	Lista de valores.....	56
4	RESULTADOS	65
4.1	Análise individual	68
4.1.1	<i>Bubble Sort.....</i>	68
4.1.2	<i>Shake Sort.....</i>	70
4.1.3	<i>Select Sort.....</i>	72
4.1.4	<i>Insert Sort.....</i>	73
4.1.5	<i>Merge Sort.....</i>	75
4.1.6	<i>Quick Sort.....</i>	77
4.1.7	<i>Heap Sort</i>	79
4.2	Análise conjunta	80
4.2.1	<i>Ordenação simples.....</i>	81

4.2.2 <i>Ordenação avançada</i>	86
4.3 Quando usar o quê	91
5 CONCLUSÃO	93
REFERÊNCIA BIBLIOGRÁFICA	97
6 FICHAS DE ATIVIDADES PRÁTICAS SUPERVISIONADAS	99

1 INTRODUÇÃO

Um computador é uma máquina que manipula informações. O estudo da ciência da computação inclui o exame da organização, manipulação e utilização destas informações num computador. Consequentemente, é muito importante compreendê-los.

O aspecto marcante da atual sociedade é a automatização de tarefas, e na ciência da computação há um processo de desenvolvimento simultâneo e interativo de máquinas e dos elementos lógicos que gerenciam a execução automática de uma tarefa.

Segundo Laureano (2008, p. 01), estruturas de dados são formas otimizadas de armazenamento e tratamento das informações eletronicamente.

Santos (2014, p. 242) define a estrutura de dados como um conjunto de dados que se encontram organizados sob determinada forma.

As estruturas de dados, na maioria dos casos, adaptam as informações vistas no dia a dia para serem armazenadas e manipuladas nas máquinas. Elas diferem uma das outras pela disposição ou manipulação de seus dados e cada uma possui uma vantagem e desvantagem. O tipo de estrutura influencia na escolha do algoritmo para a solução do problema.

Uma das tarefas mais comuns no dia a dia é a pesquisa de informação. Mas, a pesquisa depende muito da forma como a informação está organizada. Se a informação não estiver organizada não temos alternativa, infelizmente, que não seja percorrer todo o arranjo de informação por ordem seja ela do princípio para o fim ou vice-versa, até encontrar a desejada. Mas, se a informação estiver ordenada por uma ordem, seja ela crescente ou decrescente no caso de informação numérica, ascendente ou descendente no caso de informação textual, a pesquisa pela informação seria mais simplificada.

Laureano (2008, p. 02) evidencia que para distinguir dados de naturezas distintas e saber quais operações podem ser realizados com elas, os algoritmos lidam com o conceito de tipo de dados.

A ordenação é o processo de organizar um conjunto de informações segundo uma determinada ordem. Se o conjunto estiver ordenado é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo, a pesquisa binária. A

ordenação é uma tarefa muito importante no processamento de dados e é feita para facilitar a busca.

Laureano (2008, p. 100) relata que algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em certa ordem.

Devido ao seu uso muito frequente, é importante ter a disposição algoritmos de ordenação eficientes tanto em termos de tempo como em termos de espaço e processamento.

A partir destas considerações, visa-se responder a seguinte pergunta: Qual o desempenho dos algoritmos de ordenação em certos cenários de desordem?

Partindo-se da hipótese, que assimilando o tempo e o conteúdo de estudo com as informações obtidas através de pesquisas é possível, de forma estratégica, concretizar o desempenho de cada algoritmo em certos cenários, determinar os melhores e os piores em certos cenários e determinar as vantagens e desvantagens.

O presente trabalho busca contribuir para a sociedade científica mostrando gráficos e tabelas que relatam o desempenho de cada algoritmo e demonstrar uma análise eficiente que sirva de base para futuros desempenhos de ordenação.

Com pesquisas, avaliações e benefícios, o trabalho intenciona evidenciar o desempenho que as técnicas de ordenação apresentam destacando a melhor em cada cenário, resultando tudo isso em um *software* de ordenação de dados desenvolvido na linguagem Java.

O trabalho intenciona esclarecer conceitos, funções e aplicações das técnicas de ordenação aos leigos. Para tanto, apresentando as técnicas mais usadas e conhecidas e analisando o desempenho das mesmas em certos cenários de desorganização.

Fundamentado em obras publicadas por estudiosos especialistas, como Tenenbaum (1995), Lafore (2004), Goodrich e Tamassia (2007), Laureano (2008), Ascencio e Araújo (2010) e Santos, Júnior e Neto (2013), que já apontaram os conceitos, usos, aplicações, vulnerabilidades e falhas de certos algoritmos de ordenação, a presente pesquisa busca unificar e transmitir os principais conceitos de suas obras. A junção de treze obras, de ano e autores diferentes, resulta neste trabalho.

No primeiro capítulo reunimos conceitos gerais de estrutura de dados, tipos de dados, pesquisa e ordenação de dados. Nesta etapa consideramos as definições de dez autores dos que foram pesquisados, a saber, de Laureano (2008), Lafore (2004), Goodrich e Tamassia (2007), Ascencio e Araújo (2010), Santos, Júnior e Neto (2013) e Santos (2014).

No segundo capítulo é descrito, parcialmente, o processo de desenvolvimento do software. Nesta etapa consideramos as instruções de oito autores dos que foram pesquisados, a saber, Sierra e Bates (2005), Goodrich e Tamassia (2007), Horstmann (2009), Deitel e Deitel (2010) e Santos (2014).

O terceiro capítulo apresenta os gráficos, tabelas e descrição de cada algoritmo de ordenação abordado no presente trabalho. Abordará as vantagens e desvantagens de cada algoritmo, juntamente com o melhor e o pior de cada cenário. Os livros do autor Lafore (2004) e dos Ascencio e Araújo (2010) foram fundamentais e fortemente usados para a compreensão da lógica por trás dos principais algoritmos de ordenação.

2 ESTRUTURA DE DADOS

Um computador é uma máquina que manipula informações. O estudo da ciência da computação inclui o exame da organização, manipulação e utilização destas informações num computador. Consequentemente, é muito importante compreendê-los.

Se a ciência da computação é fundamentalmente o estudo da informação, a primeira pergunta que surge é: o que significa a informação? Infelizmente, embora o conceito de informação seja a espinha dorsal, não há existência de uma resposta exata para esta pergunta. O conceito de informação na ciência da computação é semelhante ao conceito de lógica nos estudos relacionados à área, como inteligência artificial, redes neurais, lógica de predicado, lógica clássica e etc., não pode ser definido, mas pode ser feita afirmação e explicado em termos de conceitos elementares.

O aspecto marcante da atual sociedade é a automatização de tarefas, e na ciência da computação há um processo de desenvolvimento simultâneo e interativo de máquinas e dos elementos lógicos que gerenciam a execução automática de uma tarefa.

Na atual sociedade, um fator de relevante importância é a forma de armazenar e manipular as informações. Então, partindo desde pressuposto, de nada adiantaria o grande desenvolvimento de software e máquina se a forma de armazenamento e manipulação não fosse equipolente ao desenvolvimento.

Segundo Laureano (2008, p. 01), estruturas de dados são formas otimizadas de armazenamento e tratamento das informações eletronicamente.

Uma estrutura de dados é um conjunto de dados que se encontram organizados sob determinada forma. Ela pode armazenar mais que um valor de determinado tipo de dados e define o modo como estes valores serão armazenados e manipulados. (Santos, 2014, p. 242).

As estruturas de dados, na maioria dos casos, adaptam as informações vistas no dia a dia para serem armazenadas e manipuladas nas máquinas. Elas diferem uma das outras pela disposição ou manipulação de seus dados e cada uma possui

uma vantagem e desvantagem. O tipo de estrutura influencia na escolha do algoritmo para a solução do problema.

Os dados manipulados por um algoritmo podem possuir naturezas distintas, isso é, podem ser letras, números, frases, valores lógicos, etc. As operações de um dado dependem do tipo de dado que será manipulado, porque algumas operações não fazem sentido quando aplicado a eles. Por exemplo, somar duas letras, mas existem algumas linguagens de programação que permitem que ocorra a soma dos valores ASCII.

Para poder distinguir dados de naturezas distintas e saber quais operações podem ser realizados com elas, os algoritmos lidam com o conceito de tipo de dados. O tipo de um dado define o conjunto de valores que uma variável pode assumir, bem como o conjunto de todas as operações que podem atuar sobre qualquer valor daquela variável. (LAUREANO, 2008, p. 02).

Os tipos de dados podem ser divididos em primitivos (atômicos) e estruturados (complexos ou compostos). Os atômicos são aqueles cujos elementos do conjunto são indivisíveis, a partir dos quais podemos definir os demais tipos e estruturas. Por outro lado, os tipos compostos são aqueles cujos elementos do conjunto são valores divisíveis, são construídos a partir dos primitivos e podem ser homogêneos e heterogêneos.

Dados homogêneos têm a capacidade de armazenar um conjunto de dados de mesma natureza. Os mais comuns são *array* e matrizes. Por exemplo, posso ter um *array* com capacidade de armazenar cinquenta números inteiros. Ela também é conhecida como “estática”, porque seu valor é fixado no momento em que ela é criada e, depois disso, este limite não pode mais ser modificado.

Dados heterogêneos têm a capacidade de armazenar um conjunto de dados de natureza distintos. São chamados de TADs (Tipos Abstratos de Dados). Por exemplo, podemos criar uma estrutura que além de armazenar letras, ela possa armazenar números e valores booleanos.

Segundo Pereira (1996), um tipo abstrato de dados é formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre esses valores. [...] (GOODRICH e TAMASSIA, 2007, p. 01).

2.1 Tipos de dados

Programas recebem dados, realizam processamentos com eles e retornam o resultado. A facilidade de execução é influenciada pela natureza do dado disponível para representar o mundo real.

Como supracitado há alguns tipos de dados e neste tópico estarei aprofundando um pouco mais no assunto.

2.1.1 *Tipos primitivos*

São atômicos, indivisíveis, não definidos com base em outros tipos de dados.

Associados a um nome, a um domínio simples e a um conjunto pré-definido de operação. Podem ser classificados como:

- Tipo primitivo numérico inteiro: pode armazenar apenas valores pertencentes ao conjunto dos inteiros;
- Tipo primitivo numérico ponto flutuante: pode armazenar valores pertencentes ao conjunto dos reais;
- Tipo primitivo booleano: pode armazenar apenas dois valores lógicos, verdadeiro (1) ou falso (0);
- Tipo primitivo caractere: pode armazenar apenas um caractere.
- Tipo ordinal: aquele cuja faixa de valores possíveis pode ser facilmente associada ao conjunto dos números naturais.

2.1.2 *Tipo de dados estruturais*

São tipos compostos a partir de outros tipos de dados, geralmente primitivo. Podem ser homogêneos e heterógenos.

- Tipo array: armazena um conjunto de dados de mesma natureza em um espaço linear fixo. É um tipo homogêneo;

- Tipo matriz: armazena um conjunto de dados de mesma natureza em um espaço bidimensional ou multidimensional linear fixo. É um tipo homogêneo;
- Tipo string caractere: é um array que armazena um conjunto de caractere. É um tipo homogêneo;
- Tipo registro: é um agregado possivelmente heterogêneo de elementos cujos elementos individuais são identificados por nomes;
- Tipo união: um tipo cujas variáveis podem armazenar tipos de dados diferentes em tempos diferentes da execução;

2.1.3 Tipo ponteiro

Um tipo ponteiro é aquele em que as variáveis têm uma faixa de valores que consiste em endereços de memória e um valor especial: *null*.

Null não é um endereço válido e serve para indicar que um ponteiro não pode ser usado atualmente para referenciar qualquer célula de memória.

2.1.4 Tipos abstratos de dados

Um tipo TAD é aquele em que seus valores são de naturezas distintas, ou seja, são tipos do qual o próprio programador cria para suprir as suas necessidades.

2.2 Pesquisa de dados

Uma das tarefas mais comuns no dia a dia é a pesquisa de informação. Mas, a pesquisa depende muito da forma como a informação está organizada. Se a informação não estiver organizada não temos alternativa, infelizmente, que não seja percorrer todo o arranjo de informação por ordem seja ela do princípio para o fim ou vice-versa, até encontrar a desejada. Este processo de pesquisa é normalmente lento, porque depende da quantidade de informação alocada. Imagine, por exemplo, um catálogo de telefone organizado na ordem de chegada, o último a solicitar a linha telefônica será o último no catálogo, para encontrar o telefone desejado, você teria que percorrer o catálogo inteiro para encontrar o nome desejada, porque os nomes estariam inseridos numa ordem aleatória. Como as entradas estão classificadas em

ordem cronologia, e não tem ordem alfabética, o processor de busca é sofisticado. Mas, se a informação estiver ordenada por uma ordem, seja ela crescente ou decrescente no caso de informação numérica, ascendente ou descendente no caso de informação textual, a pesquisa pela informação seria mais simplificada.

Tal como no dia a dia, a pesquisa de informação é também uma tarefa trivial em programação. Pesquisar um agregado à procura da localização de um determinado valor, ou de uma determinada característica acerca dos seus valores, é uma tarefa muito frequente e simples. Mas, é computacionalmente dispendiosa, porque o agregado pode ser constituído por centenas ou mesmo milhares de elementos. Pelo que exige o desenvolvimento de algoritmos eficientes.

A maneira mais simples de pesquisar um agregado é a pesquisa sequencial, também chama de pesquisa linear, pode ser executado em informação ordenada e não ordenada. Consiste em analisar todos os elementos agregados de maneira metódica. A pesquisa ser inicia no primeiro elemento do agregado e avança elemento a elemento até encontrar o valor procurado, ou até atingir o elemento final do agregado. Este método de pesquisa é normalmente demorado e pouco produtivo, porque é dependente do tamanho do agregado e de sua organização.

Em um vetor não ordenado, será buscado o número até que ele seja encontrado ou até se chegar ao final do vetor. Em um vetor ordenado, será buscado o número até que ele seja encontrado e enquanto for maior que o número do vetor. (ASCENCIO e ARAÚJO, 2010, p. 106).

No entanto, se os elementos estiverem ordenados por ordem crescente ou decrescente, é então possível fazer uma pesquisa binária, também chamada de pesquisa logarítmica. Esse tipo de pesquisa começa por selecionar o elemento central do agregado e o compara com o valor procurado. Se estes forem iguais, a busca termina, caso contrário, se o número procurado for maior que o do meio então podemos excluir a primeira metade do agregado e analisamos apenas a segunda metade. Caso contrário, podemos excluir a segunda metade do agregado e trabalha apenas com a primeira. O processo é repetido até que o número de elemento a ser analisado seja reduzido à zero, o que significa, que o valor procurado não existe no conjunto, ou até que seja encontrado o valor desejado.

Ascencio e Araújo (2010, p. 114) alertam que a busca binária é executada somente em dados ordenados.

A compensação por usar um vetor ordenado vem quando usamos uma pesquisa binária. Esse tipo de pesquisa é muito mais rápido do que uma pesquisa linear, especialmente para grandes vetores. (LAFORE, 2004, p. 42).

2.3 Ordenação de dados

A ordenação é o processo de organizar um conjunto de informações segundo uma determinada ordem. Se o conjunto estiver ordenado é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo, a pesquisa binária. A ordenação é uma tarefa muito importante no processamento de dados e é feita para facilitar a pesquisa.

Na verdade, ordenação de dados é o processo pelo qual é determinada a ordem na qual devem se apresentar as entradas de uma tabela de modo que obedeçam à sequência citada por um ou mais de seus campos. Estes campos especificados como determinantes da ordem são chamados chaves de ordenação. (SANTOS, JÚNIOR E NETO, 2013, p. 136).

Em diversas aplicações, os dados devem ser armazenados obedecendo a uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista computacional. Para obtermos os dados ordenados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados respeitando a ordenação, ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo de ordenação.

Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em certa ordem - em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica. (LAUREANO, 2008, p. 100).

Devido ao seu uso muito frequente, é importante ter a disposição algoritmos de ordenação eficientes tanto em termos de tempo como em termos de espaço.

3 DESENVOLVIMENTO

No desenvolvimento do sistema computacional foi utilizado um laptop da fabricante *Samsung* com processador *Intel® Core™ i5 – 3230M CPU @ 2.60 GHz*, RAM de 8.00 GB, sistema operacional *Windows 10 Pro x64* e HD de 1.00 TB. Os principais softwares utilizados no processo foi a IDE *Eclipse Java Mars x64* com acesso ao *JDK 1.8.0_92 x64* e *JRE 1.8.0_92 x64*, últimas versões até o presente momento de dissertação, *Sublime Text 3*, *Visual Studio Code*, *Notepad* e o *Prompt de Comando*.

Os principais livros utilizados foram “Programação de Computadores em Java” de Rui Rossi dos Santos e “Estruturas de Dados & Algoritmos em Java™” de Robert Lafore, ambos ser encontram na referência bibliográfica.

A linguagem utilizada foi o Java 8 (JSE 8) destinada ao desktop. Vale ressaltar que mesmo com a disponibilidade das bibliotecas da linguagem, que é uma de suas vantagens, não foram utilizadas nenhuma função de ordenação inata da mesma. Todas as funções presentes no trabalho foram reescritas pelos seus respectivos desenvolvedores.

3.1 Interface Gráfica (GUI)

No desenvolvimento da interface gráfica foi utilizado o *plug-in*, da IDE Eclipse, *Window Builder*. Este *plug-in* facilita no manuseio das APIs AWT e *Swing*, dando a possibilidade de usufruir de suas capacidades criativas de desenvolvimento de GUI.

No Java, o desenvolvimento de interface gráfica pode ser realizado com o uso de componentes disponíveis em duas APIs distintas: o AWT e o *Swing*. Uma GUI é desenvolvida utilizando-se componentes, que são objetos visuais que possibilitam ao usuário realizar a interação com o programa por meio de um ambiente gráfico. (SANTOS, 2014, p. 970).

A grande vantagem do *Window Builder* é fornecer a possibilidade de “clicar e arrastar” os componentes desejados para a janela, chamada de *JFrame*. Alguns componentes comuns são os rótulos, os botões, os campos de texto, as áreas de texto, as caixas de checagem, os botões de rádio, as listas e os menus. Nomeados,

em Java, de *JLabel*, *JButton*, *TextField*, *TextArea*, *JCheckBox*, *JRadioButton*, *JList* e *JMenuBar*, respectivamente.

3.1.1 Janelas (*JFrame*)

Um programa pode interagir com o usuário de duas formas distintas: através de uma interface gráfica (GUI) ou através do modo textual (CLI). Enquanto a interface gráfica conta com janelas e componentes visuais, o modo textual suporta apenas a entrada e saída de caracteres. A interatividade provida do modo textual é bastante precária, não sendo possível, por exemplo, o uso do mouse. Dificilmente você desejará desenvolver um programa que utilize o modo textual para interagir com o usuário.

Um programa desenvolvido em CLI não será facilmente aceito, nos dias atuais. A interação do usuário com o software é tão importante quanto o desempenho do mesmo. Este é um assunto estudado na área de IHC, onde a mesma foca na reação de primeiro contato entre computador e usuário.

Uma janela é a base fundamental da organização das interfaces gráficas, que abstraem a noção de um compartimento dentro do qual determinado programa organiza componentes visuais para interagir com o usuário. Um programa pode conter uma única janela ou abrir diversas delas ao mesmo tempo, através das quais seja possível realizar diferentes operações.

A classe *javax.swing.JFrame*, segundo Santos (2014, p. 978), é uma versão estendida da classe *java.awt.Frame* que adiciona suporte à arquitetura dos componentes do Swing. Entretanto ela é incompatível com a classe *java.awt.Frame*.

Para a criação de uma janela é necessário a realização da instanciação da classe *JFrame*, mas é recomendado a criação de uma classe filha que tenha como pai a classe *JFrame*. Para adicionar um componente a uma janela do tipo *JFrame* é preciso invocar o método *add()* do objeto que representa seu painel de conteúdo. Segue o trecho do código para a criação de uma janela e adição de um componente.

```
JFrame frame = new JFrame();  
frame.getContentPane().add(<nome_do_componente>);
```

O mesmo vale para a remoção de componentes da janela e outros métodos relacionados ao gerenciamento destes componentes. Todos esses métodos devem ser invocados a partir do painel de conteúdo ao invés de serem invocados a partir da própria janela.

3.1.2 Rótulos (*JLabel*)

Rótulo é uma área para a exibição de um texto curto, uma imagem ou ambos. Ele não reage a eventos de entrada e, por isso, não pode receber o foco do teclado.

Os rótulos contêm instruções de texto ou informações sobre outro componente e são representados pela classe *javax.swing.JLabel*, que é derivada da *javax.swing.JComponent*. Um rótulo exibe uma única linha de texto e uma vez criado raramente tem seu conteúdo alterado. Segue o trecho do código para a criação e adição de um rótulo, em uma *JFrame*.

```
JLabel rotulo = new JLabel(<conteudo_do_rotulo>);  
frame.getContentPane().add(rotulo);
```

É possível especificar onde o conteúdo será alinhado na área de um rótulo e também é permitido configurar a posição do texto em relação à imagem, quando os dois estiverem presentes.

3.1.3 Botões (*JButton*)

Um botão é um componente que pode ser pressionado pelo usuário, utilizando o mouse ou o teclado, para acionar uma ação específica. O swing oferece uma classe que representa esse tipo de componente, a classe *javax.swing.JButton*. Ela deriva de uma classe abstrata, chamada *javax.swing.AbstractButton*. As classes que representam botões de rádio e caixas de checagem também derivam dessa mesma classe.

```
JButton botao = new JButton("Título do botão");  
frame.getContentPane().add(botao);
```

O trecho do código supracitado instancia um novo botão e adiciona o mesmo à janela.

3.1.4 Painéis (*JPanel*)

Os painéis agem como contêineres que servem para dividir a janela. Eles são utilizados para possibilitar maior controle sobre a organização dos componentes que compõem interfaces gráficas mais complexas e, em conjunto com os gerenciadores de layout, permitem mais exatidão no posicionamento dos mesmos.

```
JPanel contentPane = new JPanel();
contentPane.<metodo>(<conteudo_do_panel>);
```

O trecho do código supracitado instancia um objeto *JPanel* da classe *javax.swing.JPanel* que representa um contêiner de “peso leve” genérico. Ela deriva da classe *javax.swing.JComponent*.

3.1.5 Eventos de Ação (*Action Event*)

Eventos de ação são eventos de alto nível gerados por um componente quando a ação especificada por ele ocorre. Um evento de ação típico é o pressionamento de um botão. Quando um evento desse tipo ocorre, o componente gerador do evento cria um objeto da classe *java.awt.ActionEvent*, nele armazena todas as informações acerca do evento ocorrido e o remete para cada ouvinte que tenha sido registrado através do método *addActionListener()*.

```
<Objeto JButton>.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0){
    }
});
```

O trecho do código supracitado mostra como é ativado um evento de ação. Primeiro ativamos o método *addActionListener()* de algum componente, no código

mostra um *JButton*, e passamos como parâmetro um construtor da classe *ActionListener* que tenha o método *actionPerformed(ActionEvent arg0)* no escopo.

3.1.6 Campos de Texto (*TextField*)

A classe *javax.swing.JTextField* representa um componente de “peso leve” que permite a edição de uma única linha de texto.

É possível indicar o número de colunas para um *TexteField* utilizando seu método *setColumns()*. Uma coluna é o espaço necessário para abrigar um caractere na fonte que está sendo usada para o texto e o total de colunas deveria representar exatamente a largura necessária para o componente abrigar o texto esperado.

[...] infelizmente uma coluna é um dado bastante impreciso e serve tão somente para indicar o tamanho preferencial do componente. Além disso, o número de colunas não representa o limite superior de caracteres que podem ser digitados pelo usuário. (SANTOS, 2014, p. 1066).

O trecho de código, abaixo, mostra a criação de um campo textual, sua posição no *JPanel* e seu número de coluna.

```
JTextField text = new JTextField();  
text.setBounds(x, y, x, y);  
text.setColumns(10);
```

3.1.7 Barras de Rolagem (*ScrollPane*)

A classe *javax.swing.JScrollPane* representa uma barra de rolagem comum. O usuário desloca o puxador da barra de rolagem para exibir o conteúdo de um componente em sua área de visualização e o programa ajusta a visualização conforme a barra de rolagem é pressionada.

O código abaixo mostra a instanciação da classe *JScrollPane*, aciona o método que trata de sua localização no *JPanel*, adiciona o componente, geralmente uma *JList* ou *JTextArea*, ao método *setViewportView()* e finaliza adicionando a rolagem no *contentPane*.

```

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(x, y, x, y);
scrollPane.setViewportView(<componente>);
contentPane.add(scrollPane);

```

3.1.8 Listas (JList)

As listas são componente que permite ao usuário escolher uma ou mais opções predefinidas. A classe *javax.swing.JList* é utilizada para representar a aparência de uma lista e a classe *javax.swing.ListModel* representa o seu conteúdo.

O trecho do código mostra a instanciação de uma *JList* e sua utilização.

```

JList list = new JList(<itens>);
scrollPane.setViewportView(list);

```

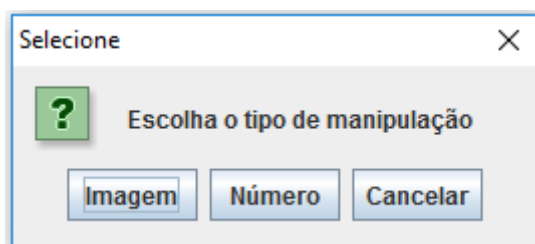
Inicialmente instanciamos a classe *JList* e atribuímos no seu construtor os itens que podem ser *DefaultListModel*, ou números, ou *string*, ou vetores e etc. O objeto do tipo *JList* é adicionado ao painel de rolagem, dando a possibilidade da lista ter um tamanho *n* e uma visualização agradável.

3.2 Obtenção de Dados

O sistema computacional é capaz de obter e ordenar imagens e números. Foram usadas setes classes padrões do Java para a obtenção e armazenamento de arquivos e números.

A imagem abaixo ilustra a opção de manipulação de dado.

Imagem 01 – Opção de manipulação.



Fonte: Própria, 2016.

3.2.1 Arquivos

A classe *javax.swing.JFileChooser* é responsável por solicitar e receber um arquivo do usuário, com a utilização de janela gráfica. Possui alguns métodos necessários para a sua prestação.

A classe *javax.swing.filechooser.FileNameExtensionFilter* é responsável por especificar o tipo, extensão, de arquivo esperado.

Os métodos e constantes usados para a solicitação e armazenamento de arquivos foram:

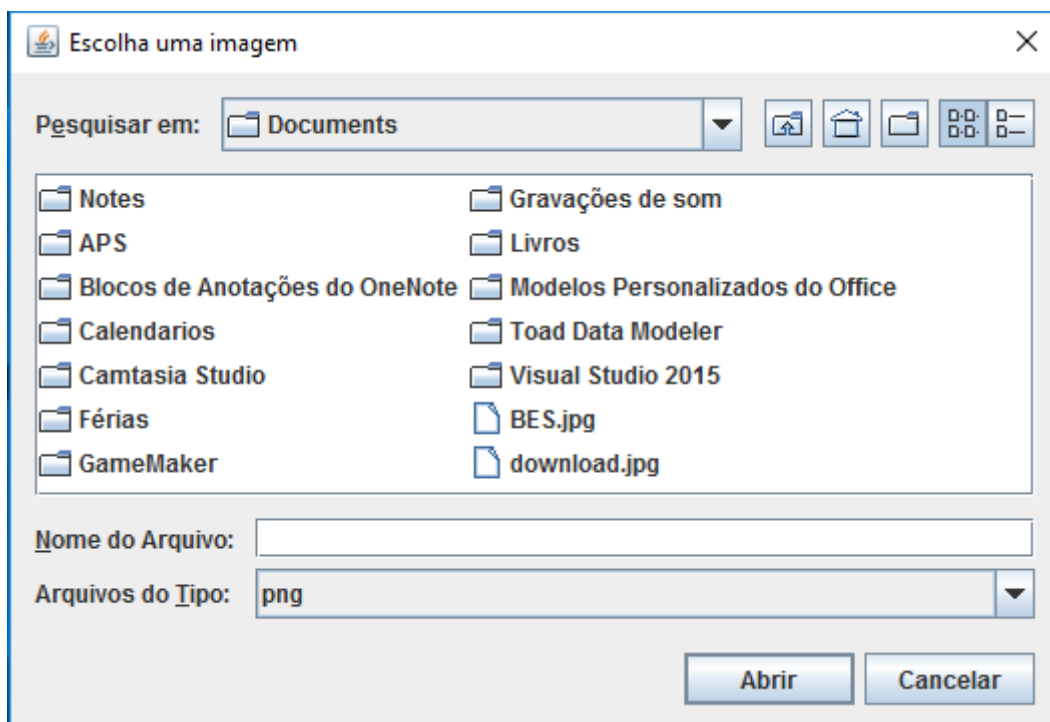
- **setFileFilter()**: responsável pela inserção de tipos, específicos, de arquivo.
- **setDialogTitle()**: responsável pela edição de título da janela.
- **getSelectedFile()**: responsável pela captação do diretório do arquivo.
- **getAbsolutePath()**: responsável pela captação absoluta do diretório.
- **JFileChooser.CANCEL_OPTION**: uma constante estática que tem o valor do cancelamento da janela.
- **JFileChooser.APPROVE_OPTION**: uma constante estática que tem o valor de aprovação da janela.

A imagem dois mostra o resultado da intersecção dos métodos supracitado acima.

Após a obtenção do diretório da imagem. O usuário deve digitar o nome da mesma, para que o programa possa listar ela com o nome exigido pelo mesmo.

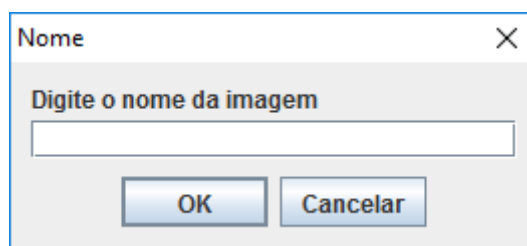
A imagem três mostra a solicitação do nome que irá referenciar o arquivo.

Imagem 02 – Obtenção de imagem.



Fonte: Própria, 2016.

Imagem 03 – Solicitação de nome.



Fonte: Própria, 2016.

A *javax.swing.JOptionPane* é a classe responsável pela geração da caixa de inserção. O método que realiza esta manipulação é o *showInputDialog()*.

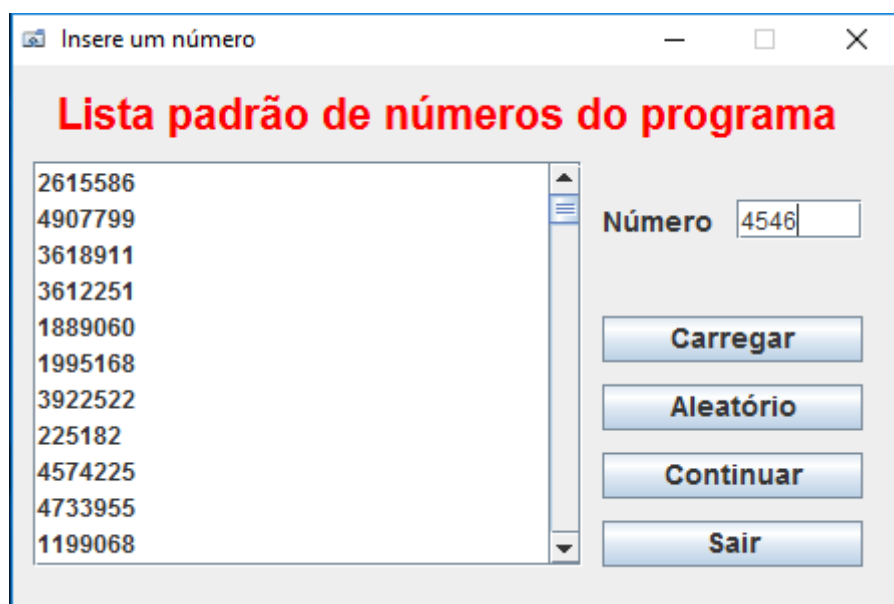
3.2.2 Números

Além de imagens, o programa também pode receber e ordenar números.

Ao contrário da imagem, os números são inseridos no mesmo *JFrame* que a listagem padrão de números.

A imagem abaixo mostra a solicitação de números, dando a possibilidade de o usuário digitar ou de gerar um número aleatório.

Imagem 04 – Obtenção de número.



Fonte: Própria, 2016.

Para receber um número do usuário foi desenvolvido um método que ao clicar no botão “carregar” é capturado o que é inserido no *JTextField* e lançado para o método responsável pela inserção do dado no vetor. Estaremos explicando apenas o método de captura, porque os demais componentes já foram explicados no subcapítulo anterior.

O escopo do método responsável pela captação é mostrado na imagem abaixo.

Imagem 05 – Método *getTxtNumber*.

```
private JTextField getTxtNumber()
{
    if(textFieldNumero == null)
    {
        textFieldNumero = new JTextField();
        textFieldNumero.setBounds(361, 67, 63, 20);
    }

    return textFieldNumero;
}
```

Fonte: Própria, 2016.

O método tem o encapsulamento *private* com o retorno do tipo *JTextField* e é nomeado como *getTxtNumber*. O *if* verifica se o campo está vazio, se estiver cria um novo campo no lugar do antigo e retorna o mesmo para quem o chamou.

O escopo do método responsável pela ação de armazenamento é mostrado na imagem 06.

Imagem 06 – Método de ação.

```
btnCarregar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            Inserir.inserir(Integer.parseInt(getTxtNumber().getText()));
            getTxtNumber().setText("");
        }

        catch (NumberFormatException nbe)
        {
            JOptionPane.showMessageDialog(null, "Digite apenas números",
                "NumberFormatException", JOptionPane.ERROR_MESSAGE);
            getTxtNumber().setText("");
        }
    }
});
```

Fonte: Própria, 2016.

O método usufrui do tratamento de exceção, *try – catch*, responsável pelo gerenciamento de erros ocorridos de segundos. Primeiro ele chama o método *inserir()* da classe *Inserir* e passa como parâmetro o retorno do método *getTxtNumber().getText()*, onde o mesmo é convertido para um tipo inteiro.

Após a obtenção do dado inserido no campo *JTextField*, ele insere no campo um texto vazio.

3.3 Geração de Dados

A geração de dados é ativada logo após a seleção de manipulação de dados. É chamada a classe responsável pela geração randômica dos números e a inserção das imagens que se encontram na pasta raiz do programa.

3.3.1 Arquivos

A classe *ImagemPadrao* é responsável pela inserção das imagens.

Segue uma imagem que ilustra o escopo da classe ImagemPadrao.

Imagem 07 – Classe ImagemPadrao.

```
public class ImagemPadrao
{
    private static Imagem imagem;

    public static void insere()
    {
        for(int i = 1; i <= 480; i++)
        {
            imagem = new Imagem("res\\apsJAVA (" + i + ").jpg");

            if(i > 0 && i < 10)
                Insere.insere(imagem, "APS JAVA (0000" + i + ")");

            else
                if(i > 9 && i < 100)
                    Insere.insere(imagem, "APS JAVA (000" + i + ")");

            else
                Insere.insere(imagem, "APS JAVA (00" + i + ")");

        }
    }
}
```

Fonte: Própria, 2016.

A classe possui um atributo encapsulado e estático e um método com o modificador *static*, onde o escopo do mesmo possui um laço de repetição responsável por armazenar e nomear quatrocentos e oito imagens do tipo JPG.

O atributo é do tipo Imagem e possui alguns métodos, como o insere() e seu construtor.

O objeto é construindo com o diretório como parâmetro do seu construtor. Após a construção do objeto é inserido o nome da imagem com base nos laços de condições, para que o nome fique padronizado.

Imagem 08 – Lista de imagem padrão.

APS JAVA (00010)	626	626
APS JAVA (00011)	400	300
APS JAVA (00012)	413	600
APS JAVA (00013)	1265	550
APS JAVA (00014)	1920	1080
APS JAVA (00015)	2435	1218
APS JAVA (00016)	1200	1200
APS JAVA (00017)	3405	2363

Fonte: Própria, 2016.

3.3.2 Números

A classe NumeroPadrao é responsável pela geração randômica dos números e é instanciada caso o usuário selecione a opção de manipulação numérica.

Segue uma imagem que ilustra o escopo da classe NumeroPadrao.

Imagem 09 – Classe NumeroPadrao.

```
public class NumeroPadrao
{
    public static void insere()
    {
        for(int i = 0; i < 5000000; i++)
            Insere.insere((int)Math.ceil(Math.random()*5000000));
    }
}
```

Fonte: Própria, 2016.

A classe possui apenas um método com o modificador *static*, onde o escopo do mesmo possui um laço de repetição que gera e insere cinco milhões de números aleatórios com intervalo de um a cinco milhões.

O número é convertido para o tipo inteiro antes de sua inserção.

A classe *Math* é responsável pela geração dos números randômico e o arredondamentos dos mesmos. O método *random()* retorna um dado *double*, conseqüentemente é necessário a utilização do método *ceil()* para arredondar uma casa à cima.

3.4 Algoritmos de ordenação

Algoritmo de ordenação coloca os elementos de uma dada sequência em uma certa ordem, efetuando sua ordenação completa ou parcial. O critério de ordenação fica a cargo do analista.

No trabalho foi utilizado sete algoritmos de ordenação, quatro simples e três avançados. O programa possui oito algoritmos, mas só foram considerados sete dos oito.

Bubble Sort, *Shake Sort*, *Select Sort* e *Insert Sort* são os algoritmos simples. *Merge Sort*, *Quick Sort* e *Heap Sort* são os algoritmos avançados. O oitavo algoritmo também é avançado e é conhecido como *Shell Sort*.

3.4.1 *Bubble Sort*

O algoritmo *Bubble Sort*, ou ordenação por flutuação, é notavelmente lenta, mas é ela conceitualmente o mais simples dos algoritmos de ordenação. Pertence à classe de métodos baseados em permutação.

A ideia é percorrer o vetor n vezes, a cada passagem realizar uma comparação entre n e $n + 1$ e realizar a troca caso o elemento do n seja maior que o seu predecessor.

O algoritmo possui dois laços *for* encadeado. O primeiro *for*($out = n - 1$; $out > 1$; $out--$) percorre o vetor da direita para a esquerda e dentro dele tem o *for*($in = 0$; $in < out$; $in++$) responsável por percorrer o vetor da esquerda para direita. A condição que verifica se o elemento da esquerda é maior que o da direita é *if*($vetor[in] > vetor[in + 1]$).

A imagem a seguir mostra o método de ordenação do *bubble*.

Imagem 10 – Método *bubble*.

```

public static void bubbleSortInt()
{
    for(out = getnElemsInt() - 1; out > 1; out--)
    {
        for(in = 0; in < out; in++)
        {
            if(getArrayInt(in) > getArrayInt(in + 1))
                swapInt(in, in + 1);

            ++count;
        }

        ++count;
    }

    System.out.println(count);
    count = 0;
}

```

Fonte: Própria, 2016.

3.4.2 Shake Sort

O algoritmo *Shake Sort* é um aperfeiçoamento do *Bubble Sort*.

Ao final de uma varredura da esquerda para a direita, é efetuada outra da direita para a esquerda para que o menor valor também se desloque para a sua posição definitiva.

O algoritmo possui dois laços *for* dentro de um laço *do while*. O laço *do while* será executado enquanto o contador for menor ou igual à quantidade *n* de dados, *do { } while(out <= n)*. O primeiro laço *for(in = n; in >= count; in--)* vai da esquerda para a direita verificando se o elemento da esquerda é maior que o da direita, *if(vetor[in - 1] > getArrayInt[in])*. Antes de entrar no segundo laço, *out* recebe o valor de *in + 1*. No segundo laço *for(in = out; in <= r; in++)* vai da direita para a esquerda verificando se o elemento da esquerda é maior que o da direita, *if(vetor[in - 1] > getArrayInt[in])*.

A imagem a seguir mostra o método de ordenação do *shake*.

Imagem 11 – Método *shake*.

```

public static void shakeSortInt()
{
    int k = 0, r = 0;
    out = 1;
    r = k = getnElemsInt() - 1;
    do
    {
        for(in = r; in >= out; in--)
        {
            if(getArrayInt(in - 1) > getArrayInt(in))
            {
                swapInt(in - 1, in);
                k = in;
            }
            ++count;
        }
        out = k + 1;
        for(in = out; in <= r; in++)
        {
            if(getArrayInt(in - 1) > getArrayInt(in))
            {
                swapInt(in - 1, in);
                k = in;
            }
            ++count;
        }
        r = k - 1;
        ++count;
    } while(out <= r);
}

```

Fonte: Própria, 2016.

3.4.3 Select Sort

O algoritmo *Select Sort*, ou seleção direta, se baseia em passar o menor valor do vetor para a primeira posição (ou o maior dependendo do critério de ordenação), depois o segundo menor valor para a segunda posição e assim sucessivamente até $n - 1$. Pertence à classe de métodos baseados em seleção.

O algoritmo possui dois laços *for* encadeado. O primeiro *for*(*out* = 0; *out* < *n*; *out*++) percorre o vetor da esquerda para a direita e armazena o elemento da posição *out* em uma variável temporária, considerando ele como o menor do conjunto. Em seguida entra no laço *for*(*in* = *out* + 1; *in* < *n*; *in*++) e verifica se o elemento armazenado na variável temporária é maior que o elemento na posição *in*, se sim armazena o valor da posição *in* na variável temporária.

A imagem a seguir mostra o método de ordenação do *select*.

Imagem 12 – Método *select*.

```

public static void selectSortInt()
{
    for(out = 0; out < getnElemsInt(); out++)
    {
        min = out;

        for(in = out + 1; in < getnElemsInt(); in++)
        {
            if(getArrayInt(min) > getArrayInt(in))
                min = in;

            ++count;
        }

        swapInt(out, min);
        ++count;
    }

    System.out.println(count);
    count = 0;
}

```

Fonte: Própria, 2016.

3.4.4 Insert Sort

O algoritmo *Insert Sort*, ou ordenação por inserção direta, considera o primeiro elemento ordenado. O segundo elemento é inserido na sua posição correta em relação ao primeiro, resultando as duas primeiras posições ordenadas. A seguir, o terceiro elemento é inserido na sua posição correta em relação aos dois primeiros, resultando nas três primeiras posições ordenadas. E assim sucessivamente. Ao inserir cada novo elemento, deslocamentos são feitos, se necessários. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.

O algoritmo possui um laço *while* dentro de um laço *for*. Primeiro o laço *for*(*out* = 1; *out* < *n*; *out*++) percorre o vetor da esquerda para a direita e armazena o elemento da posição *out* em uma variável temporária. Em seguida entra no laço *while*(*in* > 0 && *vetor*[*in* - 1] >= *temp*) e realiza uma troca entre o elemento da posição *in* com o elemento antecessor.

A imagem a seguir mostra o método de ordenação do *insert*.

Imagem 13 – Método *insert*.

```

public static void insertSortInt()
{
    for(out = 1; out < getnElemsInt(); out++)
    {
        int temp = getArrayInt(out);
        in = out;

        while(in > 0 && getArrayInt(in - 1) >= temp)
        {
            setArrayInt(getArrayInt(in-1), in);
            --in;
            ++count;
        }

        setArrayInt(temp, in);
        ++count;
    }

    System.out.println(count);
    count = 0;
}

```

Fonte: Própria, 2016.

3.4.5 Merge Sort

O algoritmo *Merge Sort* é um método de ordenação recursivo, que por sua vez se baseia na técnica dividir para conquistar.

A ideia é dividir um vetor ao meio, ordenar cada metade e então intercalar as duas metades em um único vetor ordenado. É necessário a criação de um terceiro vetor, onde o mesmo será o vetor ordenado.

Os argumentos para o método *recMergeSortInt()* determinam as extremidades esquerda e direita do vetor que será ordenado e o vetor que servirá para armazenar os dados ordenados. Primeiro, o método verifica se esse vetor consiste em apenas um elemento. Se consisti, o vetor já está ordenado por definição e o método retornará imediatamente.

Se o vetor tiver duas ou mais células, o algoritmo chamará a si mesmo para continuar dividindo o vetor e o subvetor por dois. Após o vetor está dividido em duplas, é chamado o método *mergeInt()* e passado o vetor que irá armazenar os dados ordenados, as extremidades do vetor e o menor valor da dupla.

O método *mergeInt()* é responsável por comparar os valores de cada dupla e ordená-los dentro do terceiro vetor que servirá para armazenar os dados ordenados.

As imagens a seguir mostram o método de ordenação do *merge*.

Imagem 14 – Método *recMergeSortInt*.

```

public static void recMergeSortInt(int[] workSpace, int lowerBound, int upperBound)
{
    ++count;

    if(lowerBound == upperBound)
        return ;

    else
    {
        int mid = (lowerBound + upperBound) / 2;

        recMergeSortInt(workSpace, lowerBound, mid);

        recMergeSortInt(workSpace, mid + 1 , upperBound);

        mergeInt(workSpace, lowerBound, mid + 1, upperBound);
    }
}

```

Fonte: Própria, 2016.

Imagem 15 – Método *mergeInt*.

```

public static void mergeInt(int[] workSpace, int lowPtr, int highPtr, int upperBound)
{
    int j = 0;
    int lowerBound = lowPtr;
    int mid = highPtr - 1;
    int n = upperBound - lowerBound + 1;

    while(lowPtr <= mid && highPtr <= upperBound)
    {
        if(getArrayInt(lowPtr) < getArrayInt(highPtr))
            workSpace[j++] = getArrayInt(lowPtr++);

        else
            workSpace[j++] = getArrayInt(highPtr++);

        ++count;
    }

    while(lowPtr <= mid)
    {
        workSpace[j++] = getArrayInt(lowPtr++);
        ++count;
    }

    while(highPtr <= upperBound)
    {
        workSpace[j++] = getArrayInt(highPtr++);
        ++count;
    }

    for(j = 0; j < n; j++)
    {
        setArrayInt(workSpace[j], lowerBound+j);
        ++count;
    }
}

```

Fonte: Própria, 2016.

3.4.6 Quick Sort

O algoritmo *Quick Sort* é indubitavelmente, segundo Lafore (2004, p. 298), o algoritmo de ordenação mais popular. É recursivo e pertence à classe de métodos baseados em dividir e conquistar, assim como o *Merge Sort*. *Quick Sort* foi descoberto por C.A.R Hoare em 1962.

A ideia é dividir o vetor ao meio, por meio de um procedimento recursivo, baseado em um pivô. No lado direito do pivô ficam todos os elementos maiores e do lado esquerdo os menores. Então ordenarmos o subvetor à esquerda e o subvetor à direita para que o vetor inteiro esteja ordenado.

A diferença entre o *Quick Sort* e o *Merge Sort* é que o *Quick* ordena os dois lados sem dividi-los em dupla.

Os argumentos para o método *recQuickSortInt()* determinam as extremidades esquerda e direita do vetor (ou sub vetor) que será ordenado. Primeiro, o método verifica se esse vetor consiste em apenas um elemento. Se consistir, o vetor já estará ordenado por definição e o método retornará imediatamente.

Se o vetor tiver duas ou mais células, o algoritmo chamará o método *partitionInt()* para particioná-lo. Esse método retornará o número de índice da partição. A partição marca o limite entre os subvetores.

Depois do vetor ser particionado, *recQuickSortInt()* chama a si mesmo recursivamente, uma vez para a parte esquerda de seu vetor, de *left* para *partition-1*, e outra vez para a direita, de *partition+1* para *right*. O item de dados no índice *partition* não está incluindo em nenhuma chamada recursiva.

As imagens a seguir mostram o método de ordenação do *quick*.

Imagem 16 – Método *recQuickSortInt* e *medianOf3Int*.

```

public static void recQuickSortInt(int left, int right)
{
    int size = right - left + 1;

    ++count;

    if(size < 10)    //ordenação por inserção se pequeno
        insertionSortInt(left, 10);

    else //quick se grande
    {
        int median = medianOf3Int(left, right);
        int partition = partitionItInt(left, right, median);

        recQuickSortInt(left, partition-1);
        recQuickSortInt(partition+1, right);
    }
}

public static int medianOf3Int(int left, int right)
{
    int center = (left + right)/2;

    if(getArrayInt(left) > getArrayInt(center))
        swapInt(left, center);

    if(getArrayInt(left) > getArrayInt(right))
        swapInt(left, right);

    if(getArrayInt(center) > getArrayInt(right))
        swapInt(center, right);

    swapInt(center, right-1);    //coloca pivo a direita
}

```

Fonte: Própria, 2016.

Imagem 17 – Método *partitionItInt* e *insertionSortInt*.

```

        return getArrayInt(right - 1); //retorna valor medio
    }

    public static int partitionItInt(int left, int right, int pivot)
    {
        int leftPtr = left;
        int rightPtr = right - 1;

        while(true)
        {
            while(getArrayInt(++leftPtr) < pivot)
                ++count; //encontra maior

            while(getArrayInt(--rightPtr) > pivot)
                ++count; //encontra menor

            if(leftPtr >= rightPtr) //se ponteiros cruzam
                break; //particao feita

            else
                swapInt(leftPtr, rightPtr); //troca elementos

            ++count;
        }
        swapInt(leftPtr, right-1); //retorna pivo

        return leftPtr; //retorna posicao do pivo
    }

    public static void insertionSortInt(int left, int right)
    {
        int in, out;

        for(out = left = 1; out <= right; out++)

```

Fonte: Própria, 2016.

Imagem 18 – Método *insertionSortInt*.

```

    public static void insertionSortInt(int left, int right)
    {
        int in, out;

        for(out = left = 1; out <= right; out++)
        {
            int temp = getArrayInt(out);
            in = out;

            while(in >= left && getArrayInt(in - 1) >= temp)
            {
                setArrayInt(getArrayInt(in - 1), in);
                --in;
                ++count;
            }

            setArrayInt(temp, in);
            ++count;
        }
    }

```

Fonte: Própria, 2016.

3.4.7 *Heap Sort*

O algoritmo *Heap Sort* é, segundo Lafore (2004, p. 530), uma árvore binária. Ele fica completamente preenchido, lendo da esquerda para a direita cada linha, embora a última linha não precise estar cheia. Cada nó satisfaz a condição que estabelece que a chave de todo nó é maior (ou igual) que as chaves de seus filhos. Possui o mesmo princípio de funcionamento da ordenação por seleção:

1. Selecione o maior item do vetor;
2. Troque-o pelo item da primeira posição;
3. Repita a operação com os elementos restantes do vetor.

A ideia é dividir o vetor ao meio, assim como no *Quick* e *Merge*, mas isso não quer dizer que ele seja recursivo, e utilizar uma fila de prioridade junto com a abstração de árvore binária. Após a divisão é realizado uma verificação que determinar se o vetor possui apenas um elemento ou não. Se o elemento possuir um elemento o vetor já estará ordenado por definição e o método retornará imediatamente. Caso contrário entrará em laços de repetição que chamaram o método responsável por realizar a peneira, a mesma que verifica e aloca os valores para sua posição correta dentro da “árvore”.

O procedimento *heapSortInt()* realiza a verificação das extremidades e é responsável pelo laço que chama o método *peneiraInt()* passando a posição central e decrementando a final antes de passa-la.

O procedimento *peneiraInt()* recebe a posição central da “árvore” e a posição final com o decremento realizado pelo laço de repetição dentro do *HeapSortInt()*.

```

public static void heapSortInt()
{
    int p;

    out = getnElemsInt()-1;

    for(p = out / 2; p >= 1; --p)
    {
        peneiraInt(p, out);
        ++count;
    }

    for(in = out; in >= 2; --in)
    {
        swapInt(1, in);
        peneiraInt(1, in - 1);
        ++count;
    }

    System.out.println(count);
    count = 0;
}

```

Fonte: Própria, 2016.

Imagem 20 – Método *peneiraInt*.

```

public static void peneiraInt(int p, int m)
{
    int x = getArrayInt(p);

    while(2 * p <= m)
    {
        int f = 2 * p;
        if(f < m && getArrayInt(f) < getArrayInt(f + 1))
            ++f;

        if(x >= getArrayInt(f))
            break;

        setArrayInt(getArrayInt(f), p);
        p = f;
        ++count;
    }

    setArrayInt(x, p);
}

```

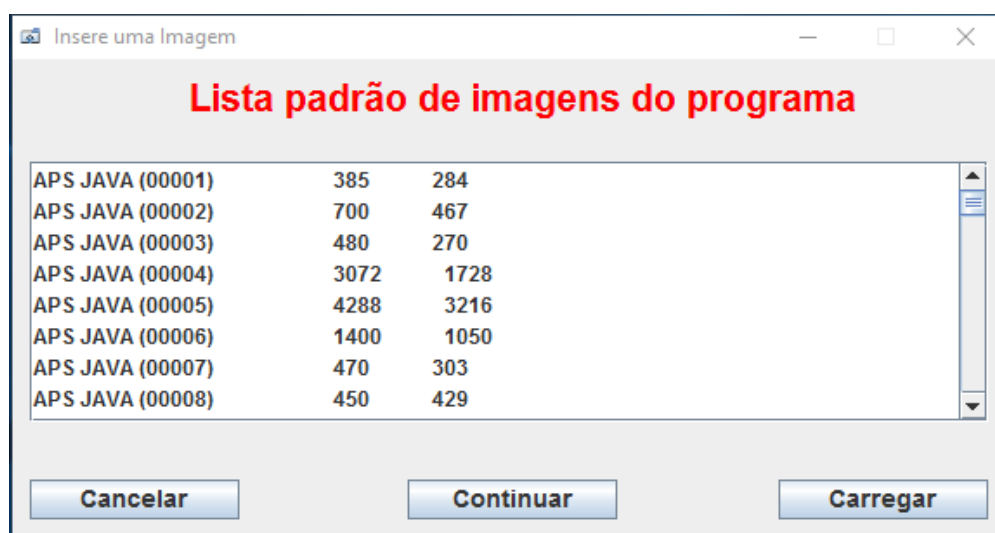
Fonte: Própria, 2016.

3.5 Lista de valores

Após as ordenações pelo *Bubble Sort*, *Sake Sort*, *Select Sort*, *Insert Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*. A lista de dado sofreu algumas alterações em sua ordem.

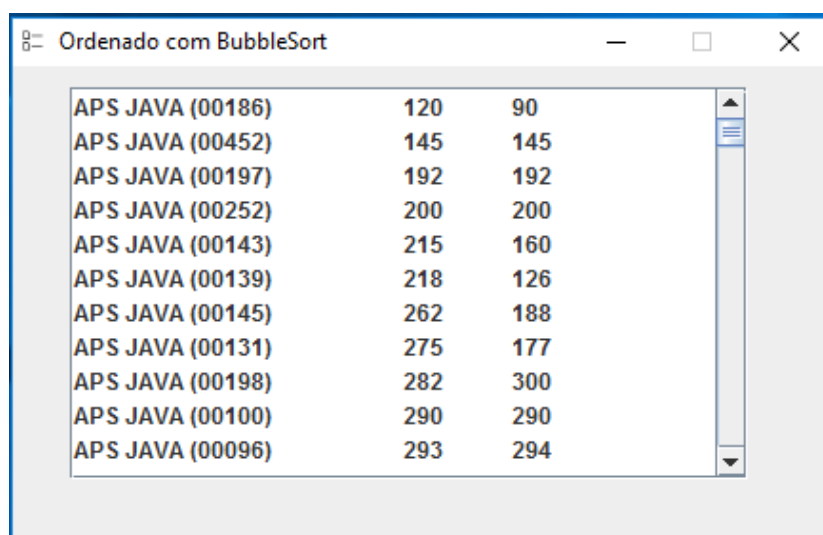
As imagens abaixo ilustram o antes e depois da ordenação, das imagens, realizada por cada algoritmo. Vale ressaltar que as imagens ilustram uma certa quantidade de dados, já que são mais de cinco milhões, seria inviável ilustrar os cinco milhões em imagem.

Imagem 21 – Lista de imagem antes.

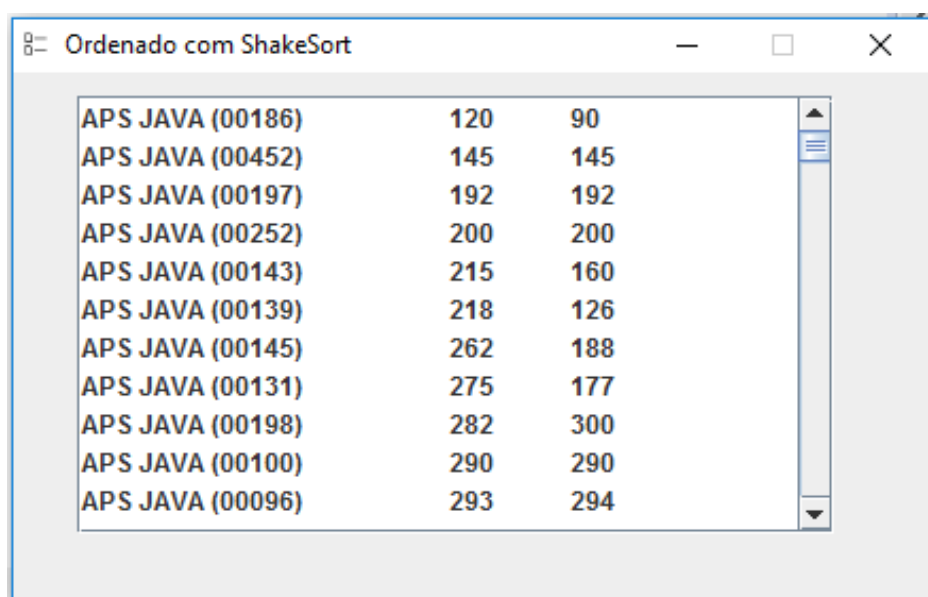


Fonte: Própria, 2016.

Imagem 22 – Lista de imagem após o *bubble*.

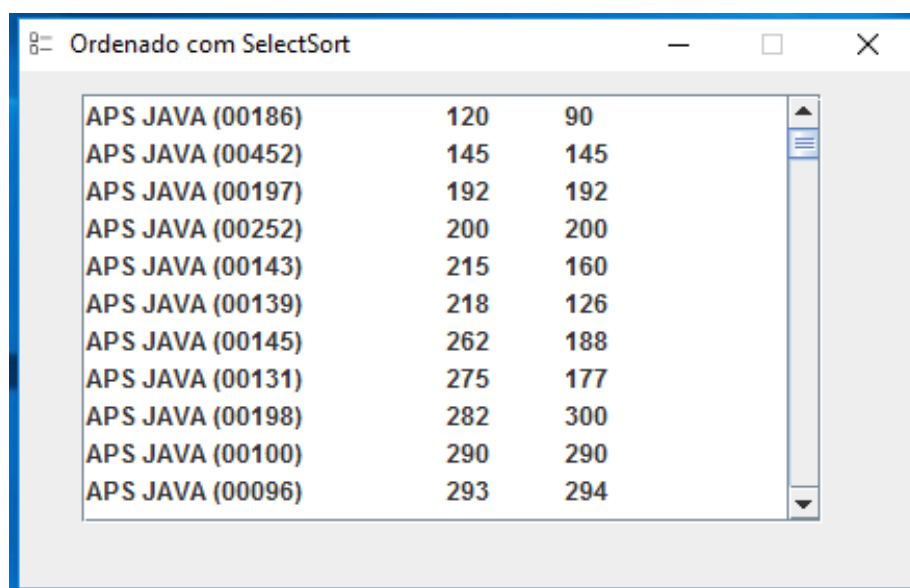


Fonte: Própria, 2016.

Imagem 23 – Lista de imagem após o *shake*.


APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

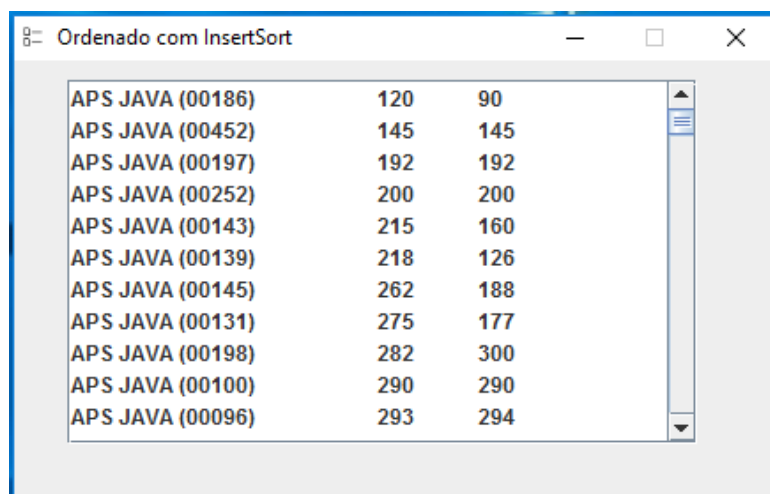
Fonte: Própria, 2016.

Imagem 24 – Lista de imagem após o *select*.


APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

Fonte: Própria, 2016.

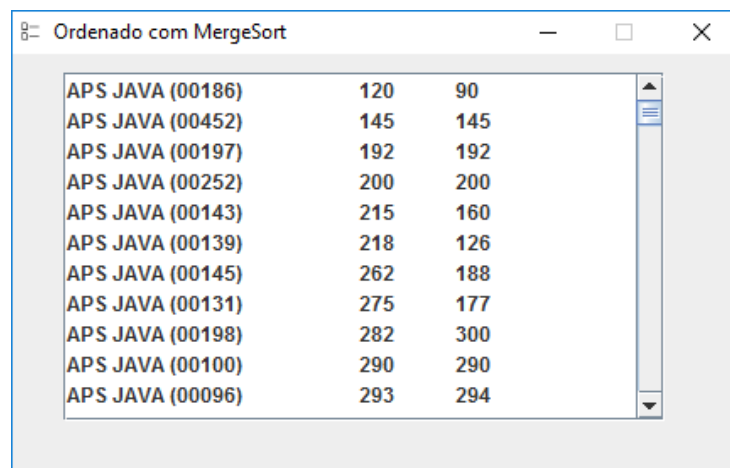
Imagem 25 – Lista de imagem após o *insert*.



APS JAVA (ID)	Value 1	Value 2
APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

Fonte: Própria, 2016.

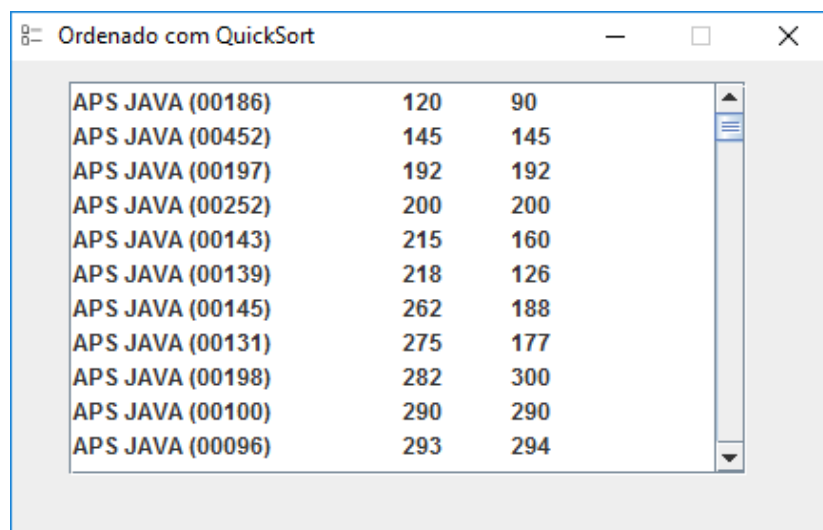
Imagem 26 – Lista de imagem após o *merge*.



APS JAVA (ID)	Value 1	Value 2
APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

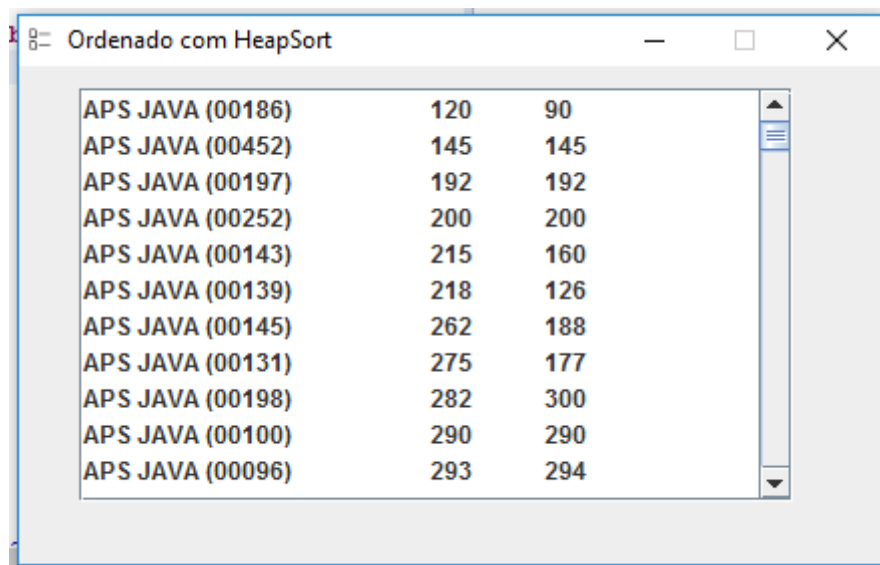
Fonte: Própria, 2016.

Imagem 27 – Lista de imagem após o *quick*.



APS JAVA (ID)	Value 1	Value 2
APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

Fonte: Própria, 2016.

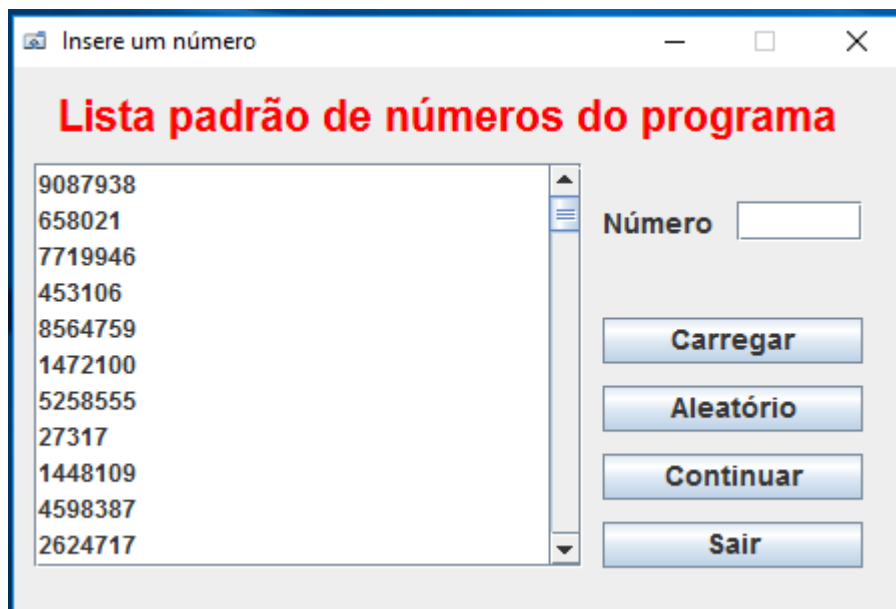
Imagem 28 – Lista de imagem após o *heap*.


Memória	Valor	Valor
APS JAVA (00186)	120	90
APS JAVA (00452)	145	145
APS JAVA (00197)	192	192
APS JAVA (00252)	200	200
APS JAVA (00143)	215	160
APS JAVA (00139)	218	126
APS JAVA (00145)	262	188
APS JAVA (00131)	275	177
APS JAVA (00198)	282	300
APS JAVA (00100)	290	290
APS JAVA (00096)	293	294

Fonte: Própria, 2016.

O conjunto de imagens abaixo ilustra a lista de números antes e depois das ordenações.

Imagem 29 – Lista de números antes.



Lista padrão de números do programa

9087938
658021
7719946
453106
8564759
1472100
5258555
27317
1448109
4598387
2624717

Número

Carregar

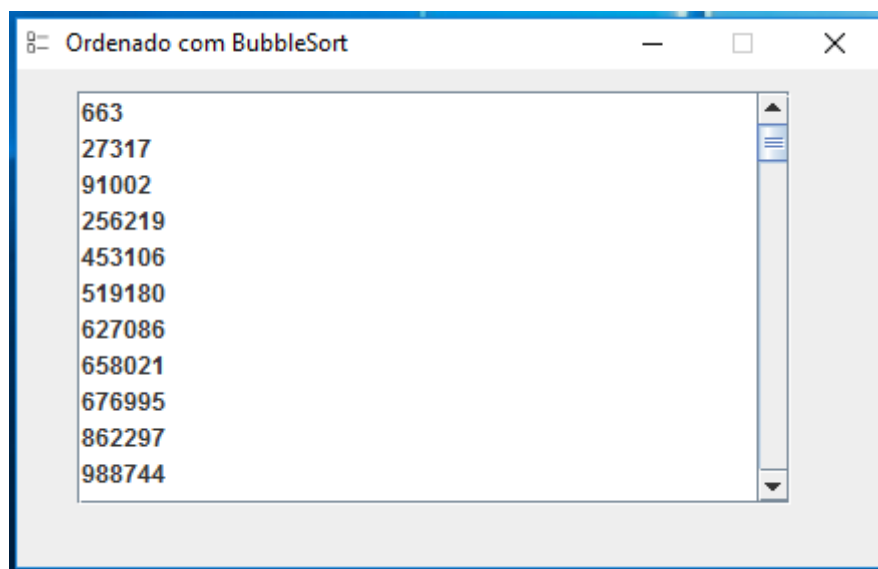
Aleatório

Continuar

Sair

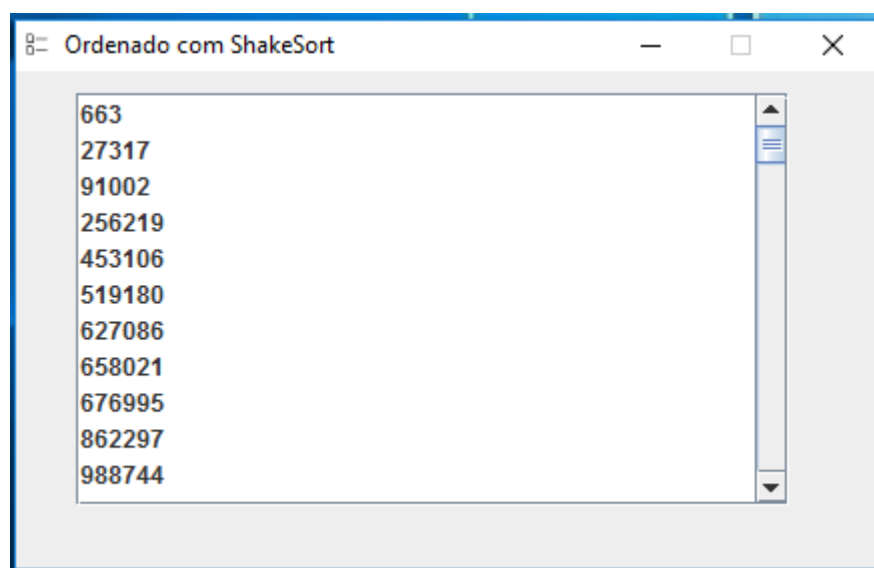
Fonte: Própria, 2016.

Imagem 30 – Lista de números após o *bubble*.



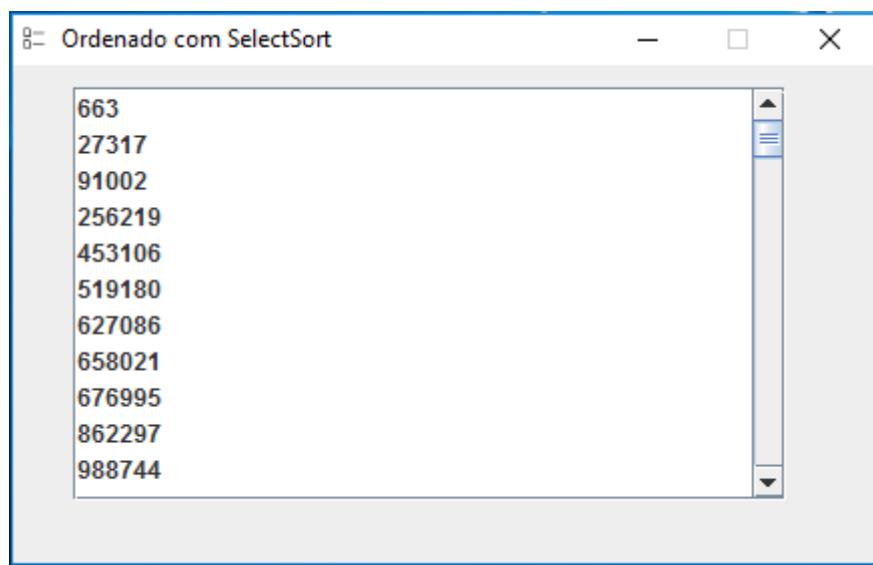
Fonte: Própria, 2016.

Imagem 31 – Lista de números após o *shake*.



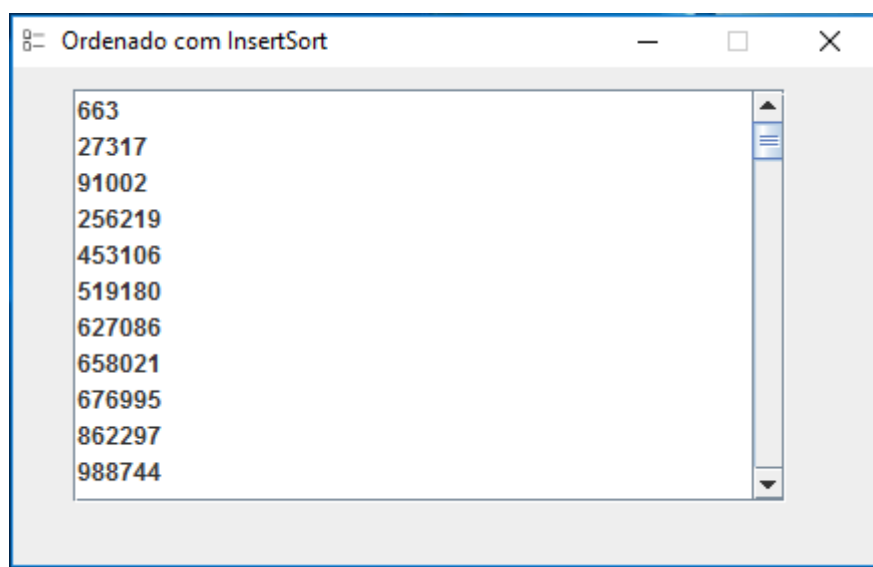
Fonte: Própria, 2016.

Imagem 32 – Lista de números após o *select*.



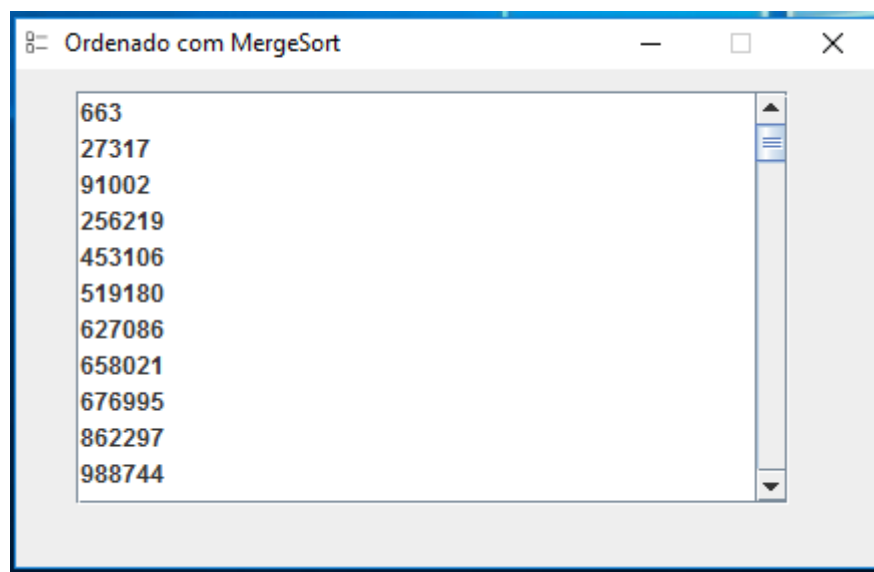
Fonte: Própria, 2016.

Imagem 33 – Lista de números após o *insert*.



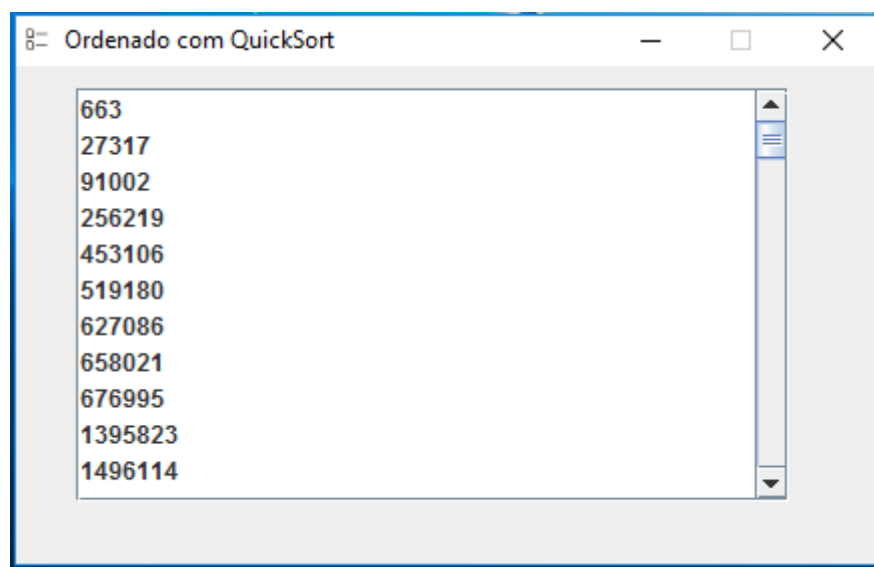
Fonte: Própria, 2016.

Imagem 34 – Lista de números após o *merge*.



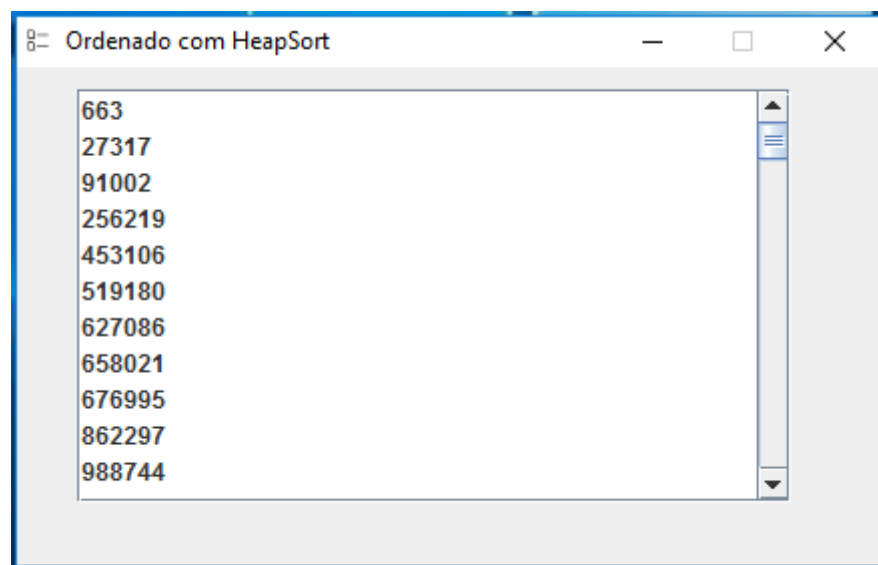
Fonte: Própria, 2016.

Imagem 35 – Lista de números após o *quick*.



Fonte: Própria, 2016.

Imagem 36 – Lista de números após o *heap*.



Fonte: Própria, 2016.

4 RESULTADOS

Nos testes do sistema computacional foram utilizados um laptop da fabricante *Samsung* com processador *Intel® Core™ i5 – 3230M CPU @ 2.60 GHz*, RAM de 8.00 GB, sistema operacional *Windows 10 Pro x64* e HD de 1.00 TB, dois computadores, um da fabricante *ACER* com processador *Intel® Core™ i5 – 525D CPU @ 1.80 GHz*, RAM de 2.00 GB, sistema operacional *Windows 10 Pro x32* e HD de 500 GB, e um da fabricante *ASUS* com processador *Intel® Core™ i3 – 4150U CPU @ 3.50 GHz*, RAM de 8.00 GB, sistema operacional *Windows 8.1 Pro x64* e HD de 500 GB e um *ultrabook* da fabricante *DELL* com processador *Intel® Atom™ - A3775 CPU @ 1.46 GHz*, RAM de 2.00 GB, sistema operacional *Windows 8.1* com *Big x32* e HD de 750 GB. O principal software utilizado no processo foi a IDE *Eclipse Java Mars x64* com acesso ao *JDK 1.8.0_92 x64* e *JRE 1.8.0_92 x64*, últimas versões até o presente momento de dissertação, *Microsoft Excel 2016*, *Microsoft Word 2016*, *Notepad* e calculadora.

Os principais livros utilizados foram “Estrutura de dados: algoritmos, análise da complexidade e implementações em Java e C/C++” de Ana Fernanda Gomes Ascencio e Graziela Santos de Araújo, e “Estruturas de Dados & Algoritmos em Java™” de Robert Lafore, ambos ser encontram na referência bibliográfica.

A utilização de quatro máquinas divergentes foi crucial para análise, porque deu a capacidade de analisarmos o desempenho de cada algoritmo em quatro cenários e ambientes distintos, e de chegarmos à uma conclusão sólida e clara.

Os testes foram capazes de demonstrar uma curiosidade que o *Bubble*, *Select* e *Merge* apresentaram, da qual estaremos abordando mais para frente.

Os cenários testados foram de cem dados aleatórios, mil dados aleatórios, cinco mil dados aleatórios, dez mil dados aleatórios, cem mil dados aleatórios, quinhentos mil dados aleatórios, um milhão de dados aleatórios e cinco milhões de dados aleatórios.

O primeiro cenário observado foi um conjunto de cem dados aleatória, onde cada máquina apresentou um conjunto divergente entre si. Nesse cenário não foi tamanha a diferença de tempo entre cada algoritmo, mas de interações foi um abismo e constatou, logo de início, quem é o melhor.

O segundo cenário observado foi o de mil dados aleatórios, onde cada máquina apresentou um conjunto divergente entre si. Nesse cenário a diferença de tempo continuou a mesma, mas as interações mantiveram a diferença.

O terceiro cenário analisado foi o de cinco mil dados, em seguida o de dez mil, onde cada máquina apresentou um conjunto divergente entre si. Nesses cenários a diferença de tempo obteve um aumento, mas mesmo assim não foi perspectivo. As interações tiveram um aumento considerável.

O quinto cenário analisado foi o de cem mil dando a capacidade de perceber a diferença de tempo, onde os anteriores não apresentaram com facilidade. As interações obtiveram uma diferença gritante em relação umas às outras.

O sexto, sétimo e oitavo cenários testados foram o de quinhentos mil, um milhão e cinco milhões de dados aleatórios, respectivamente, onde cada máquina continuou apresentando um conjunto divergente entre si. Esses cenários só deixaram claro a diferença de tempo e de interações entre cada algoritmo, tendo alguns algoritmos que demoraram dois dias para ordenar cinco milhões de dados e outros demoram apenas dois minutos para realizar o mesmo feito.

Vale ressaltar que o presente teste não avaliou o desempenho com base no tempo, porque o cenário influencia agressivamente o tempo de espera e o objetivo do trabalho é mostrar uma análise eficiente que sirva de base para futuros desempenhos de ordenação, independentemente da máquina. A análise se baseou na quantidade de interação que cada algoritmo precisou realizar para concluir a sua tarefa.

Cada algoritmo possui uma função matemática que retorna o desempenho do mesmo. Quanto maior for o retorno, pior será o algoritmo naquele caso específico. As funções utilizadas foram a quadrática e $n \log n$.

Os gráficos e tabelas apresentam os desempenhos dos algoritmos em cada máquina usada para o teste. O nome de cada máquina está representado nos gráficos e nas tabelas com as quatro primeiras letras do fabricante. Assim ficando, SAMS para *Samsung*, ASUS para *ASUS*, ACER para *ACER* e Dell para *Dell*.

A notação utilizada para classificar os algoritmos é a notação *Big O*. Ela dispensa a constante, porque ao comparar algoritmos, segundo Lafore (2004, p. 58), não é necessário se preocupar com o microprocessador ou o compilador em particular; tudo que deve ser comparado é como o algoritmo varia para valores

diferentes de n , não quais são os números realmente. Ela usa a letra O maiúscula (Ordem de).

A ideia da notação *Big O* não é fornecer valores reais para tempos de execução, mas transmitir como os tempos de execução são afetados pelo número de itens. É a maneira mais significativa de comparar algoritmos, com exceção talvez de realmente medir os tempos de execução em uma instalação real. (LAFORE, 2004, p. 59)

O computador da fabricante *ACER* com processador *Intel® Core™ i5 – 525D* CPU @ 1.80 GHz, RAM de 2.00 GB, sistema operacional *Windows 10 Pro x32* e HD de 500 GB e o *ultrabook* da fabricante *DELL* com processador *Intel® Atom™ - A3775* CPU @ 1.46 GHz, RAM de 2.00 GB, sistema operacional *Windows 8.1* com *Big x32* e HD de 750 GB não suportaram o cenário de cinco milhões de dados aleatórios, porque neste cenário foram necessários, em média, 1.4 GB de RAM para a realização da tarefa e consequentemente o programa apresentou erro por falta de memória.

Análise de algoritmo de ordenação por tempo é uma análise ineficiente e insegura, já que o cenário de desordem e a configuração da máquina influenciam no tempo de ordenação.

Durante o processo de teste foi constatado que ao chegar no cenário de cinco milhões de dados desordenado, duas máquinas, já citadas, não suportaram o processo, porque o sistema computacional estava consumindo, em média, 1.4 GB de RAM para realizar tal feito. Com base nessa incapacidade foi comprovado que o tempo não determina a sofisticação de um algoritmo, porque se estivessemos fazendo com base no tempo, não seria possível determinar o desempenho nas máquinas em questão.

O processo de análise também constatou que o cenário influencia na ordenação. Um dos cenários que apresentou, com clareza, este resultado foi o de um milhão de dados desordenados, onde os algoritmos que mais demonstraram divergência de tempo foram *Bubble* e o *Select*. O *Bubble* na máquina da fabricante *Samsung* demorou, aproximadamente, cinco horas para concluir a tarefa, na máquina da fabricante *ACER* demorou, aproximadamente, oito horas, na máquina da fabricante *ASUS* demorou, aproximadamente, três horas e na máquina da

fabricante *DELL* foram dez horas, aproximadamente. O *Select* levou basicamente a mesma diferença de tempo.

Mesmo com a mesma quantidade de dados, diferença de tempo de ordenação, de máquina e de cenário, os algoritmos apresentaram, em todas as máquinas, o mesmo número de interação, em alguns algoritmos. Constatando que a análise por tempo é ineficiente, porque máquinas com configurações mais pesadas atingem um desempenho superior as fracas, mas os números de interações são os mesmos.

Cenário não quer dizer a quantidade de dados, mas sim o grau de desordem dos dados. Em cada máquina foram utilizados a mesma quantidade de dados, porém o grau de desordem foi diferente.

Alguns algoritmos obtiveram o mesmo valor em todas as máquinas, com isso foi possível determinar a quantidade interação deles nas máquinas que não suportaram determinado cenário, mas outros algoritmos eram influenciados pelo cenário e com isso não foi possível determinar a quantidade de interação necessária para concluir a tarefa específica.

4.1 Análise individual

Primeiro iremos analisar individualmente cada algoritmo, apresentando os resultados em gráficos e tabelas. Após a análise individual, partiremos para a análise conjunta.

4.1.1 *Bubble Sort*

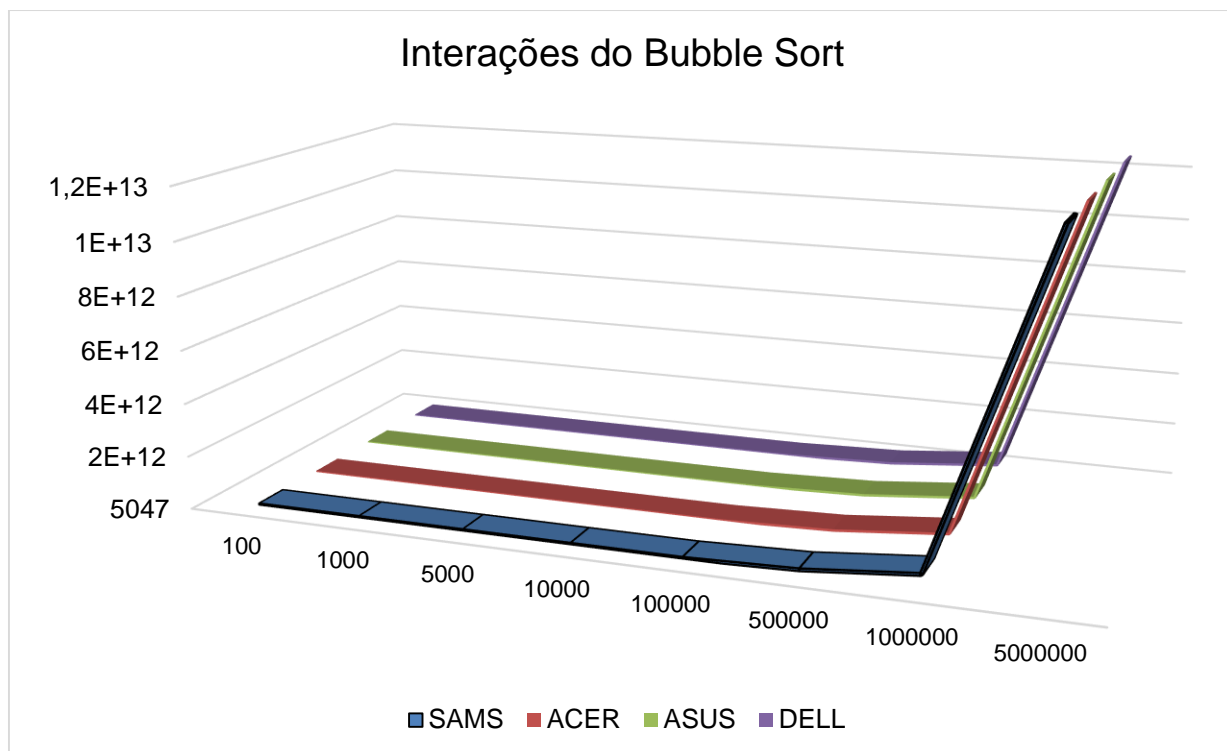
O algoritmo *Bubble Sort* apresentou o segundo pior desempenho de todos, com base nas interações, ganhando apenas do *Select Sort* no quesito interações, mas no desempenho geral ficou claro que o *Bubble Sort* não é o melhor.

Em todos os cenários os números de interações foram gigantescos em comparação com a quantidade de dados sendo ordenados.

O gráfico 01 mostra os números de interações em cada cenário.

Analisando o gráfico podemos perceber que em todas as máquinas o *Bubble Sort* manteve o mesmo número de interação, isso significa que independentemente da máquina e cenário o número de interação será o mesmo.

O *Bubble Sort* possui um laço aninhado em outro e executa em tempo $O(N^2)$. O laço externo é executado n vezes e o laço interno é executado n (ou talvez n dividido por alguma constante) vezes para cada ciclo do laço externo. Isso significa que o *Bubble Sort* está fazendo, no pior caso, alguma coisa aproximadamente $n * n$ ou N^2 vezes e no melhor caso $O(N)$, onde n é o número de dados.

Gráfico 01 – Interações do *bubble*.

Fonte: Própria, 2016.

A tabela abaixo mostra com clareza os números de interações.

Tabela 01 – Interações do *bubble*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	5047	5047	5047	5047
1000	500497	500497	500497	500497
5000	12502497	12502497	12502497	12502497
10000	50004997	50004997	50004997	50004997
100000	5000049997	5000049997	5000049997	5000049997
500000	125000249997	125000249997	125000249997	125000249997

1000000	500000499997	500000499997	500000499997	500000499997
5000000	12500002499997	12500002499997	12500002499997	12500002499997

Fonte: Própria, 2016.

Isso deixa evidente o quanto os dados influenciam o *Bubble Sort*. O interessante é que mesmo com os cenários e máquinas distintos foi mantido o mesmo número de interação e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

Vale ressaltar que cada máquina apresentou um cenário distinto, ou seja, foram realizados testes em quatro cenários diferentes para cada algoritmo.

Bubble Sort faz parte da categoria ordenação simples, porque o seu desenvolvimento não possui um grau de dificuldade elevado.

4.1.2 *Shake Sort*

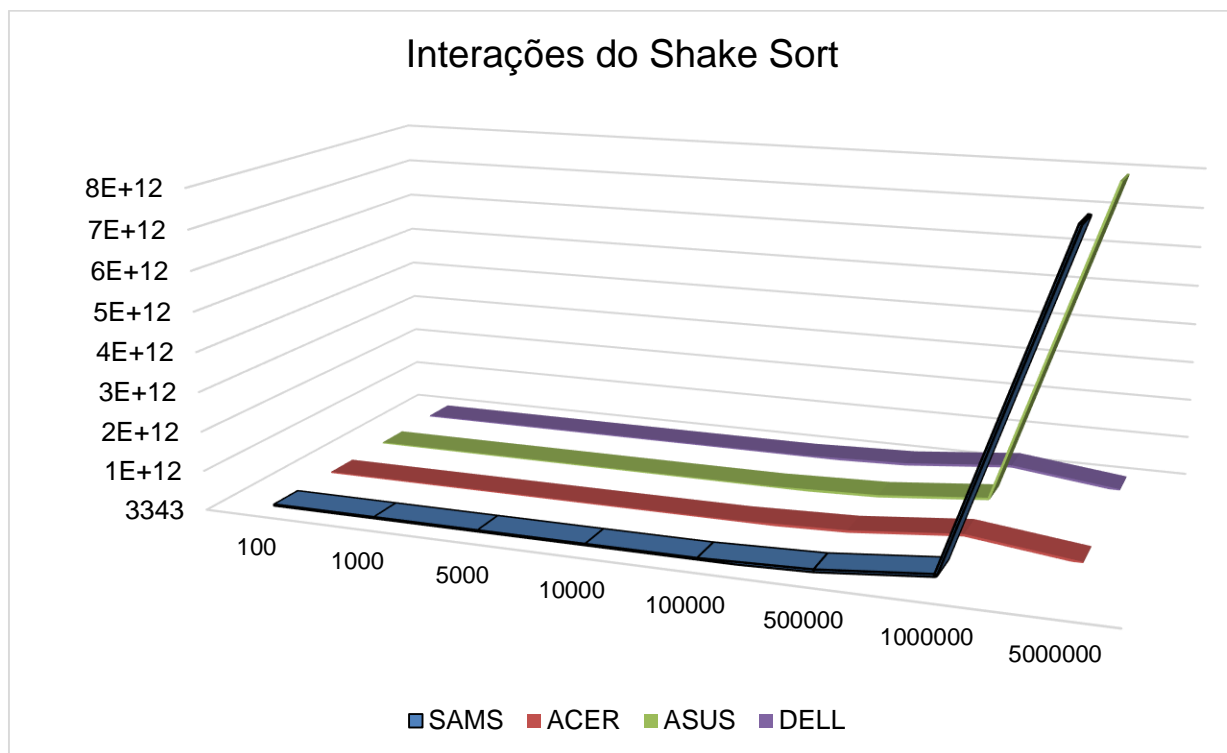
O algoritmo *Shake Sort* apresentou o terceiro pior desempenho, com base nas interações, ficando à frente do *Bubble Sort* e do *Select Sort*.

Em todos os cenários os números de interações foram grandes em comparação com as outras interações de alguns algoritmos.

O gráfico 02 mostra os números de interações em cada cenário.

Analisando o gráfico podemos perceber que em todos os cenários o *Shake Sort* sofreu uma leve alteração nos números de interações, isso significa que o cenário influencia no algoritmo, assim sendo incapaz a comprovação do seu desempenho com base no tempo.

O *Shake Sort* possui dois laços encadeados e executa em tempo $O(N^2)$. O laço externo é executado n vezes e os dois laços internos são executados n (ou talvez n dividido por alguma constante) vezes para cada ciclo do laço externo. Isso significa que o *Shake Sort* está fazendo, no pior caso, alguma coisa aproximadamente $n * n$ ou N^2 vezes e no melhor caso $O(N)$, onde n é o número de dados.

Gráfico 02 – Interações do *shake*.

Fonte: Própria, 2016.

A tabela abaixo mostra com clareza os números de interações.

Tabela 02 – Interações do *shake*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	3582	3343	3538	3096
1000	343358	344250	340852	328099
5000	8431556	8425861	8275047	8270750
10000	33021068	33334137	33377533	33146316
100000	3344022676	3337448985	3334055345	3331438158
500000	83244521252	83348708809	83285160247	83337252283
1000000	333253170626	333541489327	333462098470	333341489327
5000000	8351257698058	00000000000000	8351048770214	00000000000000

Fonte: Própria, 2016.

Isso deixa evidente o quanto a quantidade de dados influencia o *Shake Sort*, mas não quanto como o *Bubble Sort* e o *Select Sort*, que estaremos analisando em

seguida. O interessante é que o *Shake Sort* não manteve o mesmo número de interações, deixando claro que o cenário influencia consideravelmente e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

Shake Sort faz parte da categoria ordenação simples, porque o seu desenvolvimento não possui um grau de dificuldade elevado.

4.1.3 *Select Sort*

O algoritmo *Select Sort* apresentou o pior desempenho, com base nas interações, mas isso não significa, em termos gerais, que ele seja o pior dos algoritmos simples.

Em todos os cenários os números de interações foram grandes em comparação com as outras interações de alguns algoritmos.

O gráfico 03 mostra os números de interações em cada cenário.

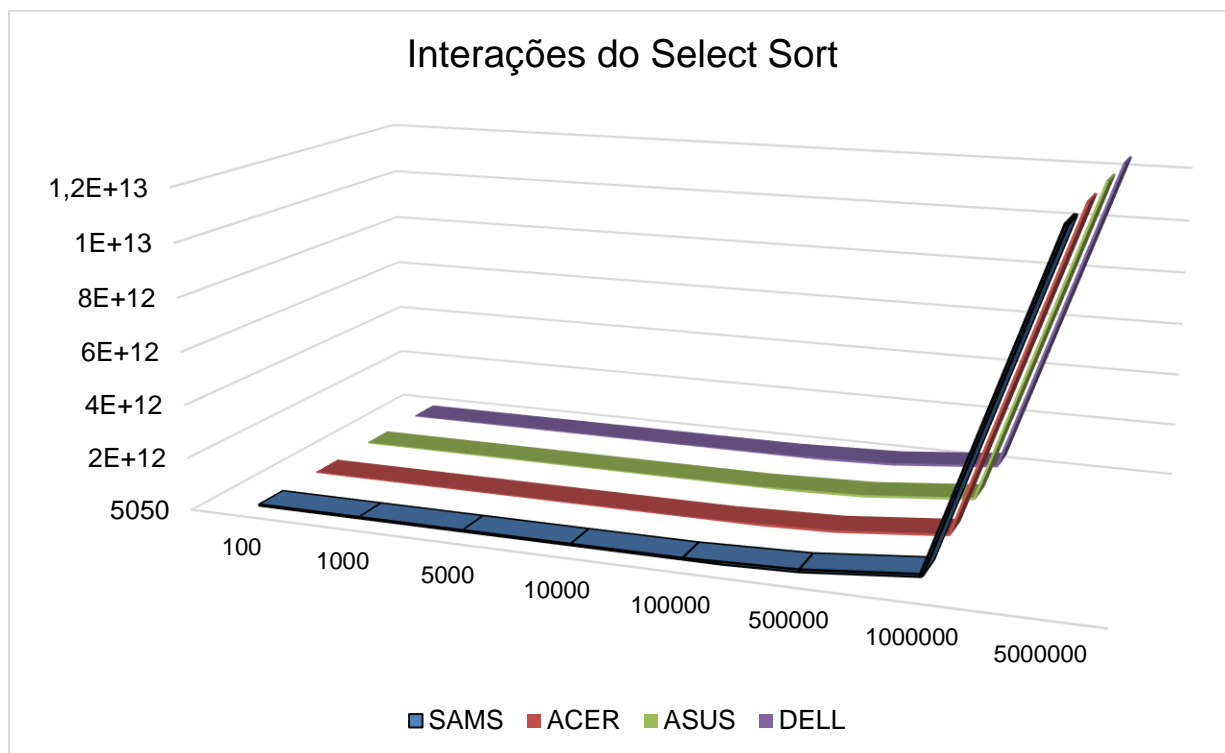
Analisando o gráfico podemos perceber que em todas as máquinas o *Select Sort* manteve o mesmo número de interação, isso significa que independentemente da máquina e cenário o número de interação será o mesmo.

O *Select Sort* possui um laço aninhado em outro e executa em tempo $O(N^2)$. O laço externo é executado n vezes e o laço interno é executado n (ou talvez n dividido por alguma constante) vezes para cada ciclo do laço externo. Isso significa que o *Select Sort* está fazendo, no pior caso, alguma coisa aproximadamente $n * n$ ou N^2 vezes e no melhor caso $O(N)$, onde n é o número de dados.

A tabela 03, mostra com clareza os números de interações e evidencia o quanto a quantidade de dados influencia o *Select Sort* mais que com o *Bubble Sort*. O interessante é que mesmo com os cenários e máquinas distintos foi mantido o mesmo número de interação e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

Na análise conjunta estaremos evidenciando o porquê do *Select Sort* ser melhor que o *Bubble Sort* e *Shake Sort*, mesmo com os números de interações sendo maiores.

Select Sort faz parte da categoria ordenação simples, porque o seu desenvolvimento não possui um grau de dificuldade elevado.

Gráfico 03 – Interações do *select*.

Fonte: Própria, 2016.

Tabela 03 – Interações do *select*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	5050	5050	5050	5050
1000	500500	500500	500500	500500
5000	12502500	12502500	12502500	12502500
10000	50005000	50005000	50005000	50005000
100000	5000050000	5000050000	5000050000	5000050000
500000	125000250000	125000250000	125000250000	125000250000
1000000	500000500000	500000500000	500000500000	500000500000
5000000	12500002500000	12500002500000	12500002500000	12500002500000

Fonte: Própria, 2016.

4.1.4 Insert Sort

O algoritmo *Insert Sort* apresentou o quarto pior desempenho, com base nas interações, ficando à frente do *Bubble Sort*, *Shake Sort* e *Select Sort*.

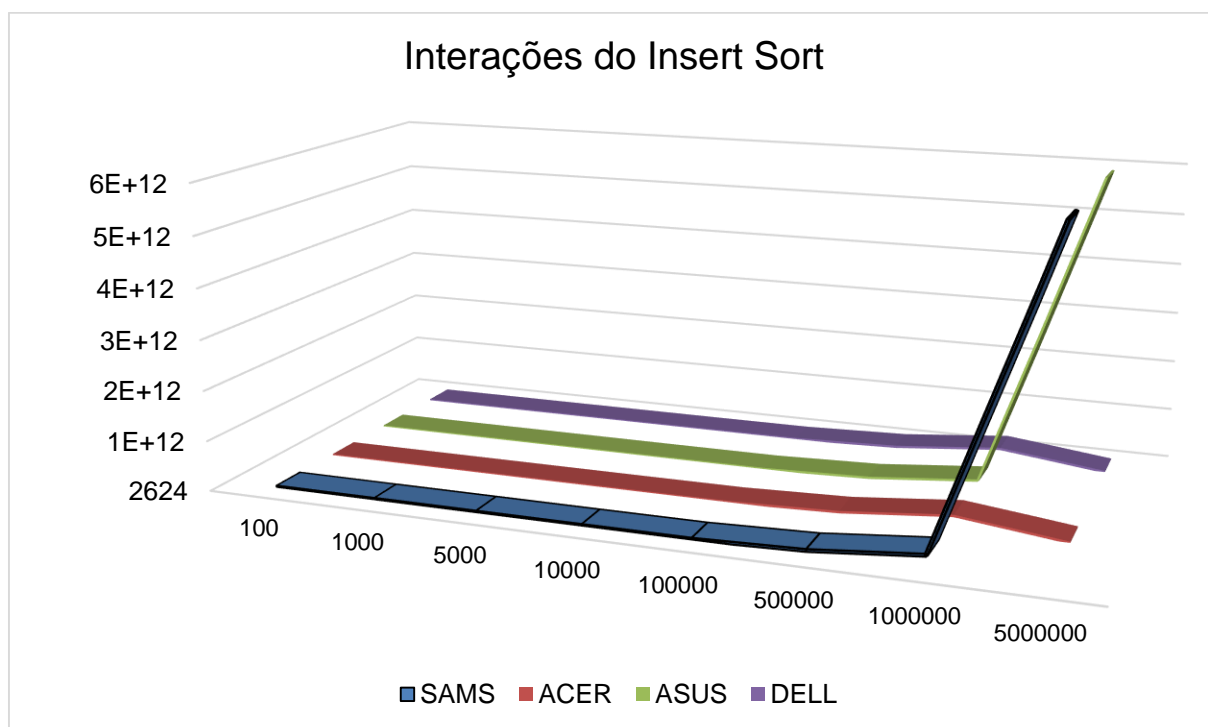
Em todos os cenários os números de interações foram grandes em comparação com as outras interações de alguns algoritmos, como o *Quick Sort*.

O gráfico 04 mostra os números de interações em cada cenário

Analisando o gráfico podemos perceber que em todos os cenários o *Insert Sort* sofreu uma leve alteração nos números de interações, isso significa que o cenário influencia no algoritmo, assim sendo incapaz a comprovação do seu desempenho com base no tempo.

O *Insert Sort* possui um laço aninhado em outro e executa em tempo $O(N^2)$. O laço externo é executado n vezes e o laço interno é executado n (ou talvez n dividido por alguma constante) vezes para cada ciclo do laço externo. Isso significa que o *Insert Sort* está fazendo, no pior caso, alguma coisa aproximadamente $n * n$ ou N^2 vezes e no melhor caso $O(N)$, onde n é o número de dados.

Gráfico 04 – Interações do *insert*.



Fonte: Própria, 2016.

A tabela abaixo mostra com clareza os números de interações e evidencia o quanto a quantidade de dados influencia o *Insert Sort*, mas não tanto como o *Bubble Sort*, *Shake Sort* e o *Select Sort*. O interessante é que o *Insert Sort* não manteve o mesmo número de interações, deixando claro que o cenário influencia consideravelmente e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

Outra coisa que podemos observar na tabela é que, praticamente, o *Insert Sort* obteve a metade de interação do *Bubble Sort* e *Select Sort* em todos os cenários e máquinas.

Tabela 04 – Interações do *insert*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	2652	2624	2498	2392
1000	255843	256154	257668	245821
5000	6336447	6271975	6245008	6168510
10000	24724004	25019704	25079237	24748364
100000	2509084602	2505535219	2500139125	2496985538
500000	62399141410	62523299490	62494372272	62468636972
1000000	249952047223	250132229685	250099189024	250052089024
5000000	6248226465121	00000000000000	6249425735378	00000000000000

Fonte: Própria, 2016.

Entre os quatro algoritmos apresentados até o momento, o *Insert Sort* é o melhor, mesmo os três sendo $O(N^2)$. Mais para frente demonstraremos a comparação conjunta e ela evidenciará os melhores de cada categoria.

Insert Sort faz parte da categoria ordenação simples, porque o seu desenvolvimento não possui um grau de dificuldade elevado.

4.1.5 Merge Sort

O algoritmo *Merge Sort* apresentou o terceiro melhor desempenho, com base nas interações, ficando atrás do *Quick Sort* e *Heap Sort*.

Em todos os cenários os números de interações foram controlados e não tiveram um salto absurdo quando a quantidade de dados foi de quinhentos mil para cinco milhões.

O gráfico 05 mostra os números de interações em cada cenário.

Lembrando que cenário é o grau de desordem e não a quantidade de dados.

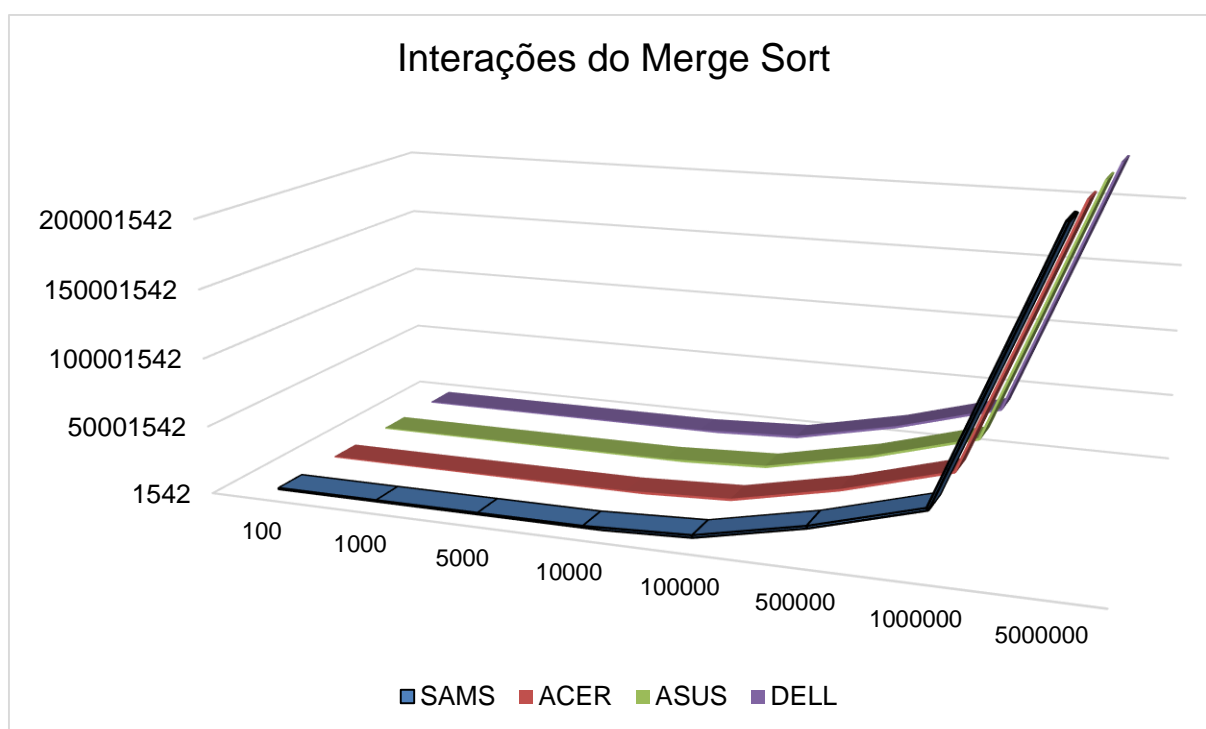
Analisando o trecho do código, capítulo 3.4.5, verifica-se que o algoritmo é recursivo e possui três chamadas de função, sendo que as duas primeiras são

chamadas recursivas e recebem metade dos elementos do vetor passado, e a outra é a chamada para a função que realiza a intercalação das duas metades.

A linha de comparação e a linha de atribuição gastam tempo constante, $O(1)$, e na notação *Big O* a constante é irrelevante para calcular o desempenho.

Merge Sort faz parte da categoria ordenação avançada e recursiva. A função que calcula o desempenho é $O(n \log n)$. Ela é uma função $n \log n$, porque é realizado n divisões ao decorrer do processo. Quando é realizado uma divisão ou multiplicação, a função se torna logarítmica e quando se tem uma soma ou subtração ela é n .

Gráfico 05 – Interações do *merge*.



Fonte: Própria, 2016.

Analisando o gráfico podemos perceber que em todos os cenários o *Merge Sort* manteve o mesmo número de interação, isso significa que independentemente da máquina e cenário o número de interação será o mesmo.

A tabela 05, mostra com clareza os números de interações e evidencia o quanto a quantidade de dados influencia o *Merge Sort*. O interessante é que mesmo com os cenários e máquinas distintos foi mantido o mesmo número de interação e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

Tabela 05 – Interações do *merge*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	1542	1542	1542	1542
1000	21950	21950	21950	21950
5000	133614	133614	133614	133614
10000	287230	287230	287230	287230
100000	3537854	3537854	3537854	3537854
500000	19951422	19951422	19951422	19951422
1000000	41902846	41902846	41902846	41902846
5000000	233222782	233222782	233222782	233222782

Fonte: Própria, 2016.

4.1.6 Quick Sort

O algoritmo *Quick Sort* apresentou o melhor desempenho, com base nas interações.

Em todos os cenários os números de interações foram controlados e não tiveram um salto absurdo quando a quantidade de dados foi de quinhentos mil para cinco milhões, assim como o *Merge Sort*.

Analisando o trecho do código, capítulo 3.4.6, verifica-se que o algoritmo é recursivo e possui cinco chamadas de função, sendo que a primeira é para quando o vetor estiver pequeno, a segunda para encontrar o pivô, a seguinte para dividir o vetor, semelhante ao *Merge Sort*, e as duas últimas são chamadas recursivas e recebem as extremidades do vetor passado, e a outra é a chamada para a função que realiza a intercalação das duas metades.

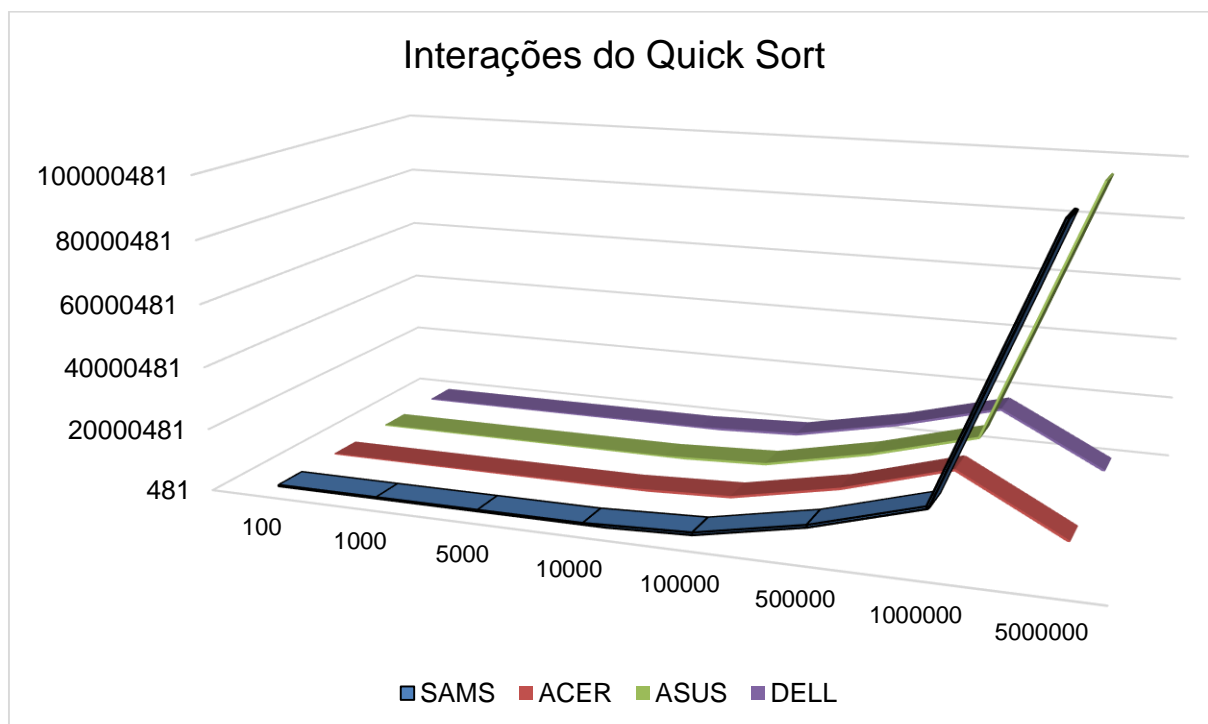
A linha de comparação e a linha de atribuição gastam tempo constante, $O(1)$, e na notação *Big O* a constante é irrelevante para a calcular o desempenho.

Quick Sort faz parte da categoria ordenação avançada e recursiva. A função que calcula o desempenho é $O(n \log n)$. Ela é uma função $n \log n$, porque é realizado n divisões ao decorrer do processo.

O desempenho do *Quick Sort* depende, segundo Lafore (2004) e Ascencio e Araújo (2010), se o particionamento é ou não balanceado, e isto depende de qual elemento será o pivô.

Ascencio e Araújo (2010, p. 89) dizem que se o pivô é balanceado, o algoritmo executa tão rápido quanto o *Merge Sort*, $O(n \log n)$. Se não é balanceando, o algoritmo é tão lento quanto o *Insert Sort*, $O(N^2)$.

Gráfico 06 – Interações do *quick*.



Fonte: Própria, 2016.

A tabela abaixo mostra com clareza os números de interações.

Tabela 06 – Interações do *quick*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	487	560	481	503
1000	8000	8103	8085	8276
5000	50878	54704	49590	53299
10000	107227	110763	108896	109037
100000	1446768	1438831	1446505	1443378
500000	8114423	8190660	8530391	8359573
1000000	17914996	18080499	17813999	17307385
5000000	101997752	00000000000000	100674737	00000000000000

Fonte: Própria, 2016.

Analisando a tabela podemos evidenciar o quanto a quantidade de dados influencia o *Quick Sort*, mas não tanto como o *Merge Sort* e o *Heap Sort*, analisaremos ele em seguida. O interessante é que o *Quick Sort* não manteve o mesmo número de interações, deixando claro que o cenário influencia consideravelmente e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

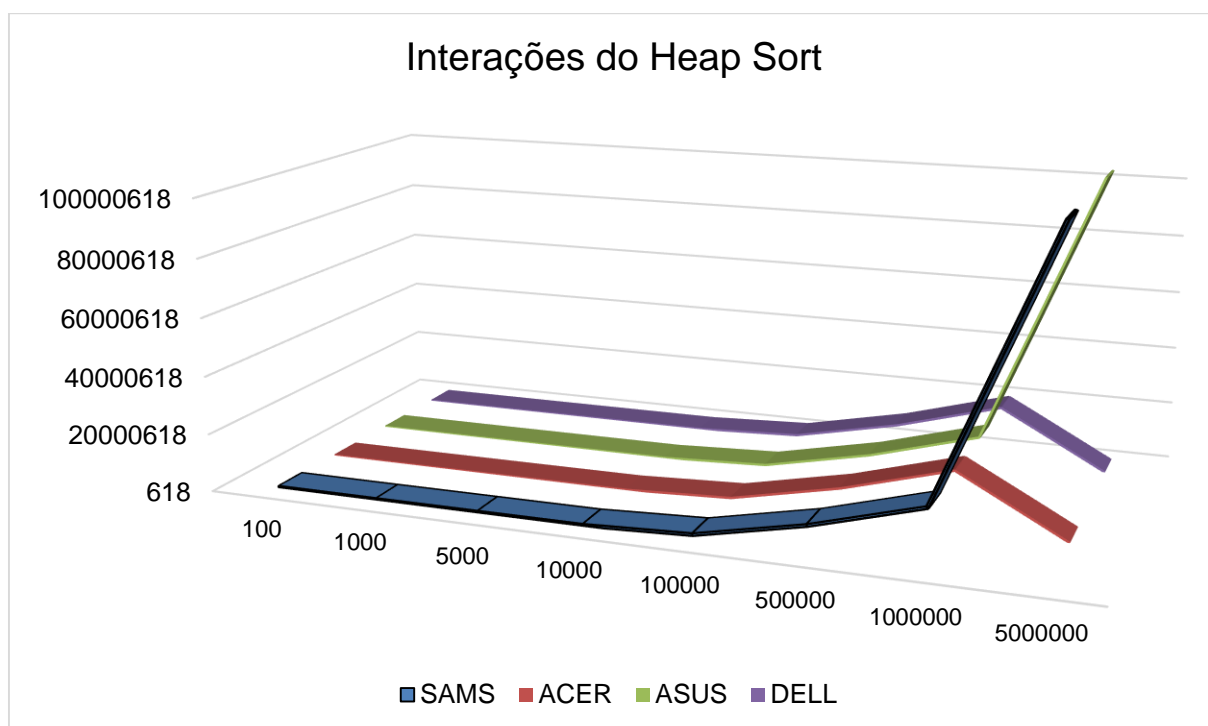
4.1.7 Heap Sort

O algoritmo *Heap Sort* apresentou o segundo melhor desempenho, com base nas interações, ficando atrás do *Quick Sort* e na frente do *Heap Sort*.

Em todos os cenários os números de interações foram controlados e não tiveram um salto absurdo quando a quantidade de dados foi de quinhentos mil para cinco milhões, assim como o *Quick Sort* e *Merge Sort*.

O gráfico abaixo mostra os números de interações em cada cenário.

Gráfico 07 – Interações do heap.



Fonte: Própria, 2016.

Analisando o trecho do código, capítulo 3.4.7, verifica-se que o algoritmo não é recursivo e possui dois laços de repetição. O primeiro laço é executado chamando uma função e ele vai até a metade do conjunto. O segundo laço é executado

chamando a mesma função que o primeiro, porém ele ordena dois elementos antes de chamar a função.

O *Heap Sort* não é recursivo, mas isso não o torna $O(N^2)$. Se analisarmos com cuidado o trecho do primeiro laço de repetição, veremos uma divisão na primeira condição do laço e isso tornar o primeiro laço $\log n$. Continuemos a análise e veremos que o segundo laço de repetição está dentro do primeiro, tornando ele múltiplo. Com isso concluímos que o *Heap Sort* é $O(n \log n)$, onde o primeiro n pertence ao segundo laço e o logarítmico pertence ao primeiro.

Heap Sort faz parte da categoria ordenação avançada.

A tabela abaixo mostra com clareza os números de interações.

Tabela 07 – Interações do *heap*.

INTERAÇÕES				
Cenários	SAMS	ACER	ASUS	DELL
100	634	618	636	630
1000	9538	9576	9532	9605
5000	59532	59631	59674	59702
10000	129226	129291	129212	129326
100000	1624593	1624677	1625086	1624660
500000	9274623	9273433	9274089	9274525
1000000	19548203	19547334	19548555	19547767
5000000	109422154	0000000000000000	109414254	0000000000000000

Fonte: Própria, 2016.

Analisando a tabela podemos evidenciar o quanto a quantidade de dados influencia o *Heap Sort*, mas não tanto como o *Merge Sort*. O interessante é que o *Heap Sort* não manteve o mesmo número de interações, assim como o *Merge Sort*, deixando claro que o cenário influencia consideravelmente e em nenhum caso foi o de pior ou melhor, mas sim caso médio.

4.2 Análise conjunta

Foi selecionado o melhor desempenho de cada algoritmo em todos os cenários para a realização da análise conjunta.

O foco desta análise é evidenciar a qualidade de cada algoritmo em certos cenários e detalhar com clareza a complexidade dos mesmos.

Separamos em duas categorias, porque os algoritmos $O(n \log n)$ são mais rápidos que os $O(N^2)$ e não seria viável compará-los.

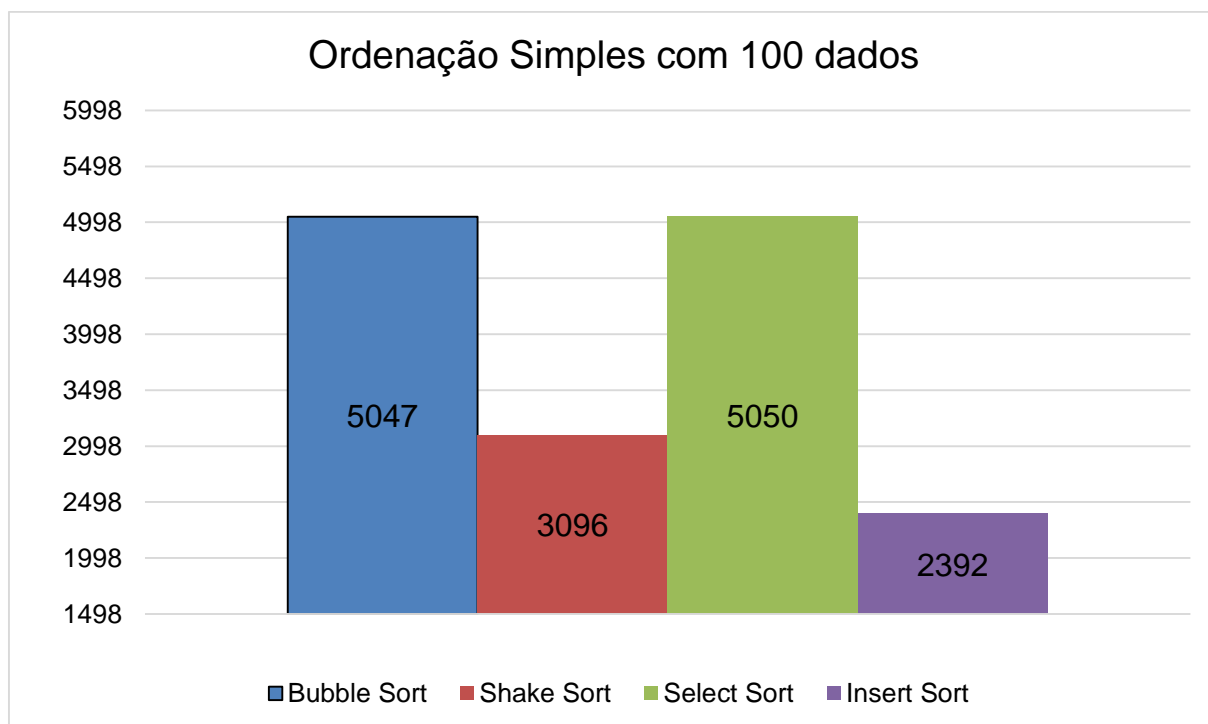
4.2.1 Ordenação simples

Os algoritmos que farão parte da análise conjunto simples são o *Bubble Sort*, *Shake Sort*, *Select Sort* e *Insert Sort*, quatro ao todo.

Nesta etapa estaremos evidenciando o motivo do *Insert Sort* ser o mais rápido entre os quatro simples.

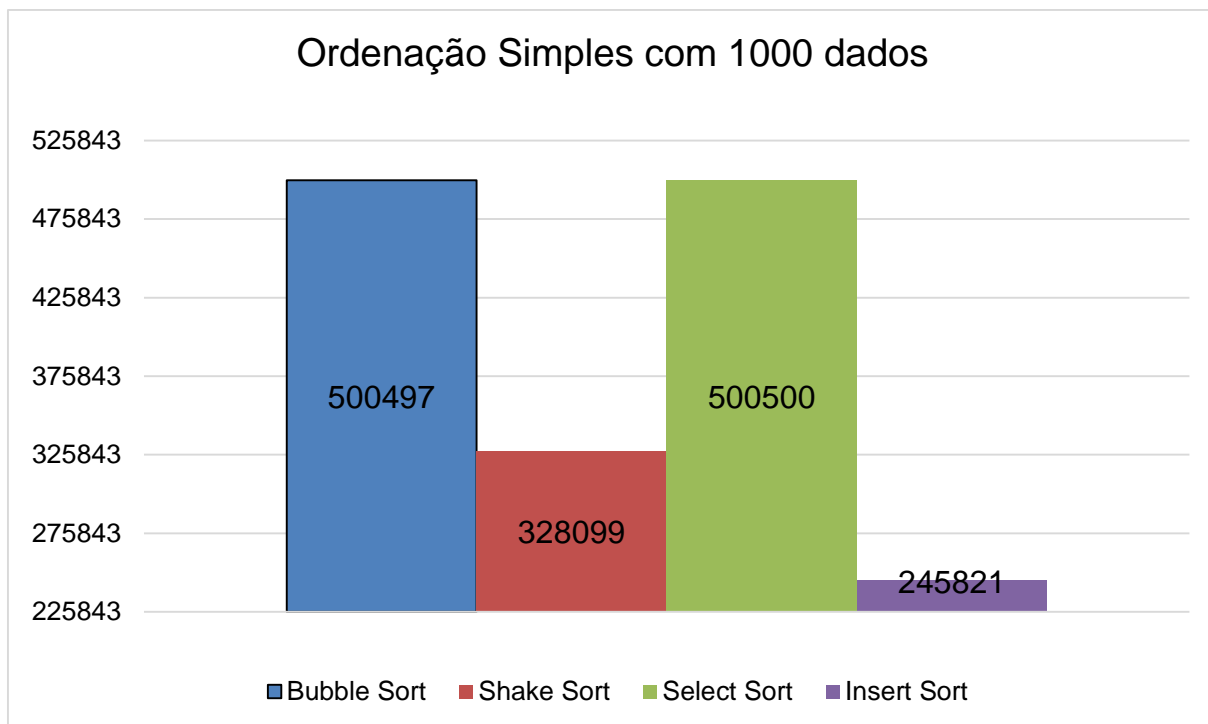
Os gráficos abaixo evidenciam o desempenho de cada algoritmo simples em todos os cenários.

Gráfico 08 – Ordenação simples de cem dados.



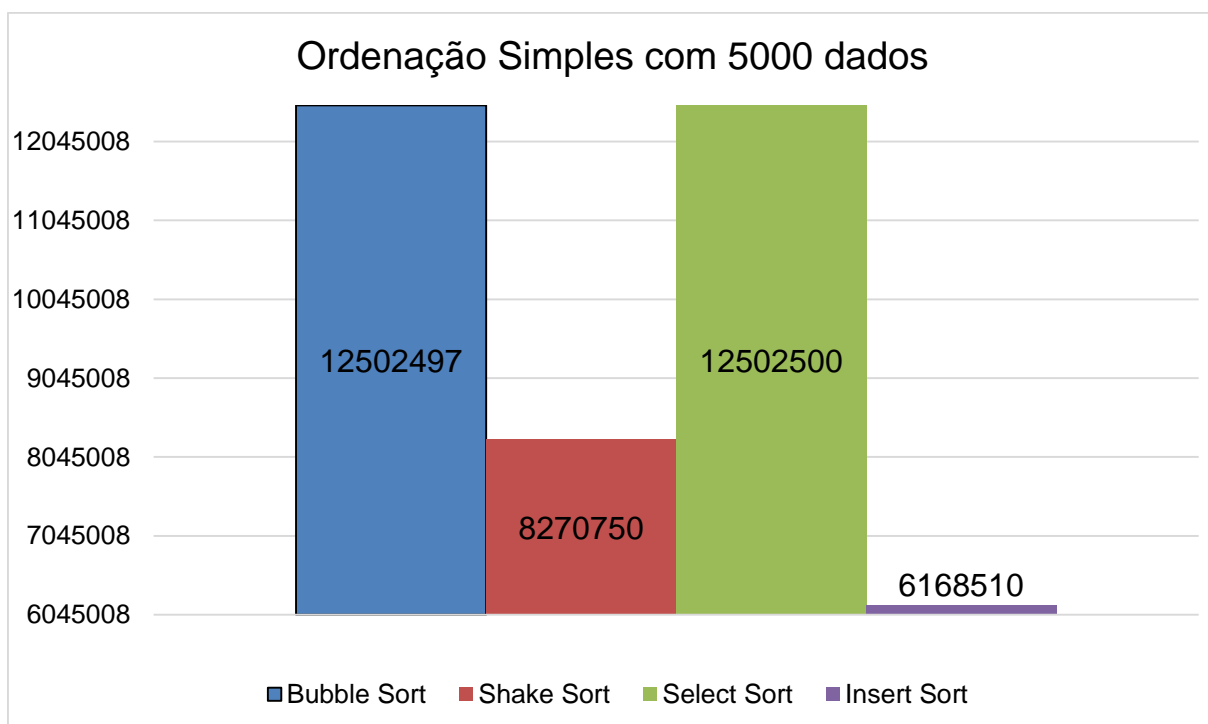
Fonte: Própria, 2016.

Gráfico 09 – Ordenação simples de mil dados.



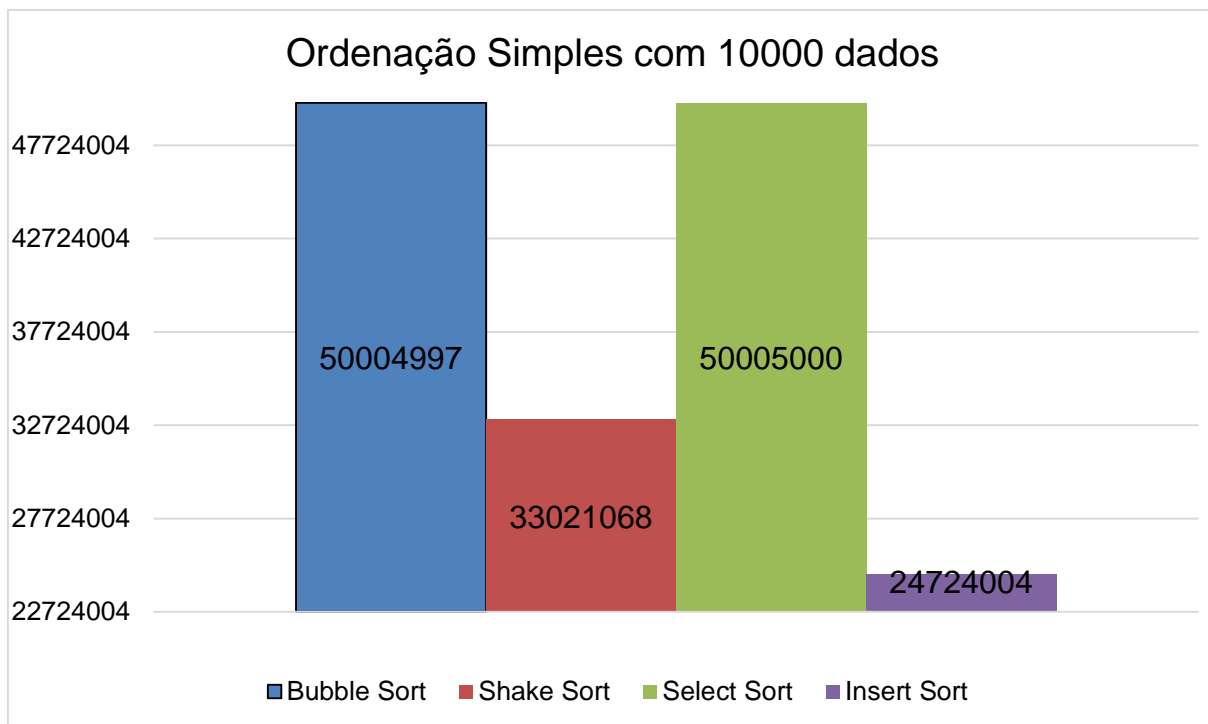
Fonte: Própria, 2016.

Gráfico 10 – Ordenação simples de cinco mil dados.



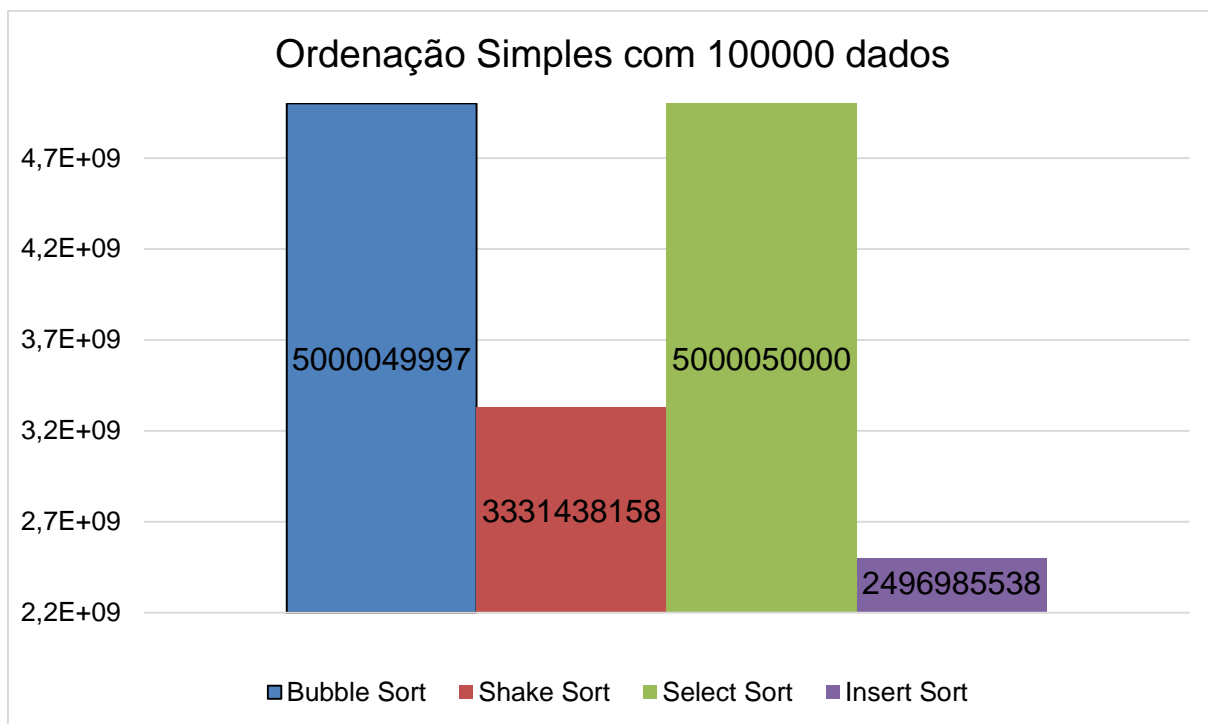
Fonte: Própria, 2016.

Gráfico 11 – Ordenação simples de dez mil dados.



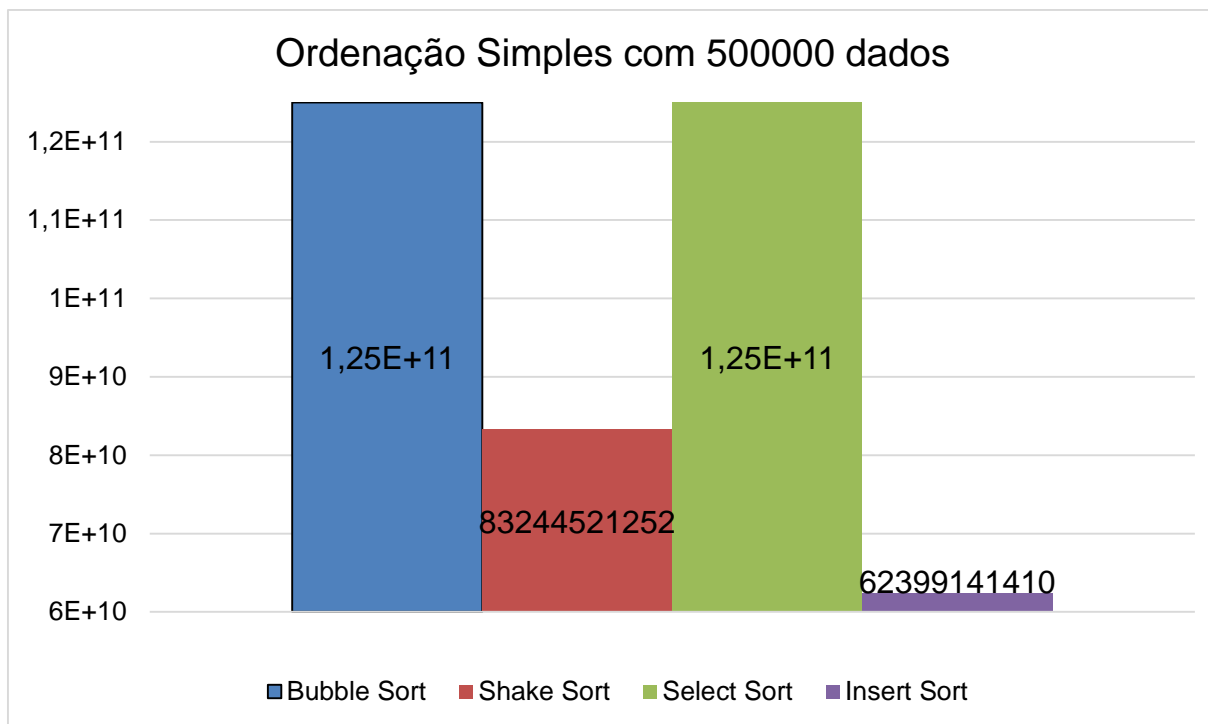
Fonte: Própria, 2016.

Gráfico 12 – Ordenação simples de cem mil dados.



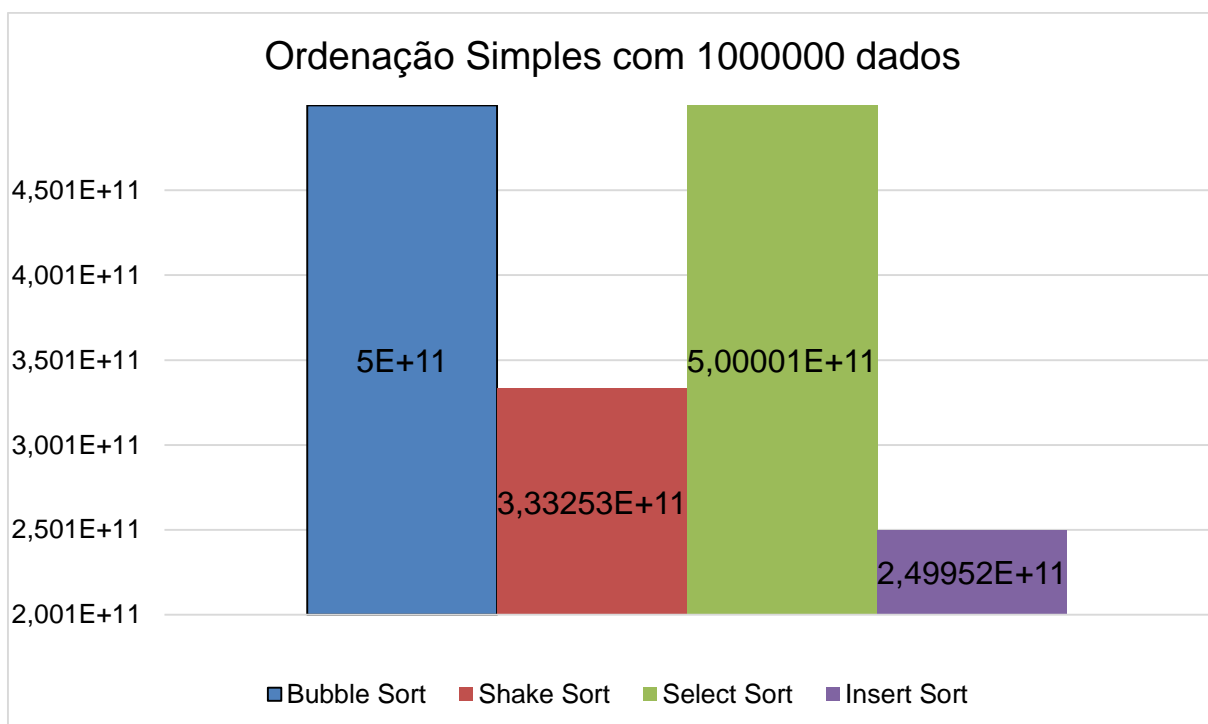
Fonte: Própria, 2016.

Gráfico 13 – Ordenação simples de quinhentos mil dados.



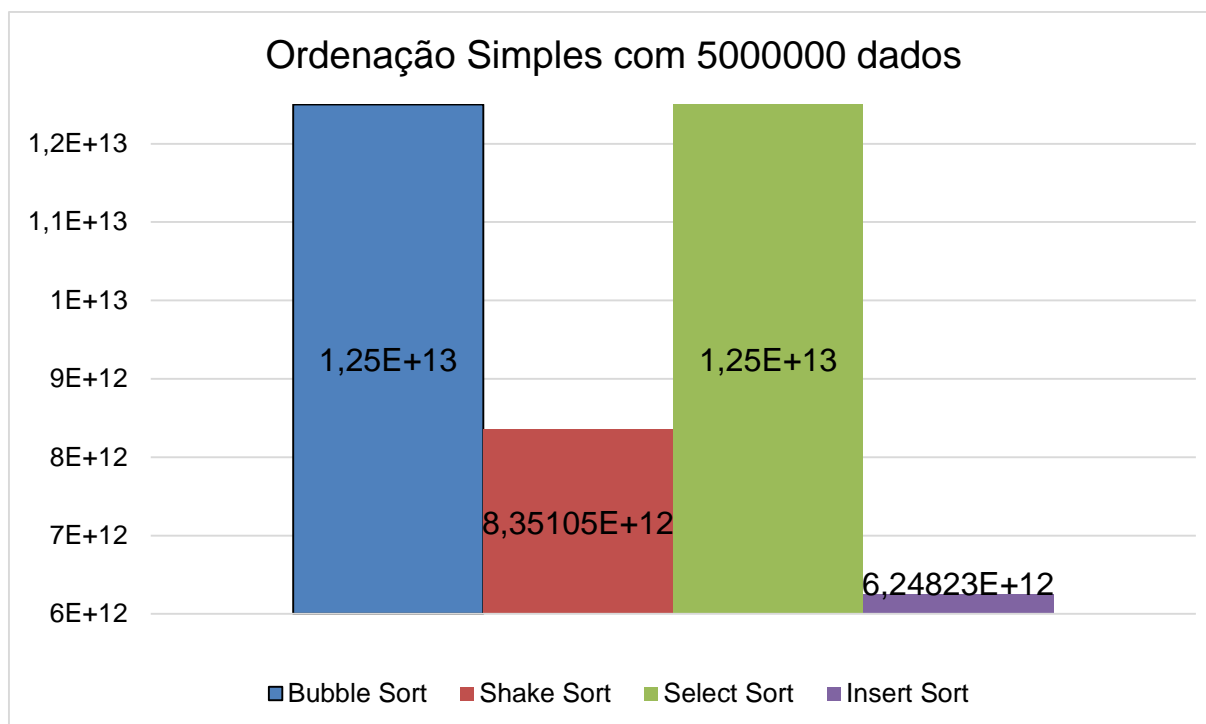
Fonte: Própria, 2016.

Gráfico 14 – Ordenação simples de um milhão de dados.



Fonte: Própria, 2016.

Gráfico 15 – Ordenação simples de cinco milhões de dados.



Fonte: Própria, 2016.

Basta olhar os gráficos e percebemos que o *Insert Sort* é, de longe, o melhor algoritmo de ordenação simples, mesmo executando em $O(N^2)$.

O algoritmo *Bubble Sort* é o menos eficiente entre os quatro, porque o algoritmo faz cerca de N^2/X comparações. Há menos trocas do que há comparações porque dois dados serão trocados apenas se precisarem ser. É necessário N^2/Y trocas. Mas como constantes não contam na notação *Big O*, podemos ignorar o x e o y e dizer que a ordenação é executada em $O(N^2)$.

O *Shake Sort* ele diminui o número de interação total, mas não diminui o número de troca e comparação do *Bubble Sort*. Ele continua na execução $O(N^2)$.

O algoritmo *Select Sort* possui três interações à mais em relação com o *Bubble Sort*, mas isso não quer dizer que ele seja mais lento. O algoritmo *Select Sort* reduz o número de trocas necessárias de $O(N^2)$ para $O(N)$, mas infelizmente o número de comparações permanece em $O(N^2)$. Contudo, ela é inquestionavelmente mais rápida porque há poucas trocas. Para valores menores de n , a ordenação pode, de fato, ser consideravelmente mais rápida, especialmente se os tempos de troca forem muito maiores que os tempos de comparação.

O *Insert Sort* é inquestionavelmente o mais rápido entre os quatro algoritmos simples. Ele ainda executa em $O(N^2)$, mas é cerca de duas vezes mais rápido que o *Bubble Sort* e um pouco mais rápido que o *Select Sort*. O número de cópias que o

Insert Sort realiza é aproximadamente o mesmo que o número de comparações. Contudo uma cópia não é tão demorada quanto uma troca, portanto, então para dados aleatórios esse algoritmo executa duas vezes mais rápido que o *Bubble Sort* e mais rápido que o *Select Sort*.

Para dados que já estejam ordenados ou quase ordenados, o *Insert Sort* é melhor. Quando os dados estão em ordem, a condição do laço *while* nunca será verdadeira, portanto, ela tornar-se uma instrução simples no laço externo. Se os dados estiverem quase ordenados, o *Insert Sort* executará em $O(N)$, o que o torna uma maneira simples e eficiente de ordenar um arquivo que esteja apenas ligeiramente fora de ordem.

Não há razão para usar o *Bubble Sort*. Ele é prático apenas se a quantidade de dados for pequena e se você souber apenas ele de cabeça.

O *Select Sort* minimiza o número de trocas, mas o número de comparações ainda é alto. Essa ordenação é útil, segundo Lafore (2004, p. 95), quando a quantidade de dados for pequena e a troca de itens de dados consumir muito tempo em relação a compará-los.

O algoritmo *Insert Sort* é o mais versátil e é a melhor escolha na maioria das situações. O *Quick Sort* usufrui do *Insert Sort* quando a quantidade de dados desordenados é pequena.

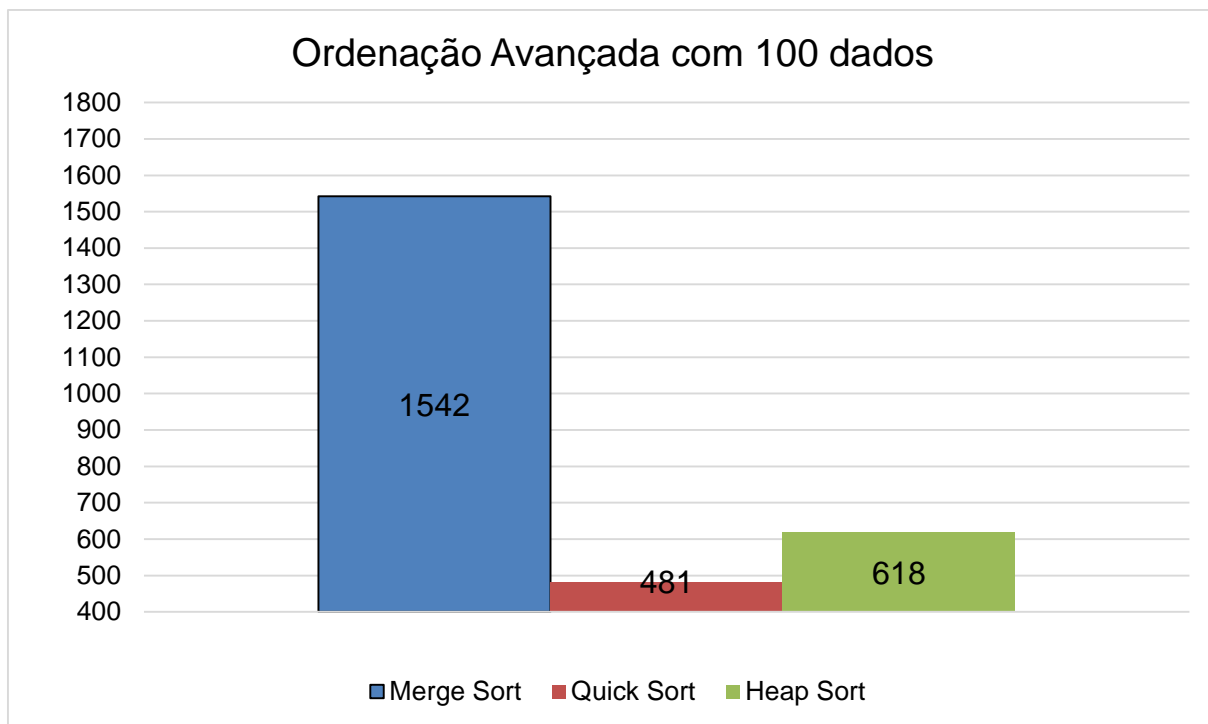
4.2.2 Ordenação avançada

Os algoritmos que farão parte da análise avançada são o *Merge Sort*, *Quick Sort* e *Heap Sort*, três ao todo.

Nesta etapa estaremos evidenciando o motivo do *Quick Sort* ser o mais rápido, na maioria dos cenários, entre os três avançados.

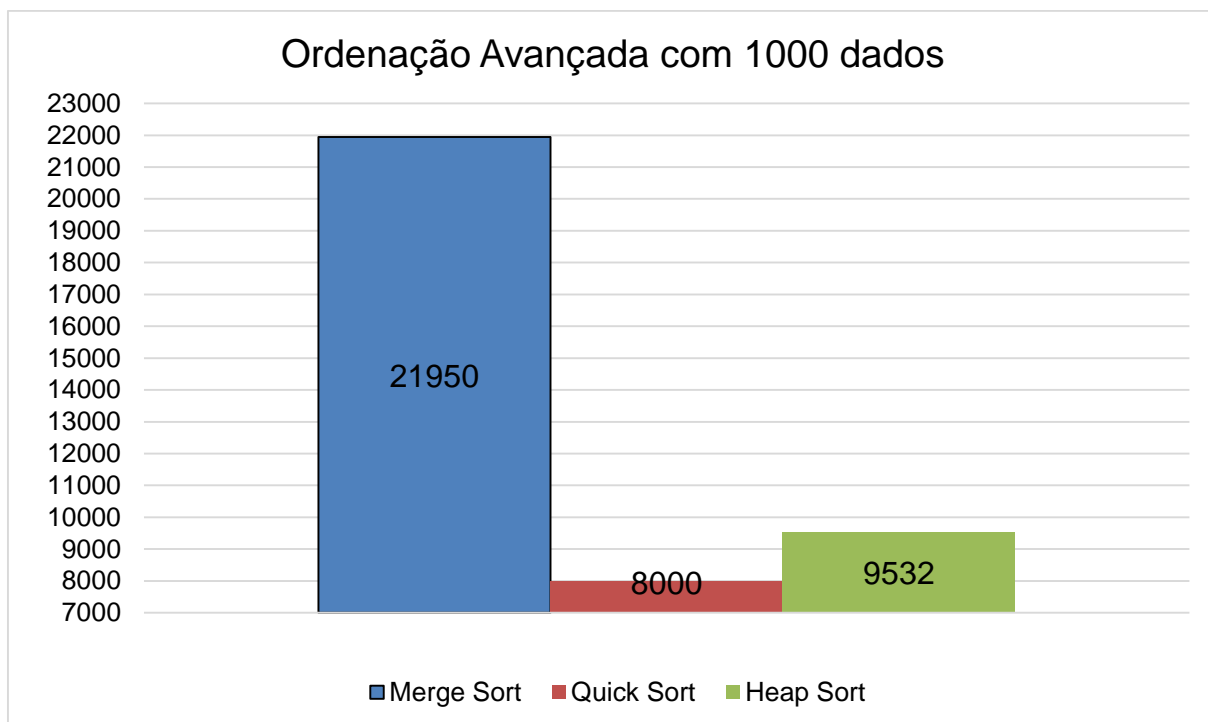
Os gráficos abaixo evidenciam o desempenho de cada algoritmo avançado em todos os cenários.

Gráfico 16 – Ordenação avançada de cem dados.



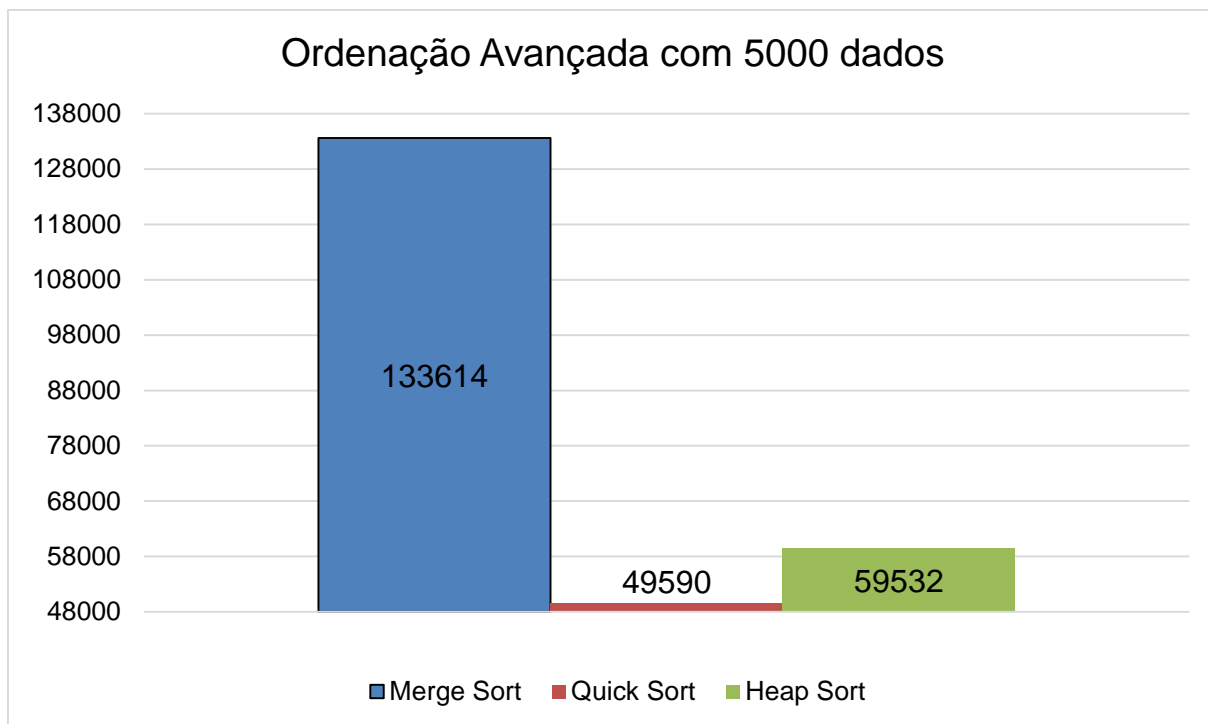
Fonte: Própria, 2016.

Gráfico 17 – Ordenação avançada de mil dados.



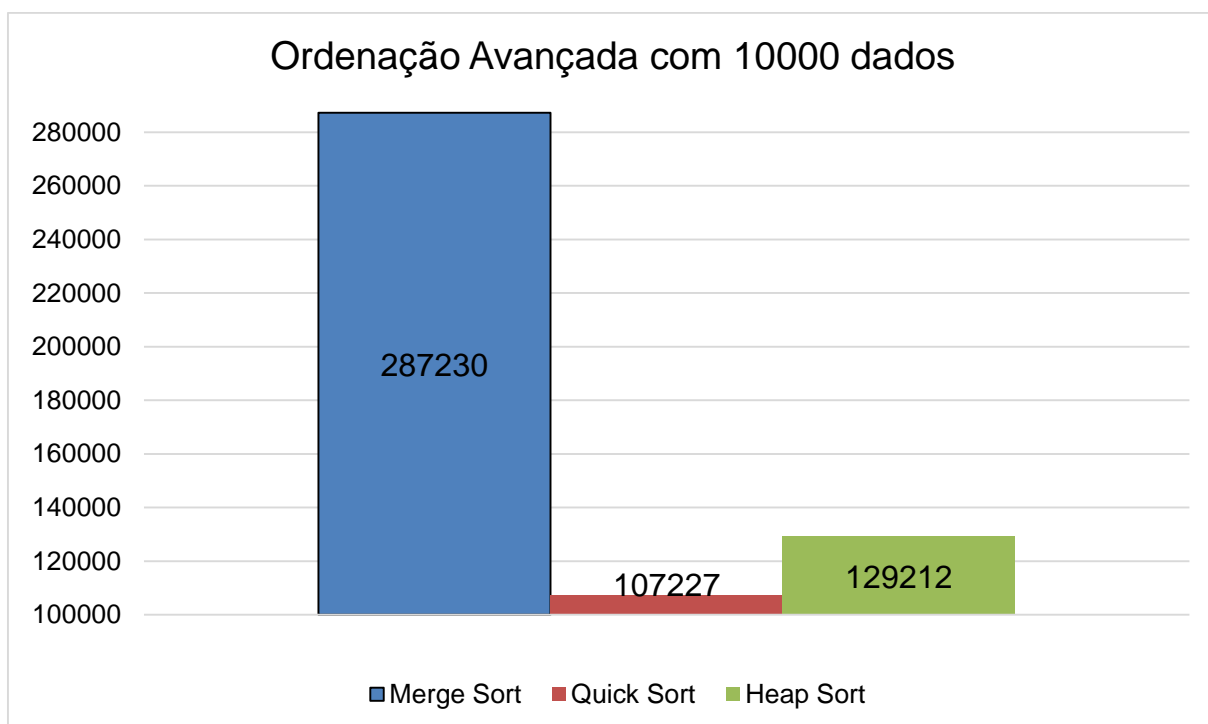
Fonte: Própria, 2016.

Gráfico 18 – Ordenação avançada de cinco mil dados.



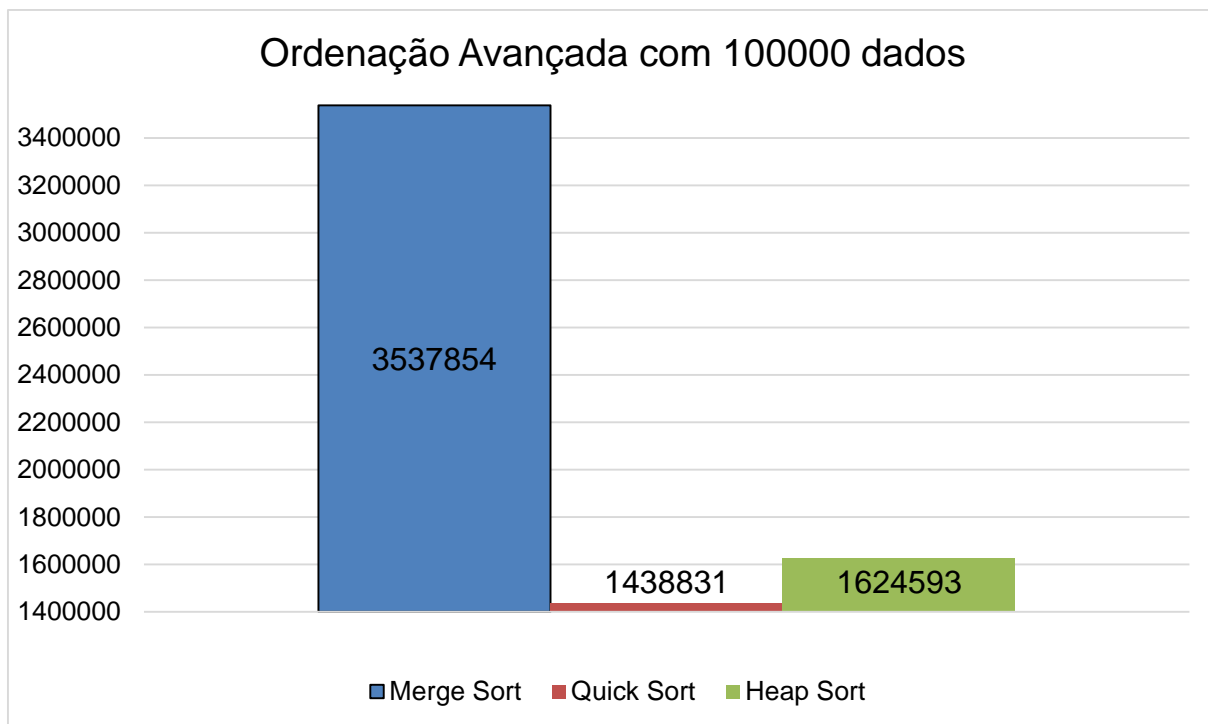
Fonte: Própria, 2016.

Gráfico 19 – Ordenação avançada de dez mil dados.



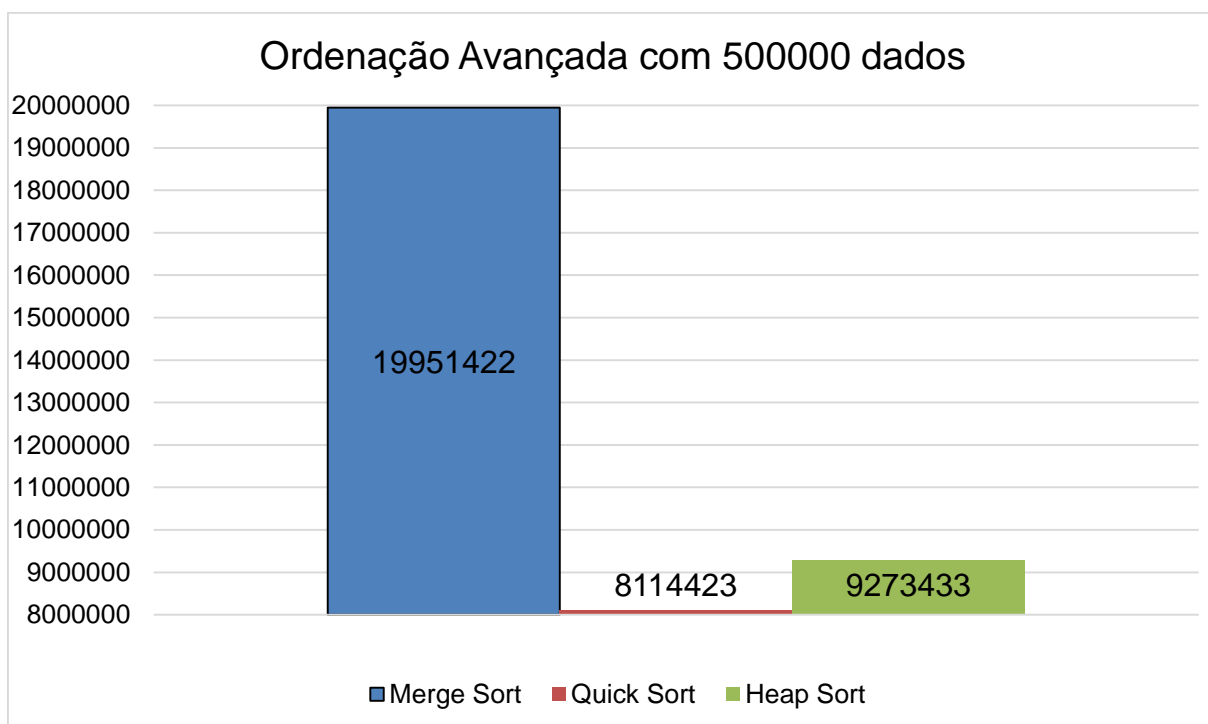
Fonte: Própria, 2016.

Gráfico 20 – Ordenação avançada de cem mil dados.



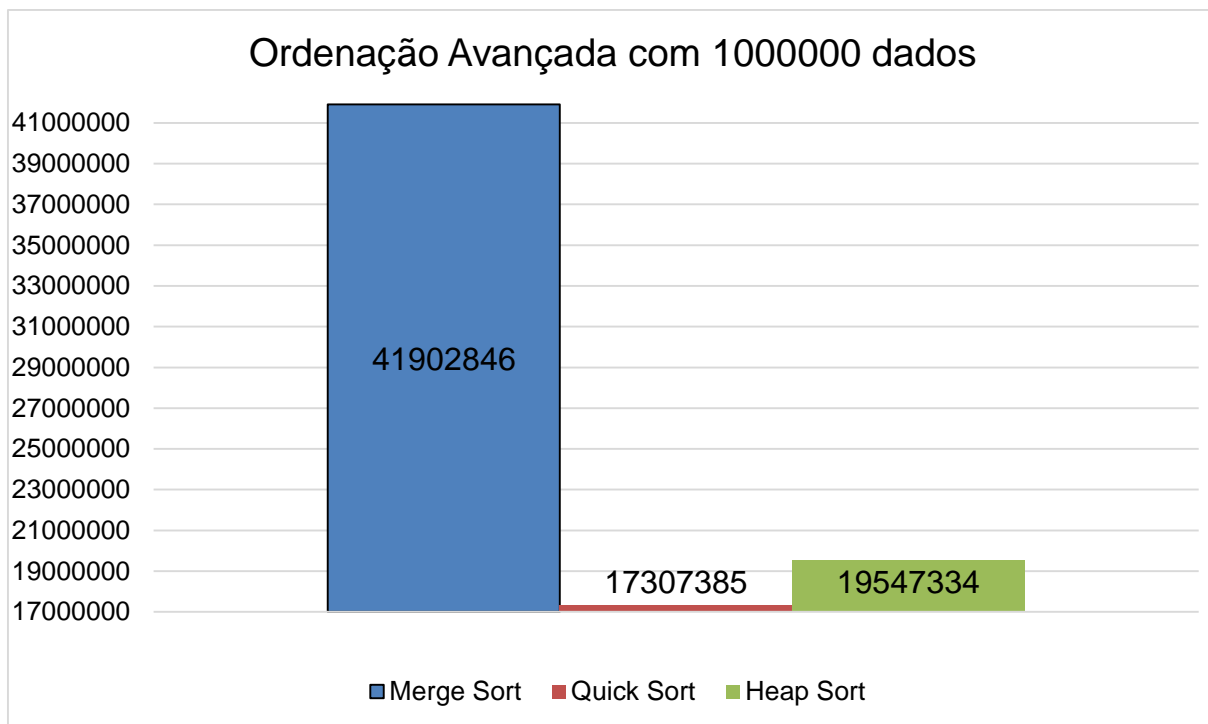
Fonte: Própria, 2016.

Gráfico 21 – Ordenação avançada de quinhentos mil dados.



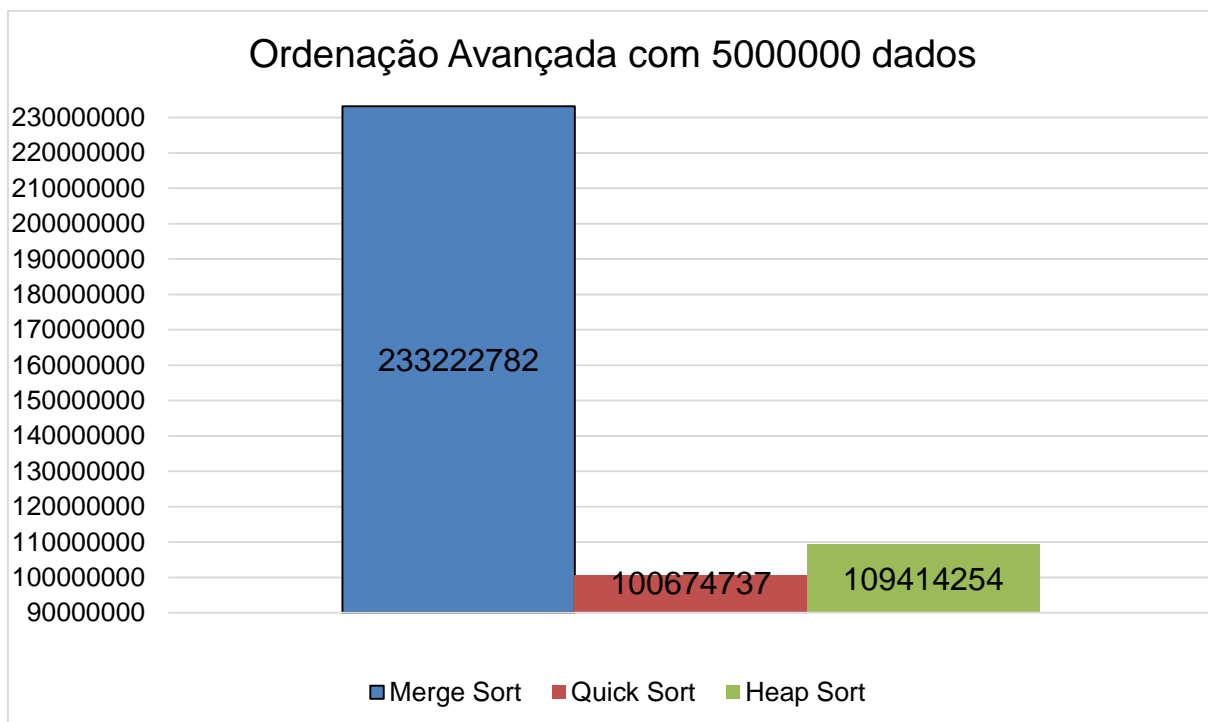
Fonte: Própria, 2016.

Gráfico 22 – Ordenação avançada de um milhão de dados.



Fonte: Própria, 2016.

Gráfico 23 – Ordenação avançada de cinco milhões de dados.



Fonte: Própria, 2016.

Basta olhar os gráficos e percebemos que o *Quick Sort* é, de longe, o melhor algoritmo de ordenação simples, mesmo executando em $O(n \log n)$.

O algoritmo *Merge Sort* é o menos eficiente entre os três, porque o algoritmo faz cerca de $n * \log n$ comparações e necessita de um vetor adicional na memória, igual em tamanho àquele sendo ordenado. Se o vetor original mal couber na memória, a ordenação por intercalação não funcionará.

O algoritmo *Heap Sort* ele é o único do conjunto que não utiliza a técnica de recursividade, mas isso não quer dizer que ele seja o mais lento. O algoritmo *Heap Sort* possui dois laços que chamam uma função que opera em tempo $O(\log n)$ e cada um tem que ser aplicado n vezes, com isso a ordenação inteira requer $O(n \log n)$, o mesmo que o *Merge Sort* e *Quick Sort*. Porém, ele não é tão rápido quanto o *Quick Sort*, em parte porque há mais operadores no laço *while* interno do que no laço interno do *Quick Sort*. Embora possa ser ligeiramente mais lento que o *Quick Sort*, o *Heap* possui uma vantagem sobre o *Quick Sort*, segundo Lafore (2004, p. 557), ele é menos suscetível à distribuição inicial de dados. Certas organizações de valores-chaves podem reduzir o *Quick Sort* para $O(N^2)$, ao passo que a ordenação *Heap Sort* é executada em $O(n \log n)$ independentemente como os dados estejam distribuídos.

O *Quick Sort* é inquestionavelmente o mais rápido entre os três algoritmos avançado. Ele executa em $O(n \log n)$, mas é cerca de duas vezes mais rápido que o *Merge Sort* e um pouco mais rápido que o *Heap Sort*. O melhor caso é $O(\log n)$, mas isso só ocorre em situações onde cada *subvetor* é particionado exatamente na metade e o pivô já esteja na sua posição final.

A escolha do pivô influencia fortemente o algoritmo *Quick Sort*. Se escolhermos um pivô mais para extremidade, poderemos torná-lo $O(N^2)$, mas se escolhermos um pivô mais centralizados tornamos ele $O(n \log n)$ no caso médio e no melhor caso podemos atingir $O(\log n)$.

4.3 Quando usar o quê

Após analisarmos e determinamos que o *Insert Sort* é o mais eficiente dos algoritmos simples e o *Quick Sort* o mais eficiente de todos. Agora estaremos evidenciando quando usar cada algoritmo.

Lafore (2004, p. 663) explica que vale a pena inicialmente tentar uma ordenação lenta, porém simples, como o algoritmo *Insert Sort*.

A ordenação *Insert* é boa para arquivos quase ordenados, operando mais ou menos em tempo $O(N)$ se não forem muitos os itens fora de lugar. Este é tipicamente o caso onde alguns itens novos sejam adicionados a um arquivo já ordenado.

Quando se deseja uma ordenação mais rápida, podemos usufruir do *Quick Sort*, em vez do *Merge Sort* que necessita de memória extra e do *Heap Sort* que necessita de estrutura de dados de heap e ambas são um pouco lentas que o *Quick Sort*.

Quick Sort será suspeito se houver um risco de que os dados não sejam aleatórios, caso em que o desempenho poderá deteriorar para $O(N^2)$. Para dados potencialmente não aleatórios, o *Heap Sort* será melhor. *Quick Sort* é também propenso a erros sutis se não for implementado corretamente. Pequenos erros de codificação poderão fazer com que ele funcione precariamente para certas organizações de dados. (LAFORE, 2004, p. 663).

O *Heap Sort* é uma boa escolha quando se deseja economizar no processamento, porque, segundo Lafore (2004, p. 262), na prática a abordagem recursiva prova ser ineficiente. Em tais casos, é útil transformar a abordagem recursiva em uma abordagem não recursiva. Tal transformação pode com frequência usar uma pilha.

5 CONCLUSÃO

O aspecto marcante da atual sociedade é a automatização de tarefas, e na ciência da computação há um processo de desenvolvimento simultâneo e interativo de máquinas e dos elementos lógicos que gerenciam a execução automática de uma tarefa.

Um computador é uma máquina que manipula informações. O estudo da ciência da computação inclui o exame da organização, manipulação e utilização destas informações num computador. Consequentemente, é muito importante compreendê-los.

As estruturas de dados, na maioria dos casos, adaptam as informações vistas no dia a dia para serem armazenadas e manipuladas nas máquinas. Elas diferem uma das outras pela disposição ou manipulação de seus dados e cada uma possui uma vantagem e desvantagem. O tipo de estrutura influencia na escolha do algoritmo para a solução do problema.

A ordenação é o processo de organizar um conjunto de informações segundo uma determinada ordem. Se o conjunto estiver ordenado é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo, a pesquisa binária. A ordenação é uma tarefa muito importante no processamento de dados e é feita para facilitar a pesquisa.

Uma das tarefas mais comuns no dia a dia é a pesquisa de informação. Mas, a pesquisa depende muito da forma como a informação está organizada. Se a informação não estiver organizada não temos alternativa, infelizmente, que não seja percorrer todo o arranjo de informação por ordem seja ela do princípio para o fim ou vice-versa, até encontrar a desejada. Mas, se a informação estiver ordenada por uma ordem, seja ela crescente ou decrescente no caso de informação numérica, ascendente ou descendente no caso de informação textual, a pesquisa pela informação seria mais simplificada.

Devido ao seu uso muito frequente, é importante ter a disposição algoritmos de ordenação eficientes tanto em termos de tempo como em termos de espaço.

O uso é tão frequente que no nosso dia-a-dia presenciamos uma ordenação e nem percebemos. Um exemplo são as fichas de funcionários de certo hospital, cada ficha tem o seu nível de prioridade e devem ser ordenadas conforme certo critério. O médico precisa ter acesso a uma pesquisa eficiente e rápida que possa retornar a

ficha desejada, porque muitas vezes a vida de uma pessoa depende de apenas uma informação e se essa informação não estiver à disposição em segundos, uma vida será perdida.

Outro exemplo é o de uma fila em algum estabelecimento, como por exemplo, um banco, onde os prioritários são os idosos, gestantes e deficientes. Inconscientemente são realizadas uma ordenação e uma pesquisa linear para realizar tal tarefa.

Voltando ao caso do médico, para que ele tenha uma pesquisa eficiente as fichas precisam estar ordenadas em poucos segundos, porque toda hora chega nova ficha, para que seja realizada uma pesquisa binária, onde a mesma é superior à uma pesquisa linear, a que usamos no caso de uma fila bancária.

Este trabalho levantou os conceitos, aplicações e usos dos algoritmos de ordenação, mas especificamente, estudando os *Bubble Sort*, *Shake Sort*, *Select Sort*, *Insert Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*, existem n algoritmos de ordenação. Para tanto, foi feita uma revisão sobre ordenação, pesquisa de dados, estrutura de dados, tipos de dados, algoritmos mais conhecidos e programação *Java*.

A partir dos sete algoritmos, citados anteriormente, foi realizado uma banca de teste com oito cenários distintos (cem dados, mil dados, cinco mil dados, dez mil dados, cem mil dados, quinhentos mil dados, um milhão de dados e cinco milhões de dados) em um programa desenvolvido na linguagem *Java*, demonstrando a capacidade de cada algoritmo em cada cenário.

Este estudo envolveu análise de algumas obras publicadas por estudiosos especialistas, comparou também o desempenho de cada algoritmo em cenários distintos.

Os testes demonstraram que uma análise realizada com base no tempo é totalmente equivocada e insegura. Durante os testes ocorreu o problema *Out of Memory*, no cenário de cinco milhões de dados, em máquinas menos robustas das quatro utilizadas no teste. Se a análise fosse calculada pelo tempo, este problema já incapacitaria o teste de evidenciar o desempenho de cada algoritmo em um cenário grande.

Os testes envolveram a análise pela interação dos algoritmos, porque independentemente da máquina os números foram, basicamente, os mesmos, mas os tempos de ordenação foram totalmente distintos, ocorrendo ordenação que durou

vinte e seis horas e outra que durou quarenta e oito horas, mas ambas resultaram no mesmo número de interação.

Após a conclusão do programa, foi possível perceber que é muito importante escolher uma linguagem de programação eficiente, já que há um processamento considerado “robusto” que pode facilmente apresentar erros futuros. Temos como exemplo a técnica *Bubble Sort*, o consumo de RAM para ordenar os dados resultou em um grande problema em máquinas menos preparadas. A escolha da linguagem também afeta a interação máquina e humano, por haver um processo significativo para o homem conseguir copiar, enviar e atualizar os dados para mais tarde usar o conjunto para futuras ordenações. Tudo isso ficou mais interativo e prático com uma interface gráfica.

Diante de tudo que foi apresentado, conclui-se que a ordenação é um conceito totalmente necessário e importante nos meios de manipulação de informação que utilizamos hoje, já que há um crescente aumento na busca e atualização das mesmas. Dependendo do ambiente, algumas técnicas de ordenação se sobressaem em relação a outras, mas se o proprietário desejar ter uma velocidade de ordenação elevada terá que investir no ambiente em que a técnica de ordenação irá atuar. Acredita-se que com o contínuo avanço da tecnologia, as ordenações atuais tendem a se tornarem obsoletas, como o *Bubble Sort* e *Shake Sort*.

Não existe a melhor técnica de ordenação, cada uma será excelente no ambiente certo. Não é porque o *Quick Sort* demonstrou o melhor desempenho que ele seja o melhor de todas em qualquer cenário, vale ressaltar que o risco dos dados não serem aleatórios e a escolha equivocada do pivô o deixa como $O(N^2)$, igualando com o *Bubble Sort*, *Shake Sort*, *Insert Sort* e *Select Sort*, e vimos que $O(N^2)$ é o pior desempenho de todos, consumindo muita memória RAM e levando mais tempos para ordenar dados em excesso. Então vale uma análise detalhada para determinar qual algoritmo será implementado.

REFERÊNCIA BIBLIOGRÁFICA

TENENBAUM, A. A; LANGSAM, Y; AUGENSTEIN, M. J. **Estrutura de dados usando C**. 1. ed. São Paulo: Makron Books, 1995. 884 p.

STALLINGS, W. **Arquitetura e organização de computadores**. 5. ed. São Paulo: Pearson Prentice Hall, 2002. 786 p.

LAFORE, R. **Estruturas de dados & algoritmos em Java**. 2. ed. Rio de Janeiro: Ciência Moderna, 2004. 702 p.

SIERRA, K; BATES, B. **Use a cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2005. 470 p.

GOODRICH, M. T; TAMASSIA, R. **Estrutura de dados e Algoritmos em Java**. 4. ed. Porto Alegre: Bookman, 2007. 600 p.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. 1. ed. Rio de Janeiro: Brasport Livros e Multimídia, 2008. 152 p.

HORSTMANN, C. **Conceitos de computação com Java**. 5. ed. São Paulo: Bookman, 2009. 720 p.

DEITEL, H. M; DEITEL, P. J. **Java como programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010. 391 p.

ASCENCIO, A. F. G; ARAÚJO, G. S. **Estrutura de dados: algoritmos, análise da complexidade e implementações em Java e C/C++**. 1. ed. São Paulo: Pearson Prentice Hall, 2010. 433 p.

FÁVERO, E. B. **Organização e arquitetura de computadores**. 1. ed. Paraná: Universidade Tecnológica Federal do Paraná, 2011. 114 p.

JUNIOR, A. V; CENTRO UNIVERSITÁRIO DE MARINGÁ. **Fundamentos e arquitetura de computadores**. 1. ed. Paraná: Universidade de Maringá, 2012. 153 p.

SANTOS, N. M; JÚNIOR, G. N. S; NETO, O. P. S. **Estrutura de dados**. 1. Ed. Piauí: Instituto Federal de Educação, Ciência e Tecnologia do Piauí, 2013. 150 p.

SANTOS, R. R. **Programação de computadores Java**. 2. ed. Rio de Janeiro: Novaterra Editora, 2014. 1437 p.

6 FICHAS DE ATIVIDADES PRÁTICAS SUPERVISIONADAS

Ficha de cada membro da APS.

Imagem 37: Gabriel de Almeida Batista.

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS				
Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)				
NOME: <u>Galvão de Almeida Batista</u>				
RA: <u>C445BF-1</u>		CURSO: <u>Ciências da Computação</u>		
CAMPUS: <u>Tatuí</u>		SEMESTRE: <u>3º Semestre</u>		TURNO: <u>Manhã</u>

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
06/06	Colêto de dados	4	Galvão Batista	
07/06	Colêto de dados	4	Galvão Batista	
08/06	Colêto de dados	4	Galvão Batista	
09/06	Colêto de dados	4	Galvão Batista	
10/06	Colêto de dados	4	Galvão Batista	
11/06	Colêto de dados	4	Galvão Batista	
12/06	Colêto de dados	4	Galvão Batista	
13/06	Colêto de dados	4	Galvão Batista	
14/06	Diversões	4	Galvão Batista	
15/06	Diversões	4	Galvão Batista	
16/06	Diversões	4	Galvão Batista	
17/06	Diversões	4	Galvão Batista	
18/06	Diversões	4	Galvão Batista	
19/06	Diversões	4	Galvão Batista	
20/06	Diversões	4	Galvão Batista	
21/06	Diversões	4	Galvão Batista	
22/06	Diversões	4	Galvão Batista	
23/06	Diversões	4	Galvão Batista	
24/06	Diversões	4	Galvão Batista	
25/06	Diversões	4	Galvão Batista	
26/06	Diversões	4	Galvão Batista	
27/06	Diversões	4	Galvão Batista	
28/06	Diversões	4	Galvão Batista	
29/06	Diversões	4	Galvão Batista	
30/06	Diversões	4	Galvão Batista	

TOTAL DE HORAS: 100

Fonte: Própria, 2016.

Imagem 38: Felipe da Silva Borges Neves.

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME: Felipe da Silva Borges Neves

RA: 649770-3

CAMPUS: Tutuapé

CURSO: Ciência da Computação

SEMESTRE: 3º

TURNO: manhã

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
06/05	Colita de baba	1	Felipe Borges	
07/05	Colita de baba	1	Felipe Borges	
08/05	Colita de baba	1	Felipe Borges	
09/05	Colita de baba	1	Felipe Borges	
10/05	Colita de baba	1	Felipe Borges	
11/05	Colita de baba	1	Felipe Borges	
12/05	Colita de baba	1	Felipe Borges	
13/05	Colita de baba	1	Felipe Borges	
14/05	Desenho	1	Felipe Borges	
15/05	Desenho	1	Felipe Borges	
16/05	Desenho	1	Felipe Borges	
17/05	Desenho	1	Felipe Borges	
18/05	Desenho	1	Felipe Borges	
19/05	Desenho	1	Felipe Borges	
20/05	Desenho	1	Felipe Borges	
21/05	Desenho	1	Felipe Borges	
22/05	Desenho	1	Felipe Borges	
23/05	Desenho	1	Felipe Borges	
24/05	Desenho	1	Felipe Borges	
25/05	Desenho	1	Felipe Borges	
26/05	Desenho	1	Felipe Borges	
27/05	Desenho	1	Felipe Borges	
28/05	Desenho	1	Felipe Borges	
29/05	Desenho	1	Felipe Borges	
30/05	Desenho	1	Felipe Borges	

TOTAL DE HORAS: 100

Fonte: Própria, 2016.

Imagem 39: César Magnun Oliveira.

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME:

César Magnun Oliveira

RA:

CG13459

CURSO:

ciências da computação

CAMPUS:

Tatuapé

SEMESTRE:

3º

TURNO:

manhã

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
08/09	Calculo de dados	1	Cesar	
07/09	Calculo de dados	1	Cesar	
06/09	Calculo de dados	1	Cesar	
09/09	Calculo de dados	1	Cesar	
10/09	Calculo de dados	1	Cesar	
11/09	Calculo de dados	1	Cesar	
12/09	Calculo de dados	1	Cesar	
13/09	Calculo de dados	1	Cesar	
14/09	Desenvolvimento	1	Cesar	
15/09	Desenvolvimento	1	Cesar	
16/09	Desenvolvimento	1	Cesar	
17/09	Desenvolvimento	1	Cesar	
18/09	Desenvolvimento	1	Cesar	
19/09	Desenvolvimento de software	1	Cesar	
20/09	Desenvolvimento de software	1	Cesar	
21/09	Desenvolvimento de software	1	Cesar	
22/09	Desenvolvimento de software	1	Cesar	
23/09	Desenvolvimento de software	1	Cesar	
24/09	Desenvolvimento de software	1	Cesar	
25/09	Desenvolvimento	1	Cesar	
26/09	Desenvolvimento	1	Cesar	
27/09	Desenvolvimento	1	Cesar	
28/09	Revisão Geral	1	Cesar	
29/09	Revisão Geral	1	Cesar	
30/09	Revisão Geral	1	Cesar	

TOTAL DE HORAS: 100

Imagem 40: José Vitor Zanoni da Costa.

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME: José Vitor Zanoni da Costa

RA: C60EF1-0CURSO: Ciência da Computação

CAMPUS: MatuariSEMESTRE: 3ºTURNO: Manhã

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
06/05	Calda de dados	1	José Vitor	
07/05	Calda de dados	1	José Vitor	
08/05	Calda de dados	1	José Vitor	
09/05	Calda de dados	1	José Vitor	
10/05	Calda de dados	1	José Vitor	
11/05	Calda de dados	1	José Vitor	
12/05	Calda de dados	1	José Vitor	
13/05	Calda de dados	1	José Vitor	
14/05	Dinâmica	1	José Vitor	
15/05	Dinâmica	1	José Vitor	
16/05	Dinâmica	1	José Vitor	
17/05	Dinâmica	1	José Vitor	
18/05	Dinâmica	1	José Vitor	
19/05	Dinâmica	1	José Vitor	
20/05	Desenvolvimento de software	1	José Vitor	
21/05	Desenvolvimento de software	1	José Vitor	
22/05	Desenvolvimento de software	1	José Vitor	
23/05	Desenvolvimento de software	1	José Vitor	
24/05	Desenvolvimento de software	1	José Vitor	
25/05	Dinâmica	1	José Vitor	
26/05	Dinâmica	1	José Vitor	
27/05	Dinâmica	1	José Vitor	
28/05	Revisão Geral	1	José Vitor	
29/05	Revisão Geral	1	José Vitor	
30/05	Revisão Geral	1	José Vitor	

TOTAL DE HORAS: 100

ANEXOS**ANEXO A – CÓDIGO DA CLASSE PRINCIPAL.JAVA**

```
package br.unip.cc.bnmc;

import java.awt.EventQueue;

public class Principal
{
    @SuppressWarnings("unused")
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                try
                {
                    JMenuFrame frame = new JMenuFrame();
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

ANEXO B – CÓDIGO DA CLASSE ARQUIVO.JAVA

```
package br.unip.cc.bnmc;

import javax.swing.filechooser.FileNameExtensionFilter;
import javax.swing.JFileChooser;

public class Arquivo
{
    private String caminho;

    public String getCaminho()
    {
        return caminho;
    }

    public void setCaminho(String caminho)
    {
        this.caminho = caminho;
    }

    public void buscar() throws EDadoInvalido
    {
        //Implementa os tipos de arquivos
        FileNameExtensionFilter fileNameExtensionFilter =
            new FileNameExtensionFilter("png", "jpg", "gif",
            "tiff", "bmp", "jpeg");

        //Instanciando o selecionador de arquivos
        JFileChooser fc = new JFileChooser();

        //Adicionando os arquivos que poderam ser selecionados
        fc.setFileFilter(fileNameExtensionFilter);
    }
}
```



```

//Nome da tela localizadora de arquivos
fc.setDialogTitle("Escolha uma imagem");

//Recebe uma resposta da janela caso algum evento do
localizador de arquivo seja acessado
int resposta = fc.showOpenDialog(null);

//Verifica se a resposta é igual a ok
if (resposta == JFileChooser.CANCEL_OPTION)
    ;

else
    if(resposta == JFileChooser.APPROVE_OPTION)

setCaminho(fc.getSelectedFile().getAbsolutePath());
    }
}

```

ANEXO C – CÓDIGO DA CLASSE BUBBLESORT.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class BubbleSort extends Sort
{
    private static int out;
    private static int in;
    private static long count;

    public static void bubbleSortIma()
    {
        for(out = getnElemIma() - 1; out >= 1; out--)
        {

```

```

        for(in = 0; in < out; in++)
        {
            if(getArraylma(in).getWidth() > getArraylma(in +
1).getWidth())

                swaplma(in, in + 1);

            ++count;
        }

        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Bubble Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}

public static void bubbleSortInt()
{
    for(out = getnElemsInt() - 1; out > 1; out--)
    {
        for(in = 0; in < out; in++)
        {
            if(getArrayInt(in) > getArrayInt(in + 1))
                swapInt(in, in + 1);

            ++count;
        }

        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Bubble Sort", JOptionPane.INFORMATION_MESSAGE);

```

```
        count = 0;
    }
}
```

ANEXO D – CÓDIGO DA CLASSE EDADOINVALIDO.JAVA

```
package br.unip.cc.bnmc;

public final class EDadoInvalido extends Exception
{
    private static final long serialVersionUID = 1L;

    public EDadoInvalido()
    {
        super("Dado inválido!");
    }

    public EDadoInvalido(String mensagem)
    {
        super("\nDado inválido!" + mensagem);
    }
}
```

ANEXO E – CÓDIGO DA CLASSE HEAPSORT.JAVA

```
package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class HeapSort extends Sort
{
    private static int out;
    private static int in;
    private static long count;
```

```

public static void heapSortlma()
{
    int p;

    out = getnElemslma()-1;

    for(p = out / 2; p >= 1; --p)
    {
        peneiralma(p, out);
        ++count;
    }

    for(in = out; in >= 2; --in)
    {
        swaplma(1, in);
        peneiralma(1, in - 1);
        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Heap Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}

public static void peneiralma(int p, int m)
{
    int x = getArraylma(p).getWidth() ;

    while(2 * p <= m)
    {
        int f = 2 * p;
        if(f < m && getArraylma(f).getWidth() < getArraylma(f +
1).getWidth() )

```

```

        ++f;

        if(x >= getArraylma(f).getWidth() )
            break;

        setArraylma(getArraylma(f), p);
        p = f;
        ++count;
    }

    setArrayInt(x, p);
}

public static void heapSortInt()
{
    int p;

    out = getnElemsInt()-1;

    for(p = out / 2; p >= 1; --p)
    {
        peneiraInt(p, out);
        ++count;
    }

    for(in = out; in >= 2; --in)
    {
        swapInt(1, in);
        peneiraInt(1, in - 1);
        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Heap Sort", JOptionPane.INFORMATION_MESSAGE);

```

```

        count = 0;
    }

    public static void peneiraInt(int p, int m)
    {
        int x = getArrayInt(p);

        while(2 * p <= m)
        {
            int f = 2 * p;
            if(f < m && getArrayInt(f) < getArrayInt(f + 1))
                ++f;

            if(x >= getArrayInt(f))
                break;

            setArrayInt(getArrayInt(f), p);
            p = f;
            ++count;
        }

        setArrayInt(x, p);
    }
}

```

ANEXO F – CÓDIGO DA CLASSE IMAGEM.JAVA

```

package br.unip.cc.bnmc;

import java.awt.Image;
import javax.swing.ImageIcon;;

public class Imagem
{

```

```
private ImageIcon imagemIcon;  
private Image image;  
private String nome;  
private int width;  
private int height;  
  
public Imagem()  
{  
    this.imagemIcon = null;  
    this.image = null;  
    this.nome = null;  
    this.width = 0;  
    this.height = 0;  
}  
  
public Imagem(String caminho)  
{  
    this.imagemIcon = null;  
    this.image = null;  
    this.nome = null;  
    this.width = 0;  
    this.height = 0;  
    this.imagemIcon = new ImageIcon(caminho);  
    this.image = imagemIcon.getImage();  
    this.width = image.getWidth(null);  
    this.height = image.getHeight(null);  
}  
  
public void setNome(String nome)  
{  
    this.nome = nome;  
}  
  
public String getNome()
```

```

    {
        return nome;
    }

    public int getWidth()
    {
        return width;
    }

    public int getHeight()
    {
        return height;
    }
}

```

ANEXO G – CÓDIGO DA CLASSE IMAGEMPADRAO.JAVA

```

package br.unip.cc.bnmc;

public class ImagemPadrao
{
    private static Imagem imagem;

    public static void insere()
    {
        for(int i = 1; i <= 480; i++)
        {
            imagem = new Imagem("res\\apsJAVA (" + i + ").jpg");

            if(i > 0 && i < 10)
                Insere.insere(imagem, "APS JAVA (0000" + i + ")");

            else
                if(i > 9 && i < 100)

```



```

        Insere.insere(imagem, "APS JAVA (000" + i + ")");

        else
            Insere.insere(imagem, "APS JAVA (00" + i + ")");
    }
}
}

```

ANEXO H – CÓDIGO DA CLASSE INSERE.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class Insere
{
    private static Sort sortNew = new Sort();
    private static Arquivo fileNew;

    public Insere()
    {
        sortNew = null;
        fileNew = null;
    }

    public static void insere(Imagem nova)
    {
        String nome = JOptionPane.showInputDialog(null, "Digite o
nome da imagem", "Nome", JOptionPane.PLAIN_MESSAGE);

        if(nome != null)
            sortNew.insert(nova, nome);
    }
}

```

```

public static void insere(int value)
{
    sortNew.insert(value);
}

public static void insere(Imagem nova, String nome)
{
    sortNew.insert(nova, nome);
}

@SuppressWarnings("static-access")
public void display()
{
    for(int i = 0; i < 5; i++)
        System.out.println(sortNew.getArrayIma(i).getWidth());
}

public static void start()
{
    try
    {
        fileNew = new Arquivo();
        fileNew.buscar();

        Imagem nova = new Imagem(fileNew.getCaminho());

        if(!fileNew.getCaminho().equals(null))
            insere(nova);
    }

    catch(NullPointerException npe)
    {
        ;
    }
}

```

```

        catch (EDadoInvalido e)
        {
            ;
        }
    }
}

```

ANEXO I – CÓDIGO DA CLASSE INSERTSORT.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class InsertSort extends Sort
{
    private static int in;
    private static int out;
    private static long count;

    public static void insertSortIma()
    {
        for(out = 1; out < getnElemIma(); out++)
        {
            Imagem temp = getArrayIma(out);
            in = out;

            while(in > 0 && getArrayIma(in - 1).getWidth() >=
temp.getWidth())
            {
                setArrayIma(getArrayIma(in-1), in);
                --in;
                ++count;
            }
        }
    }
}

```

```

        setArrayInt(temp, in);
        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Insert Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}

public static void insertSortInt()
{
    for(out = 1; out < getnElemsInt(); out++)
    {
        int temp = getArrayInt(out);
        in = out;

        while(in > 0 && getArrayInt(in - 1) >= temp)
        {
            setArrayInt(getArrayInt(in-1), in);
            --in;
            ++count;
        }

        setArrayInt(temp, in);
        ++count;
    }

    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Insert Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}
}

```

ANEXO J – CÓDIGO DA CLASSE JINSEREFRAFRAME.JAVA

```
package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Font;
import javax.swing.JLabel;
import javax.swing.JList;

import java.awt.Color;
import javax.swing.JScrollPane;
import java.awt.Toolkit;

@SuppressWarnings("serial")
public class JInsereFrame extends JFrame
{
    private JPanel contentPane;
    private JLabel lblImagensCarregadasPelo;
    @SuppressWarnings("rawtypes")
    private JList listImage;
    private JScrollPane scrollPane;
    private JButton btnCancelar;
    private JButton btnContinuar;
    private JButton btnSelecionarArquivo;

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public JInsereFrame()
    {
```

```

        setIconImage(Toolkit.getDefaultToolkit().getImage(JInsereFrame.class.getResource("/javax/swing/plaf/metal/icons/ocean/upFolder.gif")));
        setTitle("Insere uma Imagem");
        setVisible(true);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 579, 306);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        lblImagensCarregadasPelo = new JLabel("Lista padr\u00E3o de
imagens do programa");
        lblImagensCarregadasPelo.setForeground(Color.RED);
        lblImagensCarregadasPelo.setFont(new Font("Arial",
Font.BOLD, 22));
        lblImagensCarregadasPelo.setBounds(102, 11, 392, 23);
        contentPane.add(lblImagensCarregadasPelo);

        listImage = new JList(Sort.dadosIma());

        scrollPane = new JScrollPane(listImage);
        scrollPane.setBounds(10, 59, 553, 150);
        contentPane.add(scrollPane);

        btnCancelar = new JButton("Cancelar");
        btnCancelar.setFont(new Font("Arial", Font.BOLD, 14));
        btnCancelar.setBounds(10, 242, 121, 23);
        contentPane.add(btnCancelar);
        btnCancelar.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)

```

```

        {
            System.exit(0);
        }
    });

    btnContinuar = new JButton("Continuar");
    btnContinuar.setFont(new Font("Arial", Font.BOLD, 14));
    btnContinuar.setBounds(228, 242, 121, 23);
    contentPane.add(btnContinuar);
    btnContinuar.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            try
            {
                @SuppressWarnings("unused")
                JOrdenaFrame frame = new
JOrdenaFrame();

                setVisible(false);
            }

            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    });

    btnSeleccionarArquivo = new JButton("Carregar");
    btnSeleccionarArquivo.setFont(new Font("Arial", Font.BOLD, 14));
    btnSeleccionarArquivo.setBounds(442, 242, 121, 23);
    contentPane.add(btnSeleccionarArquivo);
    btnSeleccionarArquivo.addActionListener(new ActionListener()
    {

```

```

        public void actionPerformed(ActionEvent arg0)
        {
            Insere.start();
        }
    });
}
}

```

ANEXO K – CÓDIGO DA CLASSE JINSERENUMBERFRAME.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.EmptyBorder;
import java.awt.Toolkit;
import javax.swing.JList;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Font;
import javax.swing.JLabel;
import java.awt.Color;
import java.awt.EventQueue;
import javax.swing.JOptionPane;

import javax.swing.JTextField;

@SuppressWarnings("serial")
public class JInsereNumberFrame extends JFrame
{
    private JPanel contentPane;
    private JTextField textFieldNumero;

```



```

@SuppressWarnings("rawtypes")
private JList listNumber;
private JButton btnCarregar;
private JButton btnContinuar;
private JButton btnSair;
private JLabel lblListaPadro;
private JButton btnAleatrio;
private JLabel lblNmero;
private JScrollPane scrollPane;
@SuppressWarnings("unused")
private JOrdenaNumberFrame frame;

@SuppressWarnings({ "rawtypes", "unchecked" })
public JInsereNumberFrame()
{
    setTitle("Insere um n\u00FAmero");
    setResizable(false);
    setVisible(true);

    setIconImage(Toolkit.getDefaultToolkit().getImage(JInsereNumberFrame.class
.getResource("/javax/swing/plaf/metal/icons/ocean/upFolder.gif")));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 450, 300);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);

    lblListaPadro = new JLabel("    Lista  padr\u00E3o  de
n\u00FAmeros do programa");
    lblListaPadro.setForeground(Color.RED);
    lblListaPadro.setFont(new Font("Arial", Font.BOLD, 22));
    lblListaPadro.setBounds(10, 11, 414, 26);
    contentPane.add(lblListaPadro);

```

```
listNumber = new JList(Sort.dadosNum());
```

```
scrollPane = new JScrollPane();
scrollPane.setBounds(10, 48, 274, 202);
scrollPane.setViewportView(listNumber);
contentPane.add(scrollPane);
```

```
lblNmero = new JLabel("N\u00FAmero");
lblNmero.setFont(new Font("Arial", Font.BOLD, 14));
lblNmero.setBounds(294, 69, 63, 18);
contentPane.add(lblNmero);
```

```
contentPane.add(getTxtNumber());
textFieldNumero.setColumns(10);
```

```
btnCarregar = new JButton("Carregar");
btnCarregar.setFont(new Font("Arial", Font.BOLD, 14));
btnCarregar.setBounds(294, 125, 130, 23);
contentPane.add(btnCarregar);
btnCarregar.addActionListener(new ActionListener()
{
```

```
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
```

```
Insere.insere(Integer.parseInt(getTxtNumber().getText()));
                getTxtNumber().setText("");
        }
    }
```

```
        catch(NumberFormatException nbe)
        {
```

```

                                JOptionPane.showMessageDialog(null,
"Digite apenas números", "NumberFormatException",
JOptionPane.ERROR_MESSAGE);

                                getTxtNumber().setText("");
                                }
                                }
    });

```

```

    btnAleatrio = new JButton("Aleatorio");
    btnAleatrio.setFont(new Font("Arial", Font.BOLD, 14));
    btnAleatrio.setBounds(294, 159, 130, 23);
    contentPane.add(btnAleatrio);
    btnAleatrio.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            Insere.insere((int)Math.ceil(Math.random()*999999999));
        }
    });

```

```

    btnContinuar = new JButton("Continuar");
    btnContinuar.setFont(new Font("Arial", Font.BOLD, 14));
    btnContinuar.setBounds(294, 193, 130, 23);
    contentPane.add(btnContinuar);
    btnContinuar.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            EventQueue.invokeLater(new Runnable()
            {
                public void run()
                {
                    try

```

```

        {
            frame = new
JOrdenaNumberFrame();

            setVisible(false);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

}

});

btnSair = new JButton("Sair");
btnSair.setFont(new Font("Arial", Font.BOLD, 14));
btnSair.setBounds(294, 227, 130, 23);
contentPane.add(btnSair);
btnSair.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        System.exit(0);
    }
});
}

private JTextField getTxtNumber()
{
    if(textFieldNumero == null)
    {
        textFieldNumero = new JTextField();
        textFieldNumero.setBounds(361, 67, 63, 20);
    }
}

```

```

        return textFieldNumero;
    }
}

```

ANEXO L – CÓDIGO DA CLASSE JMENUFRAFRAME.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JLabel;
import javax.swing.JOptionPane;

import java.awt.Color;
import java.awt.Font;
import java.awt.Toolkit;

@SuppressWarnings("serial")
public class JMenuFrame extends JFrame
{
    static boolean sobreVisible = false;
    private JPanel contentPane;
    private JButton btnInicia;
    private JButton btnSobre;
    private JButton btnSair;
    private JLabel lblTituloMenu;
    @SuppressWarnings("unused")
    private JInsereFrame frameIma;
    @SuppressWarnings("unused")

```

```

private JInsererNumberFrame frameNum;

public JMenuFrame()
{

    setIconImage(Toolkit.getDefaultToolkit().getImage(JMenuFrame.class.getResource("/com/sun/java/swing/plaf/windows/icons/HomeFolder.gif")));

    setVisible(true);
    setResizable(false);
    setTitle("Menu Principal");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 565, 277);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);


    lblTituloMenu = new JLabel("AN\u00C1LISE DE
PERFORMANCE DE ALGORITMO DE ORDENA\u00C7\u00C3O DE DADOS");
    lblTituloMenu.setForeground(Color.RED);
    lblTituloMenu.setFont(new Font("Arial", Font.BOLD, 14));
    lblTituloMenu.setBounds(20, 11, 539, 27);
    contentPane.add(lblTituloMenu);


    btnInicia = new JButton("INICIAR");
    btnInicia.setFont(new Font("Arial", Font.BOLD, 14));
    btnInicia.setForeground(Color.BLUE);
    btnInicia.setBounds(213, 49, 136, 48);
    contentPane.add(btnInicia);
    btnInicia.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            try

```

```

        {
            int tipo =
JOptionPane.showOptionDialog(null, "Escolha o tipo de manipulação", "Selecione",
0, JOptionPane.QUESTION_MESSAGE, null, new String[]{"Imagem", "Número",
"Cancelar"}, "Cancelar");

            if(tipo == -1)
                System.exit(0);

            else
                if(tipo == 0)
                {
                    ImagemPadrao.insere();
                    frameIma = new
JInsereFrame();

                }

            else
                if(tipo == 1)
                {
                    NumeroPadrao.insere();
                    frameNum = new
JInsereNumberFrame();

                }

            else
                System.exit(0);

            setVisible(false);
        }

        catch (Exception e)
        {
            e.printStackTrace();

```

```

        }
    }
});

btnSobre = new JButton("SOBRE");
btnSobre.setForeground(Color.BLUE);
btnSobre.setFont(new Font("Arial", Font.BOLD, 14));
btnSobre.setBounds(213, 118, 136, 48);
contentPane.add(btnSobre);
btnSobre.addActionListener(new ActionListener()
{
    @SuppressWarnings("unused")
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            JSobreFrame fSobre;

            if(!sobreVisible)
            {
                sobreVisible = true;
                fSobre = new JSobreFrame();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnSair = new JButton("SAIR");
btnSair.setForeground(Color.BLUE);
btnSair.setFont(new Font("Arial", Font.BOLD, 14));

```



```

        btnSair.setBounds(213, 187, 136, 48);
        contentPane.add(btnSair);
        btnSair.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)
            {
                System.exit(0);
            }
        });
    }
}

```

ANEXO M – CÓDIGO DA CLASSE JORDENADOFRAAME.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.EmptyBorder;
import java.awt.Toolkit;

@SuppressWarnings({ "serial" })
public class JOrdenadoFrame extends JFrame
{
    private JPanel contentPane;
    private JScrollPane scrollPane;
    @SuppressWarnings("rawtypes")
    private JList listImage;

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public JOrdenadoFrame(String titulo)
    {

```

```

        setIconImage(Toolkit.getDefaultToolkit().getImage(JOrdenadoFrame.class.getResource("/com/sun/java/swing/plaf/windows/icons/DetailsView.gif")));
        setTitle("Ordenado com " + titulo);
        setVisible(true);
        setResizable(false);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setBounds(100, 100, 437, 277);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        listImage = new JList(Sort.dadosIma());

        scrollPane = new JScrollPane(listImage);
        scrollPane.setBounds(30, 11, 356, 206);
        contentPane.add(scrollPane);
    }
}

```

ANEXO N – CÓDIGO DA CLASSE JORDENADONUMBERFRAME.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.EmptyBorder;
import java.awt.Toolkit;

@SuppressWarnings({ "serial" })
public class JOrdenadoNumberFrame extends JFrame

```

```

{
    private JPanel contentPane;
    private JScrollPane scrollPane;
    @SuppressWarnings("rawtypes")
    private JList listImage;

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public JOrdenadoNumberFrame(String titulo)
    {

        setIconImage(Toolkit.getDefaultToolkit().getImage(JOrdenadoFrame.class.getResource("/com/sun/java/swing/plaf/windows/icons/DetailView.gif")));
        setTitle("Ordenado com " + titulo);
        setVisible(true);
        setResizable(false);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setBounds(100, 100, 437, 277);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        listImage = new JList(Sort.dadosNum());

        scrollPane = new JScrollPane(listImage);
        scrollPane.setBounds(30, 11, 356, 206);
        contentPane.add(scrollPane);
    }
}

```

ANEXO O – CÓDIGO DA CLASSE JORDENAFRAME.JAVA

```
package br.unip.cc.bnmc;
```

```

import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.EmptyBorder;
import java.awt.Toolkit;

import javax.swing.JButton;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

@SuppressWarnings("serial")
public class JOrdenaFrame extends JFrame
{
    private JPanel contentPane;
    @SuppressWarnings("rawtypes")
    private JList listImage;
    private JScrollPane scrollPane;
    private JButton btnBubblesort;
    private JButton btnShakeSort;
    private JButton btnSelectsort;
    private JButton btnInsertsort;
    private JButton btnMergesort;
    private JButton btnShellsort;
    private JButton btnQuicksort;
    private JButton btnHeapsort;
    private JButton btnSair;
    @SuppressWarnings("unused")
    private JOrdenadoFrame frame;

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public JOrdenaFrame()
    {

```

```

        setIconImage(Toolkit.getDefaultToolkit().getImage(JOrdenaFrame.class.getResource(
source("/com/sun/java/swing/plaf/windows/icons/File.gif"))));
        setTitle("Ordenar Imagens");
        setVisible(true);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 683, 282);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        listImage = new JList(Sort.dadosIma());

        scrollPane = new JScrollPane(listImage);
        scrollPane.setBounds(10, 11, 397, 229);
        contentPane.add(scrollPane);

        btnBubblesort = new JButton("Bubble Sort");
        btnBubblesort.setFont(new Font("Arial", Font.BOLD, 14));
        btnBubblesort.setBounds(417, 11, 121, 23);
        contentPane.add(btnBubblesort);
        btnBubblesort.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)
            {
                try
                {
                    BubbleSort.bubbleSortIma();
                    frame = new JOrdenadoFrame("Bubble
Sort");
                }
            }
        });

```

```

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnSelectsort = new JButton("Select Sort");
btnSelectsort.setFont(new Font("Arial", Font.BOLD, 14));
btnSelectsort.setBounds(417, 47, 121, 23);
contentPane.add(btnSelectsort);
btnSelectsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            SelectSort.selectSortIma();
            frame = new JOrdenadoFrame("Select
Sort");
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnInsertsort = new JButton("Insert Sort");
btnInsertsort.setFont(new Font("Arial", Font.BOLD, 14));
btnInsertsort.setBounds(417, 81, 121, 23);
contentPane.add(btnInsertsort);
btnInsertsort.addActionListener(new ActionListener()

```

```

{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            InsertSort.insertSortlma();
            frame = new JOrdenadoFrame("Insert Sort");
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnMergesort = new JButton("Merge Sort");
btnMergesort.setFont(new Font("Arial", Font.BOLD, 14));
btnMergesort.setBounds(417, 115, 121, 23);
contentPane.add(btnMergesort);
btnMergesort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            MergeSort.mergeSortlma();
            frame = new JOrdenadoFrame("Merge
Sort");

        }

        catch (Exception e)
        {
            e.printStackTrace();

```

```

    }
}

});

btnShellsort = new JButton("Shell Sort");
btnShellsort.setFont(new Font("Arial", Font.BOLD, 14));
btnShellsort.setBounds(417, 149, 121, 23);
contentPane.add(btnShellsort);
btnShellsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            ShellSort.shellSortlma();
            frame = new JOrdenadoFrame("Shell Sort");
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

```

```

btnQuicksort = new JButton("Quick Sort");
btnQuicksort.setFont(new Font("Arial", Font.BOLD, 14));
btnQuicksort.setBounds(417, 183, 121, 23);
contentPane.add(btnQuicksort);
btnQuicksort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try

```



```

        {
            QuickSort.quickSortIma();
            frame = new JOrdenadoFrame("Quick Sort");
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

```

```

btnSair = new JButton("Sair");
btnSair.setFont(new Font("Arial", Font.BOLD, 14));
btnSair.setBounds(417, 217, 121, 23);
contentPane.add(btnSair);
btnSair.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        System.exit(0);
    }
});

```

```

btnShakeSort = new JButton("Shake Sort");
btnShakeSort.setFont(new Font("Arial", Font.BOLD, 14));
btnShakeSort.setBounds(548, 11, 121, 23);
contentPane.add(btnShakeSort);
btnShakeSort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {

```

```

        ShakeSort.shakeSortlma();
        frame = new JOrdenadoFrame("Shake
Sort");

    }

    catch (Exception e)
    {
        e.printStackTrace();
    }
}

});

btnHeapsort = new JButton("Heap Sort");
btnHeapsort.setFont(new Font("Arial", Font.BOLD, 14));
btnHeapsort.setBounds(549, 47, 121, 23);
contentPane.add(btnHeapsort);
btnHeapsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            HeapSort.heapSortlma();
            frame = new JOrdenadoFrame("Heap Sort");
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});
}
}

```

ANEXO P – CÓDIGO DA CLASSE JORDENANUMBERFRAME.JAVA

```
package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import java.awt.Toolkit;
import java.awt.event.ActionListener;

import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JButton;
import java.awt.Font;
import java.awt.event.ActionEvent;

@SuppressWarnings("serial")
public class JOrdenaNumberFrame extends JFrame
{

    private JPanel contentPane;
    private JScrollPane scrollPane;
    @SuppressWarnings("rawtypes")
    private JList listNumberOrd;
    private JButton btnBubbleSort;
    private JButton btnShakeSort;
    private JButton btnSelectsort;
    private JButton btnInsertsort;
    private JButton btnMergesort;
    private JButton btnShellsort;
    private JButton btnQuicksort;
    private JButton btnHeapsort;
    private JButton btnSair;
    private int[] temp = new int[Sort.getnElemsInt()];
```

```

@SuppressWarnings("unused")
private JOrdenadoNumberFrame frame;

@SuppressWarnings({ "unchecked", "rawtypes" })
public JOrdenaNumberFrame()
{

    setIconImage(Toolkit.getDefaultToolkit().getImage(JOrdenaNumberFrame.class.getResource("/javax/swing/plaf/metal/icons/ocean/file.gif")));

    setTitle("Ordenar n\u00FAmeros");
    setVisible(true);
    setResizable(false);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 563, 282);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);

    scrollPane = new JScrollPane();
    scrollPane.setBounds(10, 11, 278, 230);
    contentPane.add(scrollPane);

    listNumberOrd = new JList(Sort.dadosNum());
    scrollPane.setViewportView(listNumberOrd);

    for(int i = 0; i < Sort.getnElemsInt(); i++)
        temp[i] = Sort.getArrayInt(i);

    btnBubbleSort = new JButton("Bubble Sort");
    btnBubbleSort.setFont(new Font("Arial", Font.BOLD, 14));
    btnBubbleSort.setBounds(298, 11, 121, 23);
    contentPane.add(btnBubbleSort);
    btnBubbleSort.addActionListener(new ActionListener()

```

```

{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            BubbleSort.bubbleSortInt();
            frame = new
JOrdenadoNumberFrame("Bubble Sort");

            volta();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnSelectsort = new JButton("Select Sort");
btnSelectsort.setFont(new Font("Arial", Font.BOLD, 14));
btnSelectsort.setBounds(298, 47, 121, 23);
contentPane.add(btnSelectsort);
btnSelectsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            SelectSort.selectSortInt();
            frame = new
JOrdenadoNumberFrame("Select Sort");

            volta();

```

```

    }

    catch (Exception e)
    {
        e.printStackTrace();
    }
}

});

btnInsertsort = new JButton("Insert Sort");
btnInsertsort.setFont(new Font("Arial", Font.BOLD, 14));
btnInsertsort.setBounds(298, 81, 121, 23);
contentPane.add(btnInsertsort);
btnInsertsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            InsertSort.insertSortInt();

            frame = new
JOrdenadoNumberFrame("Insert Sort");

            volta();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnMergesort = new JButton("Merge Sort");

```

```

btnMergesort.setFont(new Font("Arial", Font.BOLD, 14));
btnMergesort.setBounds(298, 115, 121, 23);
contentPane.add(btnMergesort);
btnMergesort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            MergeSort.mergeSortInt();
            frame = new
JOrdenadoNumberFrame("Merge Sort");

            volta();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});

btnShellsort = new JButton("Shell Sort");
btnShellsort.setFont(new Font("Arial", Font.BOLD, 14));
btnShellsort.setBounds(298, 149, 121, 23);
contentPane.add(btnShellsort);
btnShellsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            ShellSort.shellSortInt();

```

```

frame = new
JOrdenadoNumberFrame("Shell Sort");

        volta();
    }

    catch (Exception e)
    {
        e.printStackTrace();
    }
}

});

btnQuicksort = new JButton("Quick Sort");
btnQuicksort.setFont(new Font("Arial", Font.BOLD, 14));
btnQuicksort.setBounds(298, 183, 121, 23);
contentPane.add(btnQuicksort);
btnQuicksort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            QuickSort.quickSortInt();
            frame = new
JOrdenadoNumberFrame("Quick Sort");

                volta();
            }

            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}

```



```

    }
});

btnSair = new JButton("Sair");
btnSair.setFont(new Font("Arial", Font.BOLD, 14));
btnSair.setBounds(298, 217, 121, 23);
contentPane.add(btnSair);
btnSair.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        System.exit(0);
    }
});

```

```

btnShakeSort = new JButton("Shake Sort");
btnShakeSort.setFont(new Font("Arial", Font.BOLD, 14));
btnShakeSort.setBounds(429, 11, 121, 23);
contentPane.add(btnShakeSort);
btnShakeSort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            ShakeSort.shakeSortInt();
            frame = new
JOrdenadoNumberFrame("Shake Sort");

            volta();
        }

        catch (Exception e)
        {

```

```

        e.printStackTrace();
    }
}

});

btnHeapsort = new JButton("Heap Sort");
btnHeapsort.setFont(new Font("Arial", Font.BOLD, 14));
btnHeapsort.setBounds(429, 47, 121, 23);
contentPane.add(btnHeapsort);
btnHeapsort.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            HeapSort.heapSortInt();
            frame = new
JOrdenadoNumberFrame("Heap Sort");

            volta();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});
}

public void volta()
{
    for(int i = 0; i < Sort.getnElemsInt(); i++)
        Sort.setArrayInt(temp[i], i);
}

```

```

    }
}

```

ANEXO Q – CÓDIGO DA CLASSE JSOBREFRAME.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Font;
import java.awt.Color;
import java.awt.Toolkit;

@SuppressWarnings("serial")
public class JSobreFrame extends JFrame
{
    private JPanel contentPane;
    private JLabel lblEquipe;
    private JLabel lblBatista;
    private JLabel lblNeves;
    private JLabel lblMagnum;
    private JLabel lblCosta;
    private JButton btnVoltar;

    public JSobreFrame()
    {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setIconImage(Toolkit.getDefaultToolkit().getImage(JSobreFrame.class.getResource("/com/sun/java/swing/plaf/windows/icons/DetailsView.gif")));
    }
}

```

```
setType(Type.UTILITY);
JMenuFrame.sobreVisible = true;
setTitle("Equipe de Desenvolvimento");
setVisible(true);
setResizable(false);
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
setBounds(100, 100, 450, 272);
contentPane = new JPanel();
contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
setContentPane(contentPane);
contentPane.setLayout(null);
```

```
lblEquipe = new JLabel("DESENVOLVEDORES");
lblEquipe.setForeground(Color.RED);
lblEquipe.setFont(new Font("Arial", Font.BOLD, 14));
lblEquipe.setBounds(148, 11, 152, 20);
contentPane.add(lblEquipe);
```

```
lblBatista = new JLabel("Gabriel de Almeida Batista");
lblBatista.setFont(new Font("Arial", Font.BOLD, 14));
lblBatista.setBounds(123, 42, 232, 33);
contentPane.add(lblBatista);
```

```
lblNeves = new JLabel("Felipe da Silva Borges Neves");
lblNeves.setFont(new Font("Arial", Font.BOLD, 14));
lblNeves.setBounds(123, 71, 232, 33);
contentPane.add(lblNeves);
```

```
lblMagnum = new JLabel("Cesar Magnun Oliveira");
lblMagnum.setFont(new Font("Arial", Font.BOLD, 14));
lblMagnum.setBounds(123, 100, 232, 33);
contentPane.add(lblMagnum);
```

```
lblCosta = new JLabel("Jos\u00E9 Vitor Zanoni da Costa");
```

```

lblCosta.setFont(new Font("Arial", Font.BOLD, 14));
lblCosta.setBounds(123, 129, 232, 33);
contentPane.add(lblCosta);

btnVoltar = new JButton("VOLTAR");
btnVoltar.setForeground(Color.BLUE);
btnVoltar.setFont(new Font("Arial", Font.BOLD, 14));
btnVoltar.setBounds(162, 173, 116, 46);
contentPane.add(btnVoltar);

btnVoltar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JMenuFrame.sobreVisible = false;
        setVisible(false);
    }
});
}
}

```

ANEXO R – CÓDIGO DA CLASSE MERGESORT.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class MergeSort extends Sort
{
    private static long count;
    @SuppressWarnings("unused")
    private long[] theArray;

    public static void mergeSortIma()

```

```

        {
            Imagem[] workSpace = new Imagem[getnElemslma()];
            recMergeSortlma(workSpace, 0, getnElemslma() - 1);
            JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count - 1), "Merge Sort", JOptionPane.INFORMATION_MESSAGE);
            count = 0;
        }

        public static void recMergeSortlma(Imagem[] workSpace, int
lowerBound, int upperBound)
        {
            ++count;

            if(lowerBound == upperBound)
                return ;

            else
            {
                int mid = (lowerBound + upperBound) / 2;

                recMergeSortlma(workSpace, lowerBound, mid);

                recMergeSortlma(workSpace, mid + 1 , upperBound);

                mergelma(workSpace, lowerBound, mid + 1,
upperBound);
            }
        }

        public static void mergelma(Imagem[] workSpace, int lowPtr, int highPtr,
int upperBound)
        {
            int j = 0;
            int lowerBound = lowPtr;

```

```

int mid = highPtr - 1;
int n = upperBound - lowerBound + 1;

while(lowPtr <= mid && highPtr <= upperBound)
{
    if(getArrayIma(lowPtr).getWidth() <
    getArrayIma(highPtr).getWidth())
        workspace[j++] = getArrayIma(lowPtr++);

    else
        workspace[j++] = getArrayIma(highPtr++);

    ++count;
}

while(lowPtr <= mid)
{
    workspace[j++] = getArrayIma(lowPtr++);
    ++count;
}

while(highPtr <= upperBound)
{
    workspace[j++] = getArrayIma(highPtr++);
    ++count;
}

for(j = 0; j < n; j++)
{
    setArrayIma(workspace[j], lowerBound+j);
    ++count;
}
}

```

```

public static void mergeSortInt()
{
    int[] workSpace = new int[getnElemsInt()];
    recMergeSortInt(workSpace, 0, getnElemsInt() - 1);
    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count - 1), "Merge Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}

public static void recMergeSortInt(int[] workSpace, int lowerBound, int
upperBound)
{
    ++count;

    if(lowerBound == upperBound)
        return ;

    else
    {
        int mid = (lowerBound + upperBound) / 2;

        recMergeSortInt(workSpace, lowerBound, mid);

        recMergeSortInt(workSpace, mid + 1 , upperBound);

        mergeInt(workSpace, lowerBound, mid + 1, upperBound);
    }
}

public static void mergeInt(int[] workSpace, int lowPtr, int highPtr, int
upperBound)
{
    int j = 0;
    int lowerBound = lowPtr;

```



```
int mid = highPtr - 1;
int n = upperBound - lowerBound + 1;

while(lowPtr <= mid && highPtr <= upperBound)
{
    if(getArrayInt(lowPtr) < getArrayInt(highPtr))
        workspace[j++] = getArrayInt(lowPtr++);

    else
        workspace[j++] = getArrayInt(highPtr++);

    ++count;
}

while(lowPtr <= mid)
{
    workspace[j++] = getArrayInt(lowPtr++);
    ++count;
}

while(highPtr <= upperBound)
{
    workspace[j++] = getArrayInt(highPtr++);
    ++count;
}

for(j = 0; j < n; j++)
{
    setArrayInt(workspace[j], lowerBound+j);
    ++count;
}
}
```

ANEXO S – CÓDIGO DA CLASSE NUMEROPADRAO.JAVA

```

package br.unip.cc.bnmc;

public class NumeroPadrao
{
    public static void insere()
    {
        for(int i = 0; i < 5000000; i++)
            Insere.insere(((int)Math.ceil(Math.random()*10000000)));
    }
}

```

ANEXO T – CÓDIGO DA CLASSE QUICKSORT.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class QuickSort extends Sort
{
    private static long count = 0;

    public static void quickSortlma()
    {
        recQuickSortlma(0, getnElemSlma() - 1);
        JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count - 1), "Quick Sort", JOptionPane.INFORMATION_MESSAGE);
        count = 0;
    }

    public static void recQuickSortlma(int left, int right)
    {
        int size = right - left + 1;

```

```

++count;

if(size < 10) //ordenação por inserção se pequeno
    insertionSortlma(left, right);

else //quick se grande
{
    int median = medianOf3lma(left, right);
    int partition = partitionlma(left, right, median);

    recQuickSortlma(left, partition-1);
    recQuickSortlma(partition+1, right);
}
}

public static int medianOf3lma(int left, int right)
{
    int center = (left + right)/2;

    if(getArraylma(left).getWidth() > getArraylma(center).getWidth())
        swaplma(left, center);

    if(getArraylma(left).getWidth() > getArraylma(right).getWidth())
        swaplma(left, right);

    if(getArraylma(center).getWidth() >
getArraylma(right).getWidth())
        swaplma(center, right);

    swaplma(center, right-1); //coloca pivo a direita

    return getArraylma(right - 1).getWidth(); //retorna valor medio
}

```

```

public static int partitionImlma(int left, int right, int pivot)
{
    int leftPtr = left;
    int rightPtr = right - 1;

    while(true)
    {
        while(getArrayImlma(++leftPtr).getWidth() < pivot)
            ++count; //encontra maior

        while(getArrayImlma(--rightPtr).getWidth() > pivot)
            ++count; //encotra menor

        if(leftPtr >= rightPtr) //se ponteiros cruzam
            break;          //particao feita

        else
            swapImlma(leftPtr, rightPtr); //troca elementos
    }
    swapImlma(leftPtr, right-1); //retorna pivo

    return leftPtr; //retorna posicao do pivo
}

```

```

public static void insertionSortImlma(int left, int right)
{
    int in, out;

    for(out = left = 1; out <= right; out++)
    {
        Imagem temp = getArrayImlma(out);
        in = out;
    }
}

```

```

        while(in >= left && getArraylma(in - 1).getWidth() >=
temp.getWidth())
        {
            setArraylma(getArraylma(in - 1), in);
            --in;
        }

        setArraylma(temp, in);
    }
}

public static void quickSortInt()
{
    recQuickSortInt(0, getnElemsInt() - 1);
    JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count - 1), "Quick Sort", JOptionPane.INFORMATION_MESSAGE);
    count = 0;
}

public static void recQuickSortInt(int left, int right)
{
    int size = right - left + 1;

    ++count;

    if(size < 10) //ordenação por inserção se pequeno
        insertionSortInt(left, 10);

    else //quick se grande
    {
        int median = medianOf3Int(left, right);
        int partition = partitionIntInt(left, right, median);

        recQuickSortInt(left, partition-1);
    }
}

```

```

        recQuickSortInt(partition+1, right);
    }
}

public static int medianOf3Int(int left, int right)
{
    int center = (left + right)/2;

    if(getArrayInt(left) > getArrayInt(center))
        swapInt(left, center);

    if(getArrayInt(left) > getArrayInt(right))
        swapInt(left, right);

    if(getArrayInt(center) > getArrayInt(right))
        swapInt(center, right);

    swapInt(center, right-1);    //coloca pivo a direita

    return getArrayInt(right - 1); //retorna valor medio
}

public static int partitionInt(int left, int right, int pivot)
{
    int leftPtr = left;
    int rightPtr = right - 1;

    while(true)
    {
        while(getArrayInt(++leftPtr) < pivot)
            ++count; //encontra maior

        while(getArrayInt(--rightPtr) > pivot)
            ++count; //encotra menor
    }
}

```

```

        if(leftPtr >= rightPtr) //se ponteiros cruzam
            break;           //particao feita

        else
            swapInt(leftPtr, rightPtr); //troca elementos

        ++count;
    }
    swapInt(leftPtr, right-1); //retorna pivo

    return leftPtr; //retorna posicao do pivo
}

public static void insertionSortInt(int left, int right)
{
    int in, out;

    for(out = left = 1; out <= right; out++)
    {
        int temp = getArrayInt(out);
        in = out;

        while(in >= left && getArrayInt(in - 1) >= temp)
        {
            setArrayInt(getArrayInt(in - 1), in);
            --in;
            ++count;
        }

        setArrayInt(temp, in);
        ++count;
    }
}

```

```
}
```

ANEXO U – CÓDIGO DA CLASSE SELECTSORT.JAVA

```
package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

public class SelectSort extends Sort
{
    private static int out;
    private static int in;
    private static int min;
    private static long count;

    public static void selectSortlma()
    {
        for(out = 0; out < getnElemslma(); out++)
        {
            min = out;

            for(in = out + 1; in < getnElemslma(); in++)
            {
                if(getArraylma(min).getWidth()
getArraylma(in).getWidth())
                    min = in;

                ++count;
            }

            swaplma(out, min);
            ++count;
        }
    }
}
```



```

        JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Select Sort", JOptionPane.INFORMATION_MESSAGE);
        count = 0;
    }

    public static void selectSortInt()
    {
        for(out = 0; out < getnElemsInt(); out++)
        {
            min = out;

            for(in = out + 1; in < getnElemsInt(); in++)
            {
                if(getArrayInt(min) > getArrayInt(in))
                    min = in;

                ++count;
            }

            swapInt(out, min);
            ++count;
        }

        JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Select Sort", JOptionPane.INFORMATION_MESSAGE);
        count = 0;
    }
}

```

ANEXO V – CÓDIGO DA CLASSE SHAKESORT.JAVA

```

package br.unip.cc.bnmc;

import javax.swing.JOptionPane;

```

```

public class ShakeSort extends Sort
{
    private static int out;
    private static int in;
    private static long count;

    public static void shakeSortIma()
    {
        int k = 0, r = 0;

        out = 1;
        r = k = getnElemIma() - 1;

        do
        {
            for(in = r; in >= out; in--)
            {
                if(getArrayIma(in - 1).getWidth() >
getArrayIma(in).getWidth())
                {
                    swapIma(in - 1, in);
                    k = in;
                }

                ++count;
            }

            out = k + 1;

            for(in = out; in <= r; in++)
            {
                if(getArrayIma(in - 1).getWidth() >
getArrayIma(in).getWidth())

```

```

        {
            swaplma(in - 1, in);
            k = in;
        }

        ++count;
    }

    r = k - 1;
    ++count;
} while (out <= r);

JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Shake Sort", JOptionPane.INFORMATION_MESSAGE);
count = 0;
}

public static void shakeSortInt()
{
    int k = 0, r = 0;

    out = 1;
    r = k = getnElemsInt() - 1;

    do
    {
        for(in = r; in >= out; in--)
        {
            if(getArrayInt(in - 1) > getArrayInt(in))
            {
                swapInt(in - 1, in);
                k = in;
            }
        }
    }

```

```

        ++count;
    }

    out = k + 1;

    for(in = out; in <= r; in++)
    {
        if(getArrayInt(in - 1) > getArrayInt(in))
        {
            swapInt(in - 1, in);
            k = in;
        }

        ++count;
    }

    r = k - 1;
    ++count;
} while(out <= r);

JOptionPane.showMessageDialog(null, "Número de interações: "
+ String.valueOf(count), "Shake Sort", JOptionPane.INFORMATION_MESSAGE);
count = 0;
}
}

```

ANEXO W – CÓDIGO DA CLASSE SHELLSORT.JAVA

```

package br.unip.cc.bnmc;

public class ShellSort extends Sort
{
    public static void shellSortIma()
    {

```

```

int inner, outer;
Imagem temp;

int h = 1; //valor inicial de h;

while(h <= getnElemslma() / 3)
    h = h * 3 + 1; //(1, 4, 13, 40, 121, ...)

while(h > 0) //diminuindo h até 1
{
    for(outer = h; outer < getnElemslma(); outer++) //ordena
em h o arquivo
    {
        temp = getArraylma(outer);
        inner = outer;
        //uma passagem (ex 0, 4, 8)
        while(inner > h - 1 && getArraylma(inner -
h).getWidth() >= temp.getWidth())
        {
            setArraylma(getArraylma(inner - h), inner);
            inner -= h;
        }

        setArraylma(temp, inner);
    }

    h = (h - 1) / 3;    //diminui h
}

}

public static void shellSortInt()
{
    int inner, outer, temp;

```

```

int h = 1; //valor inicial de h;

while(h <= getnElemsInt() / 3)
    h = h * 3 + 1; //(1, 4, 13, 40, 121, ...)

while(h > 0) //diminuindo h até 1
{
    for(outer = h; outer < getnElemsInt(); outer++) //ordena em
        h o arquivo
        {
            temp = getArrayInt(outer);
            inner = outer;
            //uma passagem (ex 0, 4, 8)
            while(inner > h - 1 && getArrayInt(inner - h) >=
temp)
            {
                setArrayInt(getArrayInt(inner - h), inner);
                inner -= h;
            }

            setArrayInt(temp, inner);
        }

        h = (h - 1) / 3;    //diminui h
    }
}

```

ANEXO X – CÓDIGO DA CLASSE SORT.JAVA

```
package br.unip.cc.bnmc;
```

```
import java.util.ArrayList;
```

```
import javax.swing.DefaultListModel;

public class Sort
{
    private int nElems;
    private static ArrayList<Imagem> theArrayIm;
    private static ArrayList<Integer> theArrayInt;

    public Sort()
    {
        theArrayIm = new ArrayList<Imagem>();
        theArrayInt = new ArrayList<Integer>();
        nElems = 0;
    }

    public void insert(Imagem value, String nome)
    {
        theArrayIm.add(value);
        theArrayIm.get(nElems).setNome(nome);
        nElems++;
    }

    public void insert(int value)
    {
        theArrayInt.add(value);
        nElems++;
    }

    public static void setArrayIma(Imagem value, int in)
    {
        theArrayIm.set(in, value);
    }

    public static void setArrayInt(int value, int in)
```

```
{  
    theArrayInt.set(in, value);  
}  
  
public static void swapIma(int dex1, int dex2)  
{  
    Imagem temp = theArrayIm.get(dex1);  
    theArrayIm.set(dex1, theArrayIm.get(dex2));  
    theArrayIm.set(dex2, temp);  
}  
  
public static void swapInt(int dex1, int dex2)  
{  
    int temp = theArrayInt.get(dex1);  
    theArrayInt.set(dex1, theArrayInt.get(dex2));  
    theArrayInt.set(dex2, temp);  
}  
  
public static int getnElemsIma()  
{  
    return theArrayIm.size();  
}  
  
public static int getnElemsInt()  
{  
    return theArrayInt.size();  
}  
  
public static Imagem getArrayIma(int in)  
{  
    return theArrayIm.get(in);  
}  
  
public static int getArrayInt(int in)
```



```

    {
        return theArrayInt.get(in);
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static DefaultListModel dadosIma()
    {
        DefaultListModel DLM = new DefaultListModel();

        for(int i = 0; i < theArrayIm.size(); i++)
            DLM.addElement(theArrayIm.get(i).getNome() + "
" + theArrayIm.get(i).getWidth() + "
" + theArrayIm.get(i).getHeight());

        return DLM;
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public static DefaultListModel dadosNum()
    {
        DefaultListModel DLM = new DefaultListModel();

        for(int i = 0; i < theArrayInt.size(); i++)
            DLM.addElement(theArrayInt.get(i));

        return DLM;
    }
}

```