

**Atividade Pratica Supervisionada**

Trabalho Semestral.

**“DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS”**

Nome: Arthur dos Santos Marques– D36691-2 – Ciência da Computação

Nome: Matheus Soares S Araújo – N1287C-8 – Ciência da Computação

Nome: Matheus dos Santos O Pardo – N173GJ-7 – Ciência da Computação

Nome: Vítor Hugo de Souza Sierro – D29DEG-4 – Ciência da Computação

**Sumário**

1–Objetivo do Trabalho ......................................................................................3

2-Introdução.........................................................................................................4

3-Referencial Teórico............................................................................................

3.1 – Bubble Sort...............................................................................................6

3.2 – Merge Sort................................................................................................7

3.3 – Quick Sort.................................................................................................8

3.4 – Insert Sort.................................................................................................9

4-Desenvolvimento............................................................................................10

5-Resultados e Discussão.................................................................................19

6-Considerações Finais.....................................................................................25

7-Referência Bibliográfico..................................................................................26

8-Código Fonte..................................................................................................28

**Objetivo do Trabalho**

O Objetivo dessa Atividade Pratica Supervisionada é analisar o desempenho de algoritmos de ordenação, tais como:

* Bubble Sort
* Merge Sort
* Select Sort
* Quick Sort

**Introdução**

A ordenação é um dos aspectos fundamentais do mundo tecnológico.  
 Sendo assim torna-se importante reduzir ao máximo a complexidade temporal dos algoritmos. As melhores ordenações em série normalmente demoram O(n log n), tempo que tende a aumentar ou diminuir de acordo com o número de elementos. Desta forma, foram desenvolvidas funções para funcionamento em paralelo aos algoritmos, cujo objetivo é diminuir consideravelmente o tempo de execução.

Neste trabalho serão abordados alguns algoritmos de ordenação que realizam operações de comparação e troca, sendo eles o Bubble Sort, Merge Sort, Insert Sort e Quick Sort.

Existem diversos métodos de ordenação de dados, mas nesse trabalho iremos mostrar e exemplificar alguns dos métodos utilizados nos dias de hoje.

O método Bubble sort (Bolha), este método faz uma comparação dos elementos com seu anterior, ele compara o segundo valor de um vetor seguindo tanto da esquerda para direita quanto vice versa, e caso o valor seja menor que o outro ele efetua as trocas dos elementos até que os mesmos estejam ordenados.

O método Merge sort(Mistura) É um exemplo de algoritmo que faz uso da estratégia “dividir para conquistar” para resolver os devidos problemas. O seu método de funcionamento é bem simples, o mesmo divide o problema em diversos pedaços menores, resolve os pedaços e depois junta os resultados. O vetor é dividido em duas partes iguais, que serão divididas em duas partes, e assim até ficar dois elemento cuja ordenação é trivial.

O método Insert sort (Inserção), age de um modo bem simples, o que ele faz é pegar o primeiro elemento e comparar com o resto dos elementos que se encontram no vetor, após isso o mesmo posiciona o primeiro elemento na posição correta conforme sua ordenação, caso ele compare o primeiro elemento do vetor com os seguintes, se o mesmo for menor que todos ele o posiciona na primeira posição, caso o contrario ele compara os outros elementos até achar o menor e ir posicionando cada elemento em seu respectivo lugar.

O método Quick sort, é um método bastante utilizado, é conhecido como o método “dividir para se conquistar”, o quick sort funciona escolhendo aleatoriamente um pivô dos dados inseridos no vetor, e logo após escolher o pivô ele divide o vetor em duas partes, colocando os dados maiores que seu pivô a sua direita, e os menores a sua esquerda, achando a posição correta para o pivô, após organizar os dois lados, o mesmo escolhe outro pivô e repete o procedimento, até que os dados no vetor estejam corretamente ordenados.

Todos os quatros métodos apresentados acima, são tipos de ordenação bem utilizados, logicamente cada um tem seu ponto de vantagem e desvantagem, fizemos a escolha de três destes quatros métodos apresentados e iremos mostrar o porquê escolhemos cada um destes.

**Referencial Teórico**

Como citado de antemão os algoritmos de ordenação para esta atividade inclui:

* Bubble Sort: É um dos mais simples dos algoritmos de comparação. Esse método compara os elementos dois a dois, por exemplo: compara-se a primeira posição com a segunda, e na segunda repetição compara-se a segunda posição com a terceira, e assim sucessivamente, assim é possível ordenar o vetor de forma crescente ou decrescente.

Segue um exemplo simples de ordenação em ordem crescente.

Valores: 5 – 4 – 2 – 1 - 3

Primeiro compara-se as duas primeiras posições, nesse caso o número 5 e 4; caso o primeiro numero seja maior que o segundo deve acontecer a troca. Como 5 é maior que 4 então a troca é realizada:

4 – 5 – 2 – 1 – 3

5 é maior que 2, então:

4 – 2 – 5 – 1 – 3

5 é maior que 1, logo:

4 – 2 – 1 – 5 – 3

5 é maior que 3, então:

4 – 2 – 1 – 3 - 5

Agora o 5 se encontra ordenado corretamente. Agora é necessário ordenar o resto da sequencia.

* Merge Sort: É um exemplo de algoritmo que faz uso da estratégia “dividir para conquistar” para resolver os devidos problemas. O seu método de funcionamento é estável e possui uma complexidade única para todos os casos, o mesmo divide o problema em diversos pedaços menores, resolve os pedaços e depois junta os resultados. O vetor é dividido em duas partes iguais, que serão divididas em duas partes, e assim até ficar dois elemento cuja ordenação é trivial.

Para juntar as partes ordenadas os dois elementos de cada parte são separados e o menor é retirado de sua parte. Logo em sequencia os menores entre os demais são comparados e assim segue até se juntar todas as partes.

Segue um exemplo do funcionamento do Merge Sort

Valores: 6 5 3 1 8 7 2 4

Separados a primeira vez : | 6 e 5 | 3 e 1 | 8 e 7 | 2 e 4 |

Primeira ordenação : |5 e 6 | 1 e 3| 7 e 8 | 2 e 4 |

Primeira junção de dois grupos ordenada: |1, 3, 5 e 6 | 2, 4, 7 e 8|

Ordenação final: | 1, 2, 3, 4, 5, 6, 7 e 8 |

* Select Sort: É o método que realiza a ordenação selecionando o menor item e colocando o na primeira posição, e assim continua, com o segundo menor item e colocando o na segunda posição, e com todos os elementos restantes, até que reste um único elemento. O método select sort, não é um tipo de algoritmo estável e para todos os casos possíveis possui a complexidade C(n) = O(n²).

Exemplo de Funcionamento:

Valores inicias: **8 5 2 6 9 3 1 4 0 7**

Na primeira verificação é feita a substituição dos valores **0** e **8**, invertendo assim suas posições : **0 5 2 6 9 3 1 4 8 7**

Em sequencia faz a alteração dos valore **1** e **5**: **0 1 2 6 9 3 5 4 8 7**

Logo após é feita a alteração dos valores **3** e **6**: **0 1 2 3 9 6 5 4 8 7**

E assim o processo irá se repetir a quantidade de vezes necessária para conseguir chegar em seu resultado final, no caso do exemplo utilizado por nós seria: **0 1 2 3 4 5 6 7 8 9**

* Quick Sort: É o método de ordenação interna mais rápido que se conhece para uma variedade de situações e provavelmente é o mais utilizado. É um algoritmo de comparação onde o mesmo divide o problema de ordenar em um conjunto com diversos itens em dois problemas menores. Os problemas menores são ordenados independentemente e os resultados combinados para produzir a solução final.

Para seu funcionamento é definido um elemento da lista chamado de pivô, logo após é feita a reorganização da lista de forma que os elementos menores que o pivô fique de um lado e os maiores fiquem de outro, e no final é feita recursivamente a ordenação da sub-lista abaixo e acima do pivô.

**Desenvolvimento**

Serão apresentados aqui os estágios pelos quais passaram o desenvolvimento até o software ser totalmente concluído.

Estagio 1: Pesquisa, foi feita uma pesquisa detalhada sobre o tema antes de iniciar o desenvolvimento do software.

Estagio 2: Escolha dos quatro métodos de ordenação utilizados no software, após a pesquisa comparamos os métodos de ordenação para ver quais eram os mais rápidos ao serem executados.

Estagio 3: Desenvolvimento do código, após escolhes os quatro métodos de ordenação mais apropriados foi passado para o código do programa.

A obtenção de dados foi feita através de um vetor criado aleatoriamente pelo sistema do Java utilizando o método do sistema math.random, e passamos esse vetor para os métodos bubbleSort, QuickSort, MergeSort, SelectionSort que ao ser executado cada um e obtido o tempo de cada um e feito sua comparação de desempenho.

A seguir estará listado uma parte do código utilizado para gerar as obtenções de dados, e métodos de ordenação utilizados.

**Inicio do Sistema**

A linguagem programa no sistema foi Java.

Primeiramente declaramos as bibliotecas utilizadas no Sistema como mostramos logo abaixo:

import java.util.Scanner;

import javax.swing.JOptionPane;

import java.io.IOException;

Após, contém a classe utilizada dentro do sistema inteiro, sendo todos os códigos descrito dentro dela.

A classe principal, e responsável para chamar uma classe chamada métodos na qual contém todo o sistema de ordenamento, através do Methods calc = new Methods (); e logo em baixo chamo o método iniciar contendo na classe methods através de calc.iniciar();

Declaramos as variáveis de vetor como inteira, por isso se o usuário digita um valor maior que 99999 pode ocorre erro de estouro de variável pois o tipo int e igual a 2³².

Foi utilizado o JOptionPane por causa de estética caso ocorra algum erro com o index do vetor.

O Vetor aleatório foi declarado em um método chamada vetor desordenado e segue em baixo sua declaração.

**public** **int**[] vetorDesordenado() {

vetor = **new** **int**[quantidade];

**for** (**int** i = 0; i < vetor.length; i++) {

vetor[i] = (**int**) (Math.*random*() \*quantidade);

}

**return** vetor;

}

As variáveis globais foram descritas em baixo, só ressaltando criou-se um novo vetor para facilitar o manuseio desse vetor visto que já definido não precisaria ficar chamando toda vez a método vetor desordenado pois poderia ocorre perda de dados e muda, pois, todas as vezes que fosse chamado executaria o comando de novo.

int[] vetor;

int quantidade;  
 Scanner entrada = new Scanner (System.*in);*

Observação esse método scanner e executado antes por facilitar chamar o método scanner apenas uma vez caso o programado queira utilizar mais de uma vez.

Depois de declarado essas variáveis vou começar pelo método que utilizo para chamá-lo na classe principal que é o método iniciar.

**public** **void** iniciar () **throws** IOException

{

exibe ();

System.***out***.println("Qual o tamanho do vetor que você deseja para efetuar os cálculos");

System.***out***.println("recomendo maior que 10.000 pois fica mais fácil de visualizar");

quantidade = entrada.nextInt();

vetorDesordenado ();

prints ();

}

Vou explicar o que é esse método exibe, ele mostra todos os métodos utilizados o que está escrito nesse método vem logo abaixo, ele imprime na tela de console o que estão em azul um abaixo do outro.

**public** **void** exibe ()

{

System.***out***.println ("Metodologias utilizadas");

System.***out***.println ("1 - Bubble Sort");

System.***out***.println ("2 – Quick Sort");

System.***out***.println ("3 – Merge Sort");

System.***out***.println ("4 - Selection Sort");

}

Voltando ao método iniciar, como você pode ver o usuário pode declarar o valor do vetor recomendo ele declarar um vetor maior que 10 mil, pois fica mais fácil de visualizar o tempo algo que vou explicar mais tarde nesse capitulo.

Visto que já mostrei o vetor desordenado logo em cima vou passa para o método prints.

**public** **void** prints ()

{

**int** [] som1 = **new** **int** [4];

String [] let = {"QuickSort","BubbleSort","MergeSort","SelectionSort"};

**int**[] soma = **new** **int**[4];

Declaradas as variáveis desse método, que serão usadas a segui, crio duas variáveis som1 e soma para fazer ordenamento de variável para ficar certinho na hora de exibe o tempo junto com as letras do método.

**try** {

**long** tempoInicial = System.*currentTimeMillis*();

*bubbleSort* (vetor, vetor.length-1);

**long** tempoFinal = System.*currentTimeMillis* ();

soma [0] = (**int**) (tempoFinal - tempoInicial);

System.***out***.println ("Executado em BubbleSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.*currentTimeMillis* ();

*quickSort* (vetor, 0, vetor.length-1);

tempoFinal = System.*currentTimeMillis* ();

soma[1] = (**int**) (tempoFinal - tempoInicial);

System.***out***.println("Executado em quickSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.*currentTimeMillis* ();

*mergeSort* (vetor, 0, vetor.length-1);

tempoFinal = System.*currentTimeMillis*();

soma[2] = (**int**) (tempoFinal - tempoInicial);

System.***out***.println("Executado em MergeSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.*currentTimeMillis*();

*doSelectionSort*(vetor);

tempoFinal = System.*currentTimeMillis*();

soma[3] = (**int**) (tempoFinal - tempoInicial);

System.***out***.println("Executado em SelectionSort = " + (tempoFinal - tempoInicial) + " ms");

**for**(**int** i =0 ;i <soma.length;i++)

{

som1[i] = soma[i];

}

*quickSort* (soma, 0, soma.length-1);

System.***out***.println ("Vetor organizado:");

**for**(**int** q = 0; q < soma.length; q++)

{

**for**(**int** j = 0; j < soma.length; j++)

{

**if** (soma[q] == som1[j])

{ System.***out***.println(let[j]+" "+soma[q]+ "ms");

}

}

}

}**catch**(IndexOutOfBoundsException e) {

e.printStackTrace();

JOptionPane.*showMessageDialog*(**null**, "Error no vetor","Error", JOptionPane.***INFORMATION\_MESSAGE***);

}

}

Nesse método utilizei o try catch pois é a parte final do programa onde será exibido os resultados de cada método, visto isso o catch que utilizo faz um IndexOutOfBoundException pois o único erro que observei seria de index do vetor, se caso aconteça isso solta um pop-up na tela avisando que deu erro no vetor.

Disseminada a parte do try catch vamos as variáveis do tempo utilizei em cada método para ficar mais fácil de comparar visto que eu usei, só uma vez não teria comparação só ficaria aparecendo o tempo de execução, utilizei o método long pois é o maior para um tipo inteiro, e visto que os números são um pouco grandes esse seria o melhor método.

No final de cada método ele imprime na tela seu valor e o tempo percorrido pela conta tempoFinal – tempoInicial e o resultado fica em ms(Milissegundos) é depois disso executa o próximo método repetindo o que acabei de descreve, só que em outro método, após executar todos os métodos mando ordena o que foi armazenado no vetor soma(que são o tempo de cada método) dentro desse método de ordenação que se consiste de dois for um em baixo do outro utilizo um método de condição if na qual faz a verificação se o vetor soma e igual ao som1 para verificar se os vetores são iguais, visto que utilizei o método quicksort antes para ordena-lo, visto que são igual ele imprime na tela a letra e o valor ordenando pelo menor para o maior tempo.

Agora vou descreve o código de cada método de ordenamento e explicar qual as vantagens de cada um.

Começando pelo BubbleSort:

- O algoritmo (código) é muito simples e facilita o aprendizado.

- Os elementos são trocados de lugar sem utilizar armazenamento temporário

Código:

**public** **static** **void** bubbleSort (**int** [] vet, **int** fim) {

**int** aux = 0;

**for** (**int** i = 0; i < fim; i++) {

**for**(**int** j = 0; j < fim; j++){

**if**(vet[j] > vet[j + 1]){

aux = vet[j];

vet[j] = vet[j+1];

vet[j+1] = aux;

}

}

}

}

QuickSort.

- É extremamente eficiente para ordenar arquivos de dados.

- Caso médio e melhor caso rodam em uma complexidade em que n log n.

Nesse método utilizando mais utilizamos um pouco de recursividade e outro método chamado separar.

QuickSort.

**public** **static** **void** quickSort(**int**[] vetor, **int** inicio, **int** fim) {

**if** (inicio < fim) {

**int** posicaoPivo = *separar*(vetor, inicio, fim);

*quickSort*(vetor, inicio, posicaoPivo - 1);

*quickSort*(vetor, posicaoPivo + 1, fim);

}

}

Separar.

**private** **static** **int** separar(**int**[] vetor, **int** inicio, **int** fim) {

**int** pivo = vetor[inicio];

**int** i = inicio + 1, f = fim;

**while** (i <= f) {

**if** (vetor[i] <= pivo)

i++;

**else** **if** (pivo < vetor[f])

f--;

**else** {

**int** troca = vetor[i];

vetor[i] = vetor[f];

vetor[f] = troca;

i++;

f--;

}

}

vetor[inicio] = vetor[f];

vetor[f] = pivo;

**return** f;

}

Merge Sort.

- Pior caso e caso médio na qual Ɵ (n log n).

O código do merge sort também tem dois métodos o mergesort e o merge.

Merge:

**private** **static** **void** merge(**int**[] A, **int** p, **int** q, **int** r) {

**int**[] aux = **new** **int**[r - p + 1];

**int** a = p;

**int** b = q + 1;

**int** h = 0;

**while** (a <= q && b <= r) {

**if** (A[a] < A[b]) {

aux[h++] = A[a++];

} **else** {

aux[h++] = A[b++];

}

}

**while** (a <= q) {

aux[h++] = A[a++];

}

**while** (b <= r) {

aux[h++] = A[b++];

}

**for** (h = 0; h < aux.length; h++) {

A[p++] = aux[h];

}

}

MergeSort

**public** **static** **void** mergeSort(**int**[] A, **int** p, **int** r) {

**int** q = (p + r) / 2;

**if** (p < r) {

*mergeSort*(A, p, q);

*mergeSort*(A, q + 1, r);

*merge*(A, p, q, r);

}

}

Selection Sort

- Ele é um algoritmo simples de ser implementado em comparação aos demais.

-Não necessita de um vetor auxiliar (in-place).

-Por não usar um vetor auxiliar para realizar a ordenação, ele ocupa menos memória.

doSelectionSort.

**public** **static** **int**[] doSelectionSort(**int**[] arr){

**for** (**int** i = 0; i < arr.length - 1; i++)

{

**int** index = i;

**for** (**int** j = i + 1; j < arr.length; j++)

**if** (arr[j] < arr[index])

index = j;

**int** smallerNumber = arr[index];

arr[index] = arr[i];

arr[i] = smallerNumber;

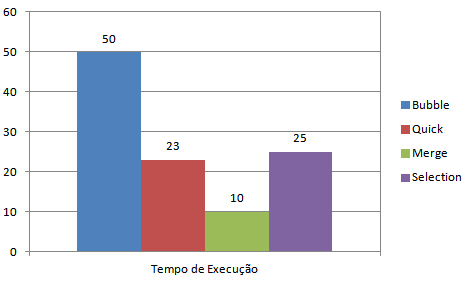
}

**return** arr; }

**Resultados e Discussão**

Neste capitulo mostraremos o tempo de execução de cada algoritmo de ordenação e suas principais vantagens.

O gráfico abaixo mostra a media tempo de execução dos algoritmos nos devidos testes realizados.



Como podemos observar de acordo com o gráfico o mais eficiente foi o Merge Sort, seguido pelo Quick Sort, Selection Sort e por ultimo Bubble Sort.

Os dados realizados para criação do gráfico foi á média aritmética do tempo de execução realizado nos diversos testes. Para os testes que realizamos fizemos a requisição de entrada de dados aleatórios para o sistema, os testes tiveram uma gama entre cinco mil elementos á trinta mil elementos.

**MERGE SORT**

O Merge Sort ou ordenação por mistura foi o método de ordenação mais eficiente em nossos testes, onde o mesmo apresentou uma média de 10 MS para execução e ordenação, conseguimos observar que ele se saiu bem em todos os devidos testes tanto na gama de numero mais baixo, quanto com uma gama de numero mais alto.

O Mesmo conseguiu se destacar dos outros métodos de ordenação tendo uma quantidade significativa de diferença para o segundo colocado que foi o Quick Sort.

As vantagens do Merge Sort:

* Fácil Implementação;
* Complexidade única para todos os casos;
* É estável.

Desvantagens:

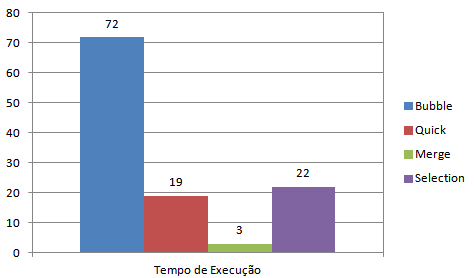
* Utiliza funções recursivas;
* Gasto extra de memória.

**QUICK SORT**

O Quick Sort, apesar de ser o segundo colocado nesses diversos testes de ordenação, não decepcionou e mostrou o porquê é considerado um dos melhores algoritmos de ordenação existente.

Com o Quick Sort conseguimos ter uma média de 23 ms para execução e ordenação dos dados, assim como o Merge Sort o mesmo apresentou um ótimo desempenho com uma gama alta de elementos, porem com uma gama baixa o mesmo não conseguiu se sair muito bem, onde quando foi solicitado a ordenação de seis mil elementos o mesmo teve um tempo de resposta relativamente alto de 19ms.

Segue abaixo gráfico de desempenho com seis mil elementos.



Apesar de ter esse problema com uma gama baixa de elementos, ao ser feito o teste com uma grande quantidade, o mesmo mostrou o porque é considerado um dos melhores ordenadores e manteve-se com um numero baixo perto dos outros.

As vantagens do Quick Sort:

* Eficiente;
* Necessita apenas de uma pequena pilha como memória extra;
* Complexidade n log n.

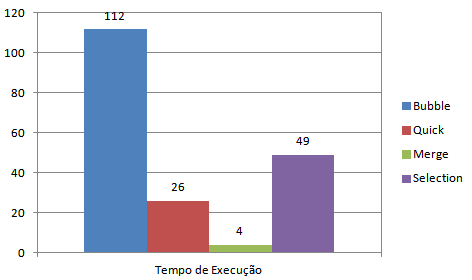
Desvantagens:

* Tem um pior caso de 0(n2);
* Implementação difícil;
* Não é estável.

**SELECTION SORT**

Assim como o Quick Sort, o SELECTION se saiu muito bem, o mesmo apresentou certas dificuldades para ordenação envolvendo uma grande quantidade de elementos, levando em consideração seu método de ordenação que consiste em selecionar o item, fazer a verificação e coloca-lo na posição correta, podemos afirmar que ele funcionou bem assim como os outros métodos utilizados.

Segue abaixo gráfico de desempenho com dez mil elementos.



O Selection Sort se mostrou bem eficiente nos testes realizados com uma baixa gama, sendo assim mostrando onde o mesmo deve ser utilizado.

As vantagens do Selection Sort:

* Simples implantação;
* Não necessita de vetor auxiliar;
* Ocupa menos memória;
* Velocidade de ordenação para vetores pequenos incríveis.

Desvantagens:

* Velocidade de ordenação para vetores grandes lenta;
* Ele sempre vai realizar (n² - n)/2 comparações, independente do elemento se encontrar ordenado ou não;
* Não é estável.

**BUBBLE SORT**

O Bubble Sort diferente dos outros três métodos de ordenação apresentados acima, se demonstrou muito lento em todos os quesitos, sendo lento com uma pequena gama de elementos ou uma grande gama de elementos.

O mesmo chegou a apresentar resultados absurdos como no gráfico abaixo, onde o mesmo diferente dos outros algoritmos que ficaram com um valor abaixo de 50ms o mesmo chegou a ter mais de 100ms

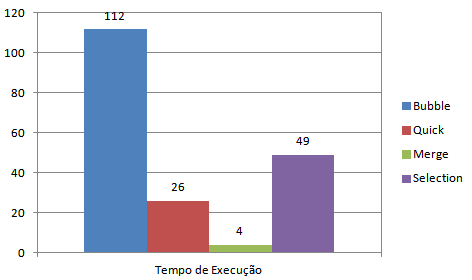


Gráfico de desempenho com dez mil elementos.

Não conseguimos definir uma vantagem para o mesmo, pois o mesmo com os devidos testes realizados por nós com números gerados automaticamente não se mostrou utilizável em nenhuma dos testes. Sendo assim todos os critérios ficaram para a desvantagem, onde os próprios gráficos de desempenho já mostram parte de sua desvantagem.

**Considerações Finais**

O propósito principal deste trabalho foi de apresentar e analisar a frequência da ordenação dos algoritmos, com uma abordagem dos três tipos de algoritmos que são Bubble Sort, Merge Sort, Select Sort, Quick Sort.

Este trabalho possibilitou ao grupo o aprofundamento mais detalhados dos algoritmos de ordenação apresentados no curso, visto que com a análise dos números, comparações e tempo foram bem eficientes. Pelos resultados que vimos, foram verificados os quatros tipos de algoritmos Bubble Sort, Merge Sort, Select Sort e Quick Sort e conseguimos ver a eficiência de ambos.

Entre os quatros algoritmos apresentados, conseguimos verificar que o Bubble Sort é bem mais lento em questão de execução comparado com os outros algoritmos, assim como na questão de Organização do vetor, o Bubble Sort continuou mais lento de todos. O mais rápido de todos os testes realizados foi o Merge Sort, mas com toda certeza não foi mais eficiente, levando em consideração que o mesmo é melhor funcional para uma quantidade pequena de dados e não para uma quantidade grande de dados.

**Referencias Bibliográficas**

PTCOMPUTADOR. Vantagens e Desvantagens de Bubble Sort. Disponivel em: < http://ptcomputador.com/P/computer-programming-languages/87814.html>. Acessado em: 21 de Novembro de 2018

WIKIPEDIA. Selection Sort. Disponivel em: < https://pt.wikipedia.org/wiki/Selection\_sort>. Acessado em: 21 de Novembro de 2018

WIKIPEDIA. Selection Sort. Disponivel em: < https://pt.wikipedia.org/wiki/Selection\_sort>. Acessado em: 21 de Novembro de 2018

WIKIPEDIA. Merge Sort. Disponivel em: < https://pt.wikipedia.org/wiki/Merge\_sort>. Acessado em: 21 de Novembro de 2018

WIKIPEDIA. Bubble Sort. Disponivel em: < https://pt.wikipedia.org/wiki/Bubble\_sort>. Acessado em: 21 de Novembro de 2018

ORDENARC. O Que é Merge Sort e como ele funciona. Disponivel em: < http://ordenarc.blogspot.com/2011/06/o-que-e-merge-sort-e-como-ele-funciona.html>. Acessado em: 21 de Novembro de 2018

TREINA WEB. Conheça os principais algoritmos de ordenação. Disponivel em: < https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>. Acessado em: 21 de Novembro de 2018

DEVMEDIA. Conheça Entendendo o Algoritmo Bubble Sort em Java. Disponivel em: < https://www.devmedia.com.br/entendendo-o-algoritmo-bubble-sort-em-java/24812 >. Acessado em: 21 de Novembro de 2018

EMBARCADOS. Algoritmos de Ordenação: Bubble Sort. Disponivel em: < https://www.embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/>. Acessado em: 21 de Novembro de 2018

KHAN ACADEMY. Visão Geral do Merge Sort em: < https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort>. Acessado em: 21 de Novembro de 2018

KHAN ACADEMY. Analise do Merge Sort em: < https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort>. Acessado em: 21 de Novembro de 2018

IMES USP. MergeSort: Ordenação por intercalação em: < https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html>. Acessado em: 21 de Novembro de 2018

IMES USP. MergeSort: Ordenação por intercalação em: < https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html>. Acessado em: 21 de Novembro de 2018

PESC – Programa de Engenharia de Sistemas e Computação. MergeSort – Estrutura de Dados em: < https://www.cos.ufrj.br/~rfarias/cos121/aula\_07.html>. Acessado em: 21 de Novembro de 2018

**Código Fonte**

**Methods:**

//---------------------------------Pacotes------------------------------------------------

package APS;

//--------------------------------Biblioteca--------------------------------------------------------------------------------

import java.io.IOException;

import java.util.Scanner;

import javax.swing.JOptionPane;

//---------------------------------Classes-Metodos----------------------------------------

public class Methods {

//----------------------------------Variaveis--Globais------------------------------------------------------------------------

Scanner entrada = new Scanner(System.in);

int[] vetor;

int quantidade;

//---------------------------------Metodo-Construtor--------------------------------------------------------------------------

public Methods()

{

}

//---------------------------------Bubble Sort---------------------------------------------------------------------------------

public static void bubbleSort(int[] vet, int fim) {

int aux = 0;

for(int i = 0;i < fim; i++){

for(int j = 0; j < fim; j++){

if(vet[j] > vet[j + 1]){

aux = vet[j];

vet[j] = vet[j+1];

vet[j+1] = aux;

}

}

}

}

//---------------------------------QuickSort----------------------------------------------------------------------------------

public static void quickSort(int[] vetor, int inicio, int fim) {

if (inicio < fim) {

int posicaoPivo = separar(vetor, inicio, fim);

quickSort(vetor, inicio, posicaoPivo - 1);

quickSort(vetor, posicaoPivo + 1, fim);

}

}

private static int separar(int[] vetor, int inicio, int fim) {

int pivo = vetor[inicio];

int i = inicio + 1, f = fim;

while (i <= f) {

if (vetor[i] <= pivo)

i++;

else if (pivo < vetor[f])

f--;

else {

int troca = vetor[i];

vetor[i] = vetor[f];

vetor[f] = troca;

i++;

f--;

}

}

vetor[inicio] = vetor[f];

vetor[f] = pivo;

return f;

}

//---------------------------------MergeSort----------------------------------------------------------------------------------

private static void merge(int[] A, int p, int q, int r) {

int[] aux = new int[r - p + 1];

int a = p;

int b = q + 1;

int h = 0;

while (a <= q && b <= r) {

if (A[a] < A[b]) {

aux[h++] = A[a++];

} else {

aux[h++] = A[b++];

}

}

while (a <= q) {

aux[h++] = A[a++];

}

while (b <= r) {

aux[h++] = A[b++];

}

for (h = 0; h < aux.length; h++) {

A[p++] = aux[h];

}

}

public static void mergeSort(int[] A, int p, int r) {

int q = (p + r) / 2;

if (p < r) {

mergeSort(A, p, q);

mergeSort(A, q + 1, r);

merge(A, p, q, r);

}

}

//---------------------------------SelectionSort----------------------------------------------------------------------------------

public static int[] doSelectionSort(int[] arr){

for (int i = 0; i < arr.length - 1; i++)

{

int index = i;

for (int j = i + 1; j < arr.length; j++)

if (arr[j] < arr[index])

index = j;

int smallerNumber = arr[index];

arr[index] = arr[i];

arr[i] = smallerNumber;

}

return arr;

}

//------------------------------------Vetor--Desordenado--------------------------------------------------------------------

public int[] vetorDesordenado() {

vetor = new int[quantidade];

for (int i = 0; i < vetor.length; i++) {

vetor[i] = (int) (Math.random()\*quantidade);

}

return vetor;

}

//---------------------------------Metodo-Iniciar-----------------------------------------------------------------------------

public void iniciar() throws IOException

{

exibe();

System.out.println("Qual o tamanho do vetor que você deseja para efetuar os cálculos");

System.out.println("recomendo maior que 10.000 pois fica mais fácil de visualizar");

quantidade = entrada.nextInt();

vetorDesordenado();

prints();

}

//---------------------------------Exibe-Menu---------------------------------------------------------------------------------

public void exibe()

{

System.out.println("Metodologias utilizadas");

System.out.println("1 - BubbleSort");

System.out.println("2 - QuickSort");

System.out.println("3 - MergeSort");

System.out.println("4 - SelectionSort");

}

//---------------------------------Escolher-Prints-----------------------------------------------------------------------------

public void prints()

{

int[] som1 = new int[4];

String[] let = {"QuickSort","BubbleSort","MergeSort","SelectionSort"};

int[] soma = new int[4];

try {

long tempoInicial = System.currentTimeMillis();

bubbleSort(vetor,vetor.length-1);

long tempoFinal = System.currentTimeMillis();

soma[0] = (int) (tempoFinal - tempoInicial);

System.out.println("Executado em BubbleSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.currentTimeMillis();

quickSort(vetor,0,vetor.length-1);

tempoFinal = System.currentTimeMillis();

soma[1] = (int) (tempoFinal - tempoInicial);

System.out.println("Executado em quickSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.currentTimeMillis();

mergeSort(vetor,0,vetor.length-1);

tempoFinal = System.currentTimeMillis();

soma[2] = (int) (tempoFinal - tempoInicial);

System.out.println("Executado em MergeSort = " + (tempoFinal - tempoInicial) + " ms");

tempoInicial = System.currentTimeMillis();

doSelectionSort(vetor);

tempoFinal = System.currentTimeMillis();

soma[3] = (int) (tempoFinal - tempoInicial);

System.out.println("Executado em SelectionSort = " + (tempoFinal - tempoInicial) + " ms");

for(int i =0 ;i <soma.length;i++)

{

som1[i] = soma[i];

}

quickSort(soma,0,soma.length-1);

System.out.println("Vetor organizado:");

for(int q = 0; q < soma.length; q++)

{

for(int j = 0; j < soma.length; j++)

{

if (soma[q] == som1[j])

{

System.out.println(let[j]+" "+soma[q]+ "ms");

}

}

}

}catch(IndexOutOfBoundsException e) {

e.printStackTrace();

JOptionPane.showMessageDialog(null, "Error no vetor","Error", JOptionPane.INFORMATION\_MESSAGE);

}

}

}//fecha a classe Methods

**Principal Class:**

//------------------------------------Pacote----------------------------------------------------------------------------------------

**package** APS;

//---------------------------------Biblioteca---------------------------------------------------------------------------------------

**import** java.io.IOException;

//---------------------------------Classe-------------------------------------------------------------------------------------------

**public** **class** Principal\_Class

{

//---------------------------------Metodo--Principal--------------------------------------------------------------------------------

**public** **static** **void** main(String[] args) **throws** IOException

{

System.***out***.println("Bem vindo Ao sistema de Ordenamento");

Methods calc = **new** Methods();

calc.iniciar();

} }