UNIVERSIDADE PAULISTA

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

Atividade Prática Supervisionada (APS)

Ciência da Computação

Integrantes do grupo:

Jose Vitor da Silva - RA: N652535 – CC4A40

Juliana da Silva Dantas - RA: N579919 – CC4A40

Laura Busin Campos - RA: N6184D1 – CC4A40

Thiago Muller Rodrigues - RA: F1383F0 – CC4A40

SÃO PAULO

2021

ÍNDICE

1. OBJETIVO 2
2. INTRODUÇÃO 2
3. REFERENCIAL TEÓRICO 5
   3.1. Insertion Sort 5
   3.2. Merge Sort 7
   3.3. Bubble Sort 9
4. DESENVOLVIMENTO 10
   4.1. Main.c 10
   4.2. Main.h 12
   4.3. Abertura.c 13
   4.4. Vetor.c 14
   4.5. Cronômetro.c 19
   4.6. MergeSort.c 20
   4.7. BubbleSort.c 21
   4.8. InsertionSort.c 22
5. RESULTADOS E DISCUSSÃO 22
6. CONSIDERAÇÕES FINAIS 27
7. REFERÊNCIAS BIBLIOGRÁFICAS 28
CÓDIGO FONTE 30
FICHAS DE ATIVIDADES PRÁTICAS SUPERVISIONADAS 40

1. OBJETIVO

O objetivo do presente trabalho é criar um programa em C utilizando todos os conceitos aprendidos durante as aulas da disciplina de Estrutura de Dados (ED) com a finalidade de desenvolver um sistema computacional completo utilizando três métodos de ordenação de dados.

2. INTRODUÇÃO

O ato de ordenar está presente em nossas vidas desde sempre, é um ato tão corriqueiro, que acabamos nem percebendo como ordenamos tudo em nossas vidas, nomes em listas de chamadas, números em ordem crescente ou decrescente, cores em degradê, e muitas outras coisas. Na vida de um cientista da computação, o ato de ordenação pode ser ainda mais corriqueiro.

Normalmente, o usuário que vai inserir os dados em um programa, não está preocupado com a ordem de entrada destes dados, pois isso fica a cargo do sistema de acordo com a necessidade de utilização. Por isso, normalmente encontramos elementos armazenados de forma aleatória dentro do sistema. Mas, na hora de utilizar esses dados, muitas vezes é preciso ordená-los para facilitar a sua visualização e assim encontrar o dado desejado mais facilmente. Encontrar elementos em uma lista torna-se simples quando os dados estão ordenados (PUGA & RISSETTI, 2009).

Algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem. Ou seja, ordena uma série de dados de forma completa ou parcial para facilitar a recuperação desses dados em uma lista (HONORATO, 2013).

Foram desenvolvidos diversos algoritmos de ordenação que realizam comparações sucessivas e trocam os elementos de posição (PUGA & RISSETTI, 2009). Os principais algoritmos de ordenação existentes são: Insertion sort, Selection sort, Bubble sort, Quick sort, Merge sort, Heap sort (HONORATO, 2013).

O **Insertion Sort** ou **Ordenação por Inserção**, pode ser considerado um método de ordenação simples, e por isso é mais adequado para vetores menores e normalmente são programas mais fáceis de entender (VIANA, 2017).

As comparações serão iniciadas pelo segundo número do vetor. Conforme as iterações vão acontecendo, os números à esquerda do número eleito (que está sendo

comparado com todos) estão sempre ordenados de forma crescente ou decrescente (ASCENCIO & ARAÚJO, 2010).

O funcionamento do algoritmo é bem simples: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo no local apropriado de acordo com o critério de ordenação (VIANA, 2017).

O **Selection Sort** ou **Ordenação por Seleção**, também pode ser considerado um dos métodos de ordenação mais simples e consiste nos seguintes passos: Selecionar o menor elemento da sequência; Trocá-lo com o que está na primeira posição; Repetir as operações anteriores até o final dos elementos (PUGA & RISSETTI, 2009).

Para a ordenação em ordem crescente, cada número do vetor, a partir do primeiro, será ativado e comparado com o menor número dentre os que estão à direita do ativo. Quando um número menor que o ativo for encontrado, ocorrerá a troca de posições entre eles, e dessa forma, os números à esquerda do número ativo sempre estarão em ordem (ASCENCIO & ARAÚJO, 2010).

A técnica procura o menor consumo de memória, evitando a criação de um novo vetor ordenado (PUGA & RISSETTI, 2009).

O **Bubble Sort** ou **Ordenação por Trocas**, é também um algoritmo muito simples, mas acaba sendo menos eficiente. Por causa da sua forma de execução, o vetor terá que ser percorrido quantas vezes forem necessárias, tornando o algoritmo ineficiente para listas muito grandes (HONORATO, 2013).

Consiste em comparar pares consecutivos de valores e trocá-los caso estejam fora de ordem. O algoritmo determina uma sequência de comparações sistemáticas que passam por toda a sequência de dados como um todo, sempre fazendo com que o menor valor acabe no início da sequência, e assim se inicie uma nova série de comparações.

Em cada passagem, um elemento é deslocado para o topo da lista. Então, a cada iteração um novo elemento é ordenado. O deslocamento do elemento menor para a posição inicial é feito por comparações e trocas sucessivas, iniciando-se pelos penúltimo e último. Se o último for menor que o penúltimo, eles são trocados, senão, permanecem na mesma posição. Depois a comparação é feita entre o penúltimo e o antepenúltimo, e assim vai até chegar ao topo. Desse modo, ao mesmo tempo que os menores valores são jogados para cima, os maiores vão sendo jogados para o fim da lista (PUGA & RISSETTI, 2009).

O **Quick Sort** ou **Ordenação Rápida**, pode ser considerado eficiente e mais complexo nos detalhes do que os citados anteriormente. Normalmente é utilizado para trabalhar uma quantidade maior de dados (VIANA, 2017). Esse método trabalha com o conceito de pivô, onde um elemento do conjunto é selecionado para servir como referência para ser comparado com os outros valores (BORIN, 2020).

A estrutura é dividida em duas partes, à esquerda do pivô, com apenas números menores, e à direita do pivô com apenas números maiores. Uma nova instância da função de ordenação é aberta sempre que um novo pivô precisa ser encontrado (BORIN, 2020).

O pivô inicialmente é o valor do meio do vetor, o algoritmo irá primeiro compará-lo com os valores à sua esquerda e ver se os valores são realmente menores, depois irá compará-lo com os valores à direita para ver se são realmente maiores que ele. Caso algum valor da esquerda seja maior que o pivô, ou um valor da direita seja menor, eles trocam de posição e o outro número passa a ser o pivô (BORIN, 2020).

As varreduras em ambos os lados continuam até que as duas atinjam a posição do pivô, nesse momento uma instância da ordenação é encerrada e o novo pivô será escolhido com base no ponto de parada na iteração anterior (BORIN, 2020).

A principal desvantagem deste método é que ele possui uma implementação difícil e delicada, um pequeno engano pode gerar efeitos inesperados para determinadas entradas de dados (VIANA, 2017).

O **Merge Sort** ou **Ordenação por Intercalação**, assim como o Quick Sort, pode ser considerado um método eficiente e complexo (VIANA, 2017). Esse método se baseia no conceito de “dividir para conquistar”, pois divide o problema em pedaços menores, resolve cada pedaço e depois junta (merge) os resultados. O vetor será dividido em duas partes iguais, que serão cada uma dividida em duas partes, e assim até que restem partes indivisíveis (VIANA, 2017).

Quando duas partes subsequentes atingem tamanho um, elas fazem a intercalação, ou seja, é feita a ordenação seguida da agregação das partes para formar um vetor maior (BORIN, 2020). Para juntar as partes ordenadas, os dois elementos de cada parte são separados e o menor deles é selecionado e retirado de sua parte. Em seguida os menores entre os restantes são comparados e assim se prossegue até juntar as partes (VIANA, 2017).

O **Heap Sort** ou **Ordenação por Heaps**, é um método eficiente e complexo, assim como os dois últimos citados (VIANA, 2017). Esse método tem duas fases:

A primeira transforma o vetor em heap, ou seja, uma estrutura de dado que enxerga um vetor como se fosse uma árvore binária, onde o valor de todo pai é maior ou igual ao valor de cada um dos dois filhos (FEOFILOFF, 2018).

Já a segunda parte, rearranja o heap em ordem crescente. Um array é criado removendo repetidamente o maior elemento do heap (a raiz do heap) e inserindo-o no array. O heap é atualizado após cada remoção para manter a propriedade do heap. Depois que todos os objetos foram removidos do heap, o resultado é uma matriz ordenada (FEOFILOFF, 2018).

No presente trabalho decidimos pesquisar e desenvolver os métodos de ordenação *Insertion Sort*, *Merge Sort* e *Bubble Sort*.

3. REFERENCIAL TEÓRICO

3.1. Insertion Sort

Propriedades e Análise de eficiência:

O Insertion Sort é estável, in-place e O(n2) (BRUNET, 2019). A propriedade que tem relação a ordem dos valores iguais no array original é a estabilidade, por exemplo, se existir dois valores 46 no array antes da ordenação, depois da execução, esses valores devem seguir a ordem inicial, ou seja, a primeira ocorrência do 46, deve vir antes da segunda ocorrência do 46. As trocas sempre são feitas com os valores vizinhos, sendo afastados um a um, por isso que um elemento nunca trocará de posição com o elemento do mesmo valor (VIANA, 2017).

O Insertion Sort é in-place porque é feito o rearranjando dos elementos no próprio array, ao invés de usar outros arrays ou estruturas auxiliares.

O pior caso de ser usado: em ordem reversa (ordem descrente), pois toda tentativa de inserção ordenada deve percorrer o array todo à esquerda trocando os elementos até encaixar o atual na primeira posição (ASCENCIO & ARAÚJO, 2010).

Esse método não é um algoritmo eficiente para grandes entradas, nesses casos existem outras alternativas como Quick Sort e Merge Sort, além de alternativas lineares como o Counting Sort (BRUNET, 2019).

Traçando o paralelo entre o Selection Sort e o Insertion Sort. O Selection efetua menos trocas do que o Insertion, pois há uma troca apenas por iteração, ou seja, no total o Selection Sort efetua n trocas. Já o insertion sort efetua ao menos uma troca

por iteração, pois deve efetuar trocas para afastar cada elemento avaliado (BRAGA, 2018).

Porém o Insertion Sort efetua menos comparações do que o Selection Sort, pois nem sempre o elemento a ser inserido de forma ordenada deve ir até o final. Na verdade, isso só acontece no pior dos casos, em que o array está ordenado em ordem reversa (BRAGA, 2018).

Na teoria, Insertion Sort, Selection Sort e Bubble Sort estão na mesma classe de complexidade, qual seja O(n2). Na prática, o Insertion Sort apresenta o melhor desempenho entre esses 3 algoritmos (BRAGA, 2018).

Funcionamento:

Esse método percorre um vetor de elementos da esquerda para direita e à medida em que vai avançando ele ordena os elementos à esquerda. Ele é considerado um método de ordenação estável (se a ordem relativa dos itens não se altera durante a ordenação).

O funcionamento é simples, consiste em selecionar a partir do segundo elemento, o próximo item da sequência e colocá-lo no local apropriado. Ele percorre todo o vetor uma única vez, porém para fazer o ordenamento ele utiliza outro laço interno (ASCENCIO & ARAÚJO, 2010).

A seguir mostraremos um exemplo do funcionamento desse algoritmo retirado do site DEVFURIA (2012):

Vetor incial → v = {5, 3, 2, 4, 7, 1, 0, 6}

Inicia-se com o segundo valor do vetor, ou seja, o de índice 1, no caso o valor 3.

```
 5 [3] 2  4  7  1  0  6
(5  3) 2  4  7  1  0  6   compara com o valor anterior
 3--5  2  4  7  1  0  6   troca
```

Na segunda iteração usa o próximo valor (2) e compara com os valores anteriores.

```
 3  5 [2] 4  7  1  0  6
 3 (5  2) 4  7  1  0  6   compara com o valor anterior
 3  2--5  4  7  1  0  6   troca
(3  2) 5  4  7  1  0  6   compara com o valor anterior
 2--3  5  4  7  1  0  6   troca
```

Na terceira iteração pega o valor 4 e, novamente, compara com os valores anteriores.

```
 2  3  5 [4] 7  1  0  6
```

```
 2  3 (5  4) 7  1  0  6   compara com o valor anterior
 2  3  4--5  7  1  0  6   troca
 2 (3  4) 5  7  1  0  6   compara com o valor anterior
```

Na quarta iteração, o valor chave (7) é maior que o valor anterior, não ocorrendo troca.

```
 2  3  4  5 [7] 1  0  6
 2  3  4 (5  7) 1  0  6   não troca
```

Na quinta iteração, pega o 1 e leva até o início do vetor.

```
 2  3  4  5  7 [1] 0  6
 2  3  4  5 (7  1) 0  6   compara com o valor anterior
 2  3  4  5  1--7  0  6   troca
 2  3  4 (5  1) 7  0  6   pega o par anterior
 2  3  4  1--5  7  0  6   troca
 2  3 (4  1) 5  7  0  6   pega o par anterior
 2  3  1--4  5  7  0  6   troca
 2 (3  1) 4  5  7  0  6   pega o par anterior
 2  1--3  4  5  7  0  6   troca
(2  1) 3  4  5  7  0  6   pega o par anterior
 1--2  3  4  5  7  0  6   troca
```

Na sexta iteração, é preciso levar o 0 (zero) até o início do vetor.

```
 1  2  3  4  5  7 [0] 6
 1  2  3  4  5 (7  0) 6   compara com o valor anterior
 1  2  3  4  5  0--7  6   troca
 1  2  3  4 (5  0) 7  6   pega o par anterior
 1  2  3  4  0--5  7  6   troca
 1  2  3 (4  0) 5  7  6   pega o par anterior
 1  2  3  0--4  5  7  6   troca
 1  2 (3  0) 4  5  7  6   pega o par anterior
 1  2  0--3  4  5  7  6   troca
 1 (2  0) 3  4  5  7  6   pega o par anterior
 1  0--2  3  4  5  7  6   troca
 (1 0) 2  3  4  5  7  6   pega o par anterior
  0--1  2  3  4  5  7  6   troca
```

Na sétima e última iteração realiza apenas uma troca.

```
 0  1  2  3  4  5  7 [6]
 0  1  2  3  4  5 (7  6)  compara com o valor anterior
 0  1  2  3  4  5  6--7   troca
 0  1  2  3  4 (5  6) 7   não troca
```

3.2. Merge Sort

Merge Sort é um método de ordenação em programação criado pelo matemático John Von Neumann em 1945 (VIANA, 2017). Esse método tem como objetivo “Dividir para Conquistar”, ou seja, sua ideia é criar uma sequência ordenada

a partir de outras, primeiro ela divide a sequência original em pares de dados, ordena e depois agrupa em sequência de quatro elementos até ter a sequência dividida em duas partes (FARIAS, 2014). É um método estável e possui complexidade C(n) = O(n log n) para todos os casos independente da composição inicial do vetor (VIANA, 2017).

Esse algoritmo possui três passos principais para conseguir fazer a ordenação (ASCENCIO & ARAÚJO, 2010):

1. Dividir: Divide os dados em subsequências pequenas.
2. Conquistar: Soluciona as subsequências recursivamente.
3. Combinar: Junte as subsequências em um único conjunto classificado.

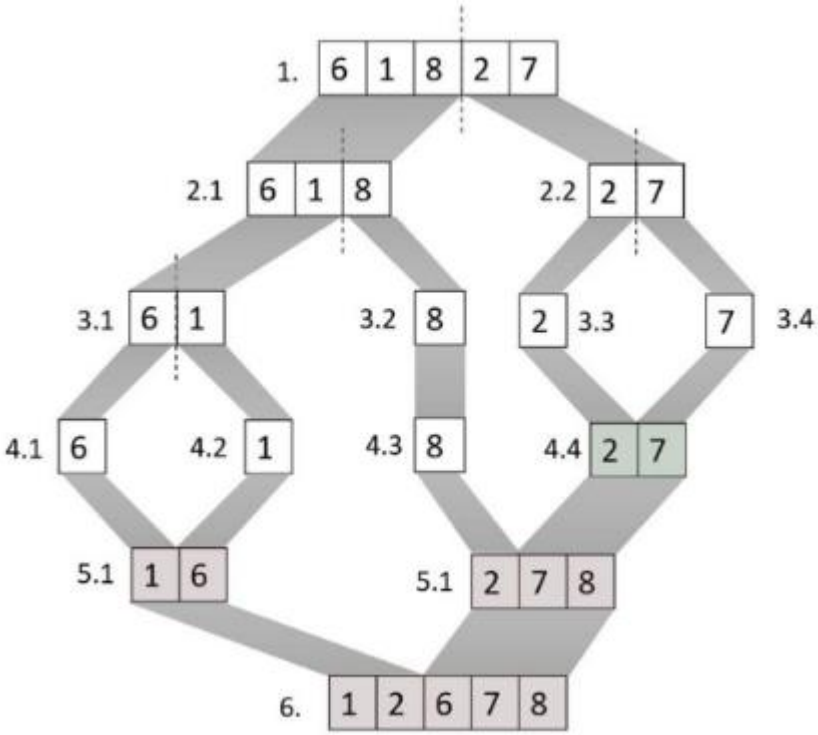


Figura 1 - Ilustração do funcionamento do Algoritmo Merge Sort retirada do Livro Estrutura de Dados (BORIN, 2020).

Quando comparado a outros algoritmos que utilizam esse mesmo método de divisão e conquista, como o Quicksort, o Merge Sort possui a mesma complexidade. Mas, quando comparado a algoritmos mais básicos de ordenação por comparação e troca (bubble, insertion e selection sort), o Merge Sort é mais rápido e eficiente quando é utilizado sobre uma grande quantidade de dados (FELIPE, 2018).

A desvantagem do Merge Sort é que ele é mais complexo por usar funções recursivas e requer muita memória, ou seja, precisa de um vetor com as mesmas dimensões do vetor que está sendo classificado para poder armazenar o vetor ordenado para cada chamada recursiva (FELIPE, 2018).

3.3. Bubble Sort

Bubble Sort é um método de ordenação em algoritmo que tem como objetivo ordenar valores em forma crescente e decrescente. Ela trabalha comparando a posição atual com a próxima posição e, se a posição atual for maior que a posição posterior é realizada a troca de valores nessa posição, apenas passa-se para o próximo par de comparações (BORIN, 2020).

Na forma crescente ela compara a posição atual com a próxima posição posterior e, se a posição atual for menor que a posição posterior, é realizada a troca, caso contrário, a troca não é feita e passa (GATTO, 2017).

O algoritmo Bubble sort, recebeu este nome pela ideia de que os elementos menores são mais leves, e sobem como bolhas até suas posições corretas, enquanto os mais pesados permanecem no fundo como pedras (PUGA & RISSETTI, 2009).

No melhor caso, quando o vetor está ordenado, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vetor. No pior caso, quando o vetor está em ordem inversa, são feitas n^2 operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados (ASCENCIO & ARAÚJO, 2010).

Posição	Valores iniciais	Iteração 1	Iteração 2	Iteração 3	Iteração 4	Iteração 5	Iteração 6	Iteração 7
1	23	4	4	4	4	4	4	4
2	4	23	12	12	12	12	12	12
3	33	12	23	19	19	19	19	19
4	45	33	19	23	23	23	23	23
5	19	45	33	28	28	28	28	28
6	12	19	45	33	33	33	33	33
7	28	28	28	45	40	40	40	40
8	40	40	40	40	45	45	45	45

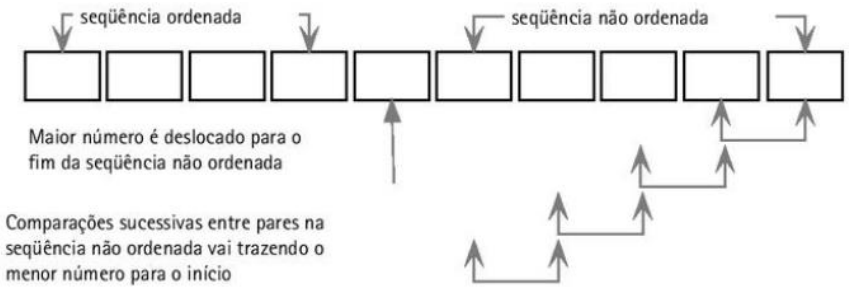


Figura 2 - Ilustração do funcionamento do Algoritmo Bubble Sort retirada do Livro Lógica de Programação e Estrutura de Dados: com aplicações em Java (*PUGA & RISSETTI, 2009*).

O algoritmo de Bubble Sort é fácil de escrever independente da linguagem de programação, e bem compreendido, os dados que são ordenados no lugar tem um fator de ter pouca sobrecarga de memória e uma vez ordenado, os dados estão na memória, preparado para processamento. A única desvantagem é que leva muito tempo para classificar. O tempo médio aumenta quase exponencialmente à medida que o número de elementos aumenta.

4. DESENVOLVIMENTO

A estrutura fora dívida em um total de oito arquivos, afim de uma melhor organização, sendo as divisões: Main.c, Main.h, Abertura.c,Vetor.c, Cronometro.c, MergeSort.c, BubleSort.c, Insertion.c.

4.1. Main.c

É a estrutura principal, ela utiliza alguns dados/comandos já definidos nos outros arquivos.

Setlocale (LC_ALL, " portuguese ")
Habilita caracteres especiais e acentos pertencentes a língua portuguesa.

Abertura (1)
Dá o valor para que a abertura do programa possa ser executada, essa feita no arquivo Abertura.c.

VETOR v

Gera o vetor “v”, produzido no arquivo Vetor.c.

Int escolha, escolhaCompararVal

Permite a entrada de um número inteiro, para a escolha do método a ser utilizado e se deseja comparações de valores.

InicializaVetor (& escolha, & v);

Inicializa o vetor.

OrganizaVetor (& escolha, & v);

Organiza o vetor a partir do método escolhido.

ComparaVetores (& escolhaCompararVal, & escolha, & v);

Compara o tempo que os vetores são organizados.

Obs: “escolha”, “v” e “escolhaCompararVal” são dados já definidos no arquivo Vetor.c e o “&” resulta o endereço da memória em questão.

LiberaMem (&v);

Limpa a memória.

printf (" \ n Você deseja usar outro vetor? \ n 1) Sim \ n 2) Não \ n \ n Escolha: ");
scanf (" % d " , & acabou)

Permite entrar com um valor inteiro, para que seja possível a escolha entre encerrar o programa ou utilizar outro vetor.

```
if (acabou == 1 ) {
    printf ( " \ n Tudo bem, até mais!: 3 \ n \ n " );
} else {
    printf ( " \ n Valor inválido! \ n \ n " );
```

Caso tenha escolhido encerrar o programa o número escolhido não seja o dado entre as opções, ele declara que o valor é invalido.

4.2. Main.h

É a biblioteca que armazena as variáveis.

void abertura ();
Abertura

struct vetor { int * vetor [3], tamanhoDoVetor;};
vetor de estrutura de typedef VETOR;
Os dados são armazenados na mesma entidade, tornando apenas uma variável, no caso “vetor” e também é definido que o tipo de dado que será usado é o vetor.

void qualTamanhoDoVetor (int * tamanhoDoVetor);
Permite a entrada de quantos itens terá no vetor.

void geraAleatorio (int tamanhoDoVetor, int vetor []);
Gera um vetor aleatório.

void leDados (int tamanhoDoVetor, int vetor []);
Lê os itens do vetor.

void printVetor (int vetor [], int tamanhoDoVetor);
Mostra o vetor feito.

void printMerge (int vetor [], int tamanhoDoVetor);
Mostra o vetor.

void executaBubbleSort (int vetor [], int tamanhoDoVetor);
Executa o método Bubble Sort .

void executaInsertionSort (int vetor [], int tamanhoDoVetor);
Executa o método Insertion sort.

Void executaMergeSort (int vetor [], int tamanhoDoVetor);
Executa o método Merge Sort.

void inicializaVetor (int * escolha, VETOR * v);
Inicializa o vetor a partir da escolha.

void organizaVetor (int * escolha, VETOR * v);
Organiza o vetor de acordo com a escolha do método.

void comparaVetores (int * escolhaCompararVal, int * escolha, VETOR * v);
Permite escolher se terá comparação entres os vetores.

void liberaMem (VETOR * v);
Libera a memória.

void printTempo (int escolha);
Mostra o tempo de execução do método.

void printTempos ();
void contagemCronometro (int escolha, unsigned long inicio, unsigned long fim);
Mostra o tempo de execução de todos os três métodos para que possa ser comparado.

void bubbleSort (int * vetor, int tamanhoDoVetor);
troca nula (int * a, int * b);
Organização pelo método bubble Sort.

void insertionSort (int vetor [], int tamanhoDoVetor);
Organização pelo método Insertion Sort.

void merge (int vetor [], int comeco, int meio, int fim);
void mergeSort (int vetor [], int comeco, int fim);
Organização pelo método Merge Sort.

4.3. Abertura.c

Consiste em fazer a abertura gráfica do programa. Onde existe a função abertura e é permitido entrar com valores inteiros, se esse valor for igual a 1, será impresso o contorno das letras, e caso esse valor for igual a 2, as letras serão preenchidas. No final é impresso “ALGORITMO DE ORDENAÇÃO”.

4.4. Vetor.c

Armazena as funções dos vetores.

```
void  qualTamanhoDoVetor ( int * tamanhoDoVetor) {
   printf ( " \ n Insira uma quantidade de números dentro do vetor: \ n " );
        scanf ( " % d " , tamanhoDoVetor);
```

Pergunta a quantidade de valores dentro do vetor e recebe os números de itens solicitados.

```
void  geraAleatorio ( int tamanhoDoVetor, int vetor []) {
        int limiteCasasAleatorios = 100 ; // 100 = de 0 a 99.
        srand ( tempo ( 0 ));
        para ( int i = 0 ; i <tamanhoDoVetor; i ++) {
      vetor [i] = rand ()% limiteCasasAleatorios;
```

A função srand gera valores aleatórios com base no tempo para não os repetir.

```
void  leDados ( int tamanhoDoVetor, int vetor []) {
   printf ( " \ n Insira os valores dentro do vetor: \ n " );
        para ( int i = 0 ; i <tamanhoDoVetor; i ++) {
      scanf ( " % d " , & vetor [i])
```

Caso se solicite a opção de escolher os valores do vetor, a função irá receber os valores que o usuário forneceu.

```
void  printVetor ( int vetor [], int tamanhoDoVetor) {
        para ( int i = 0 ; i <tamanhoDoVetor; i ++) {
      printf ( " % d  " , vetor [i]);
        }
   printf ( " \ n " );
}
```

Mostra o vetor fornecido pelo usuário.

```
void  printMerge ( int vetor [], int tamanhoDoVetor) {
        para ( int i = 1 ; i <= tamanhoDoVetor; i ++) {
      printf ( " % d  " , vetor [i]);
        }
   printf ( " \ n " );
}
void  executaBubbleSort ( int vetor [], int tamanhoDoVetor) {
        clock_t inicioCronometro = clock ();
   bubbleSort (vetor, tamanhoDoVetor - 1 ); // (função em BubbleSort.c)
   clock_t fimCronometro = clock ();
   contagemCronometro ( 0 , inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}
void  executaInsertionSort ( int vetor [], int tamanhoDoVetor) {
        clock_t inicioCronometro = clock ();
   insertionSort (vetor, tamanhoDoVetor); // (Função em InsertionSort.c)
   clock_t fimCronometro = clock ();
   contagemCronometro ( 1 , inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}
void  executaMergeSort ( int vetor [], int tamanhoDoVetor) {
   clock_t inicioCronometro = clock ();
   mergeSort (vetor, 0 , tamanhoDoVetor); // (Função em MergeSort.c)
   clock_t fimCronometro = clock ();
   contagemCronometro ( 2 , inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}
void  inicializaVetor ( int * escolha, VETOR * v) {
   qualTamanhoDoVetor (& v-> tamanhoDoVetor );
```

Executa o método escolhido, inicia a contagem dos cronômetros, organiza o vetor, pausa o cronometro quando o vetor está da maneira desejada e mostra o tempo da execução.

Obs: O print do merge sort precisa ser diferente, pois a função sempre gera um número 0 no final do vetor e este valor é ordenado junto com os outros. Para corrigir, iniciamos a variável do loop em 1 para pular o primeiro valor e dizemos que o loop só vai finalizar quando i <= que o tamanhoDoVetor.

```
for ( int i = 0 ; i < sizeof (v-> vetor ) / sizeof (v-> vetor [ 0 ]); i ++) {
v-> vetor [i] = ( int *) malloc ( sizeof ( int ) * v-> tamanhoDoVetor + 1 );
}
```

Alocação dinâmica de memória.

```
printf ( " \ n Você deseja usar números aleatórios ou escolher os valores de serem ordenados? \ n 1) Aleatórios \ n 2) Escolher os valores \ n \ n Digite: " );
scanf ( " % d " , escolha);

switch (* escolha) {
caso 1 :
geraAleatorio (v-> tamanhoDoVetor , v-> vetor [ 0 ]);
pausa ;
caso 2 :
leDados (v-> tamanhoDoVetor , v-> vetor [ 0 ]);
pausa ;
padrão :
printf ( " Erro! Valor inválido! \ n " );
saída ( 1 );
}
```

Permite a entrada de valores e traz a opção de criar um vetor aleatório ou escolher os valores, caso o número digitado seja diferente doa das opções ele declara o valor com invalido.

```
para ( int i = 0 ; i <= v-> tamanhoDoVetor ; i ++) {
v-> vetor [ 2 ] [i] = v-> vetor [ 1 ] [i] = v-> vetor [ 0 ] [i];
}
}

void organizaVetor ( int * escolha, VETOR * v) {
```

```
   printf ( " \ n Qual algorítmo de ordenação você deseja usar? \ n 1) Bubble Sort \ n 2)
Insertion Sort \ n 3) Merge Sort \ n \ n Digite: " );
        scanf ( " % d " , escolha);

        if (* escolha == 1 ) {
        printf ( " \ n \ n - CLASSIFICAÇÃO DA BOLHA - \ n " );
        } else  if (* escolha == 2 ) {
     printf ( " \ n \ n - CLASSIFICAÇÃO DE INSERÇÃO - \ n " );
        } else  if (* escolha == 3 ) {
     printf ( " \ n \ n - MERGE SORT - \ n " );
        } else {
     printf ( " \ n Erro! Opção Inválida! \ n " );
     saída ( 1 );
        };

   printf ( " \ n \ 033 [1; 31mVetor antes da ordenação: \ n " );
   printVetor (v-> vetor [ 0 ], v-> tamanhoDoVetor );

   printf ( " \ n \ 033 [0; 32mVetor depois da ordenação: \ n " );

   switch (* escolha) {
   caso  1 :
      executaBubbleSort (v-> vetor [ 0 ], v-> tamanhoDoVetor );
      printVetor (v-> vetor [ 0 ], v-> tamanhoDoVetor );
      printTempo (* escolha); // (Função em cronometro.c)
      pausa ;
   caso  2 :
      executaInsertionSort (v-> vetor [ 1 ], v-> tamanhoDoVetor );
      printVetor (v-> vetor [ 1 ], v-> tamanhoDoVetor );
      printTempo (* escolha); // (Função em cronometro.c)
      pausa ;
   caso  3 :
      executaMergeSort (v-> vetor [ 2 ], v-> tamanhoDoVetor );
      printMerge (v-> vetor [ 2 ], v-> tamanhoDoVetor );
      printTempo (* escolha); // (Função em cronometro.c)
      pausa ;
        }
```

}

Mostra o vetor antes da organização, permite escolher o método que será usado, cronometra o tempo de duração da execução do método e mostra o vetor depois da execução.

Obs: Há uma cópia do vetor desordenado para os outros 2 vetores. Por exemplo: o vetor que tem 5 valores gerados aleatoriamente (Lembrando que começa em [0] e o valor do último [5] é um "\ 0"). Então, na memória, o primeiro vetor ficará: vetor [0] [5] = {21, 23, 43, 2, 5}, e será copiado para os vetores [1] [5] e vetor [2] [5].

```
void  comparaVetores ( int * escolhaCompararVal, int * escolha, VETOR * v) {
   printf ( " Você deseja comparar os tempos de cada algoritmo para o mesmo vetor? \
n 1) Sim \ n 2) Não \ n \ n Escolha: " );
        scanf ( " % d " , escolhaCompararVal);

        if (* escolhaCompararVal == 1 ) {
      switch (* escolha) {
        caso  1 :
        // Caso tenha escolhido o tipo bolha
        executaInsertionSort (v-> vetor [ 1 ], v-> tamanhoDoVetor );
        executaMergeSort (v-> vetor [ 2 ], v-> tamanhoDoVetor );
        printTempos (); // (Função em cronometro.c)
        pausa ;
      caso  2 :
        // Caso tenha escolhido o tipo de inserção
        executaBubbleSort (v-> vetor [ 0 ], v-> tamanhoDoVetor );
        executaMergeSort (v-> vetor [ 2 ], v-> tamanhoDoVetor );
        printTempos (); // (Função em cronometro.c)
        pausa ;
      caso  3 :
        // Caso tenha escolhido o merge sort
        executaBubbleSort (v-> vetor [ 0 ], v-> tamanhoDoVetor );
        executaInsertionSort (v-> vetor [ 1 ], v-> tamanhoDoVetor );
        printTempos (); // (Função em cronometro.c)
        pausa ;
      padrão :
```

```
        printf ( " \ n Erro! Opção Inválida! \ n " );
        saída ( 1 );
        }
        }
}
```

Permite entrar com um número equivalente a uma opção e caso queira comparar o tempo de execução de cada método, o programa organizará o vetor novamente, computando apenas o tempo gasto.

```
void  liberaMem (VETOR * v) {
        for ( int i = 0 ; i < sizeof (v-> vetor ) / sizeof (v-> vetor [ 0 ]); i ++) {
      livre (v-> vetor [i]);
        }
}
```

Libera a memória alocada de modo dinâmico.

4.5. Cronômetro.c

Cronometra o tempo de execução dos métodos.

```
void  printTempo ( int escolha) {
   printf ( " \ n \ 033 [0; 37mTempo de processamento da ordenação: % .5lf ms \ n \ n " , algoritmos [escolha - 1 ]);
}
void  printTempos () {
   printf ( " \ n Tempo de processamento da ordenação Bubble Sort: % .5lf ms \ n " , algoritmos [ 0 ]);
   printf ( " Tempo de processamento da ordenação Insertion Sort: % .5lf ms \ n " , algoritmos [ 1 ]);
   printf ( " Tempo de processamento da ordenação Merge Sort: % .5lf ms \ n " , algoritmos [ 2 ]);
}
```

Mostra o tempo gasto no processamento dos métodos.

Obs:O algoritmos [0] sempre guardará o tempo do Bubble Sort, [1] sempre guardará o tempo do Insertion Sort e [2] sempre guardará o tempo do Merge Sort.

```
void  contagemCronometro ( int escolha, unsigned  long inicio, unsigned  long fim) {
   algoritmos [escolha] = ( duplo ) (fim - inicio) / (CLOCKS_PER_SEC / 1000 );
```

Cálculo do tempo gasto pela função de ordenação com base nos tempos de iniciais e finais.

4.6. MergeSort.c

Sintaxe do método Merge Sort.

```
void  merge ( int vetor [], int comeco, int meio, int fim) {
      int com1 = comeco, com2 = meio + 1 , comAux = 0 , tam = fim-comeco + 1 ;
      int * vetAux;
      vetAux = ( int *) malloc (tam * sizeof ( int ));
      while (com1 <= meio && com2 <= fim) {
    if (vetor [com1] <vetor [com2]) {
      vetAux [comAux] = vetor [com1];
      com1 ++;
      } else {
      vetAux [comAux] = vetor [com2];
      com2 ++;
      }
    comAux ++;
      }
      while (com1 <= meio) {   // Caso ainda haja elementos na primeira metade
        vetAux [comAux] = vetor [com1];
    comAux ++;
    com1 ++;
      }
      while (com2 <= fim) {          // Caso ainda haja elementos na segunda metade
    vetAux [comAux] = vetor [com2];
    comAux ++;
    com2 ++;
      }
```

```
        for (comAux = comeco; comAux <= fim; comAux ++) {     // Mover os elementos
de volta para o vetor original
      vetor [comAux] = vetAux [comAux-comeco];
        }
        grátis (vetAux);
}
void  mergeSort ( int vetor [], int comeco, int fim) {
        if (comeco <fim) {
      int meio = (fim + comeco) / 2 ;
      mergeSort (vetor, comeco, meio);
      mergeSort (vetor, meio + 1 , fim);
      fusão (vetor, comeco, meio, fim);
        }
}
```

4.7. BubbleSort.c

Sintaxe do método Bubble Sort.

```
nula  troca ( int * a, int * b) {
        int temp = * a;
        * a = * b;
        * b = temp;
}
void  bubbleSort ( int * vetor, int tamanhoDoVetor) {

        if (tamanhoDoVetor < 1 ) {
      retorno ;
        }
        para ( int i = 0 ; i <tamanhoDoVetor; i ++) {
      if (vetor [i]> vetor [i + 1 ]) {
        troca (& vetor [i], & vetor [i + 1 ]);
        }
        }
   bubbleSort (vetor, tamanhoDoVetor - 1 );
}
```

4.8. InsertionSort.c

Sintaxe do método Insertion Sort.

```
void  insertionSort ( int vetor [], int tamanhoDoVetor) {
        int i, j, valorDovetor;
        para (i = 1 ; i <tamanhoDoVetor; i ++) {
      valorDovetor = vetor [i];
        j = i - 1 ;
      while (j> = 0 && vetor [j]> valorDovetor) { // Enquanto j> = 0 e o numero j do vetor> valorDovetor, faça:
         vetor [j + 1 ] = vetor [j];
        j = j - 1 ;
        }
        vetor [j + 1 ] = valorDovetor;
        }
```

5. RESULTADOS E DISCUSSÃO

Os métodos de ordenação escolhidos para fazer esse estudo foram o Bubble Sort, o Insertion Sort e o Merge Sort. Os testes foram realizados utilizando vetores de diferentes tamanhos, de cinco até 50.000 algarismos que variavam de 0 a 100. Os vetores utilizados eram sempre os mesmos entre os diferentes métodos para evitar vieses na hora da comparação. Em cada uma das ordenações utilizando os diferentes métodos foram registrados o tempo de processamento em milissegundos (ms). Os resultados destes testes estão listados na Tabela 1 e retratados na Figura 3. É importante lembrar que o tempo de execução do algoritmo vai depender do processador que está sendo utilizado, podendo variar de acordo com o computador onde está sendo executado.

Como é possível observar, em todos os métodos o tempo de processamento aumentou de acordo com o tamanho de vetor que foi ordenado (Tabela 1 e Figura 3). Além disso, em todos os métodos o tempo de ordenação foi tão pequeno para os vetores de 5 a 100 algarismos que não foi possível medir com precisão e por isso foram registrados como 0.

Tabela 1 - Resultados dos testes para cada um dos métodos de ordenação, em milissegundos (ms), usando diferentes tamanhos de vetores.

Tamanho Vetor	Tempo de processamento (ms)		
	Bubble	Insertion	Merge
5	0	0	0
10	0	0	0
100	0	0	0
500	3	1	1
1.000	14	2	2
5.000	314	22	7
10.000	724	66	15
15.000	2.260	274	22
25.000	5.638	667	39
50.000	24.025	2.991	68

Observando a Tabela 1 é possível reparar que o Bubble Sort é o método que exige um maior tempo de processamento parar realizar a ordenação, seguido do método Insertion Sort e do método Merge Sort. O tempo de processamento de ordenação começa a se diferenciar mais a partir do vetor com 5.000 algarismos, mas essa diferença fica mais clara a partir do vetor 10.000 para o método Bubble Sort e a partir do vetor 25.000 para o método Insertion Sort (Figura 3).

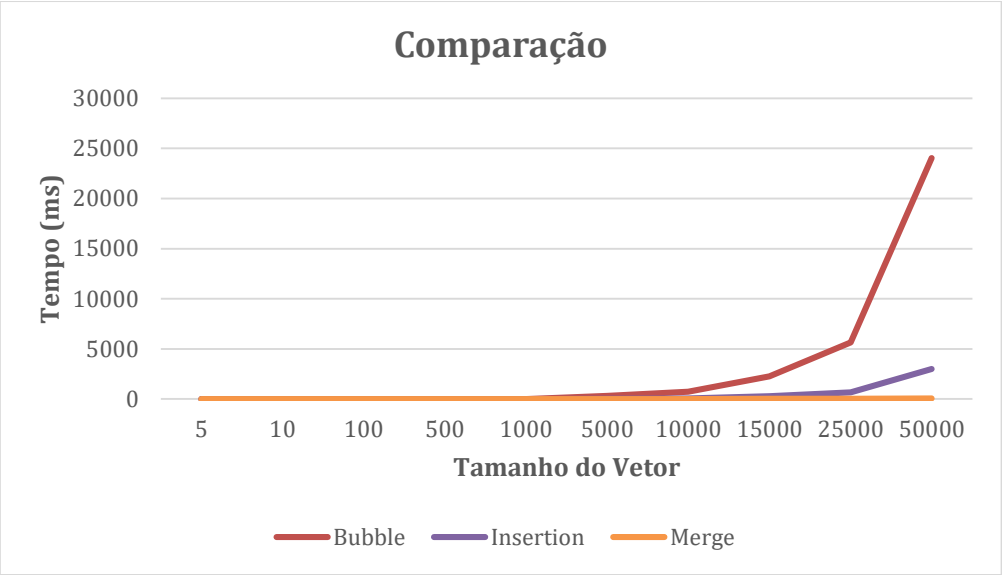


Figura 3 - Resultados dos testes para cada um dos métodos de ordenação, em milissegundos (ms), usando diferentes tamanhos de vetores.

Todos os métodos apresentaram um crescimento exponencial do tempo de processamento com o aumento do tamanho do vetor (Figuras 4, 5 e 6), mas

observando as equações das linhas de tendência de cada um dos métodos, o crescimento é muito mais acentuado e rápido para o método Bubble Sort, seguido do método Insertion Sort e por último do método Merge Sort. Dessa forma, a variação de tempo de processamento levando em consideração do vetor 500 até o 50.000 segue a mesma ordem, com o Bubble Sort variando 24.022 ms, o Insertion Sort variando 2990 ms e o Merge Sort variando 67 ms (Tabela 1).

Esse resultado já era esperado, pois o Bubble Sort consiste em um método de comparação de pares consecutivos, ou seja, cada iteração consiste na comparação de todas posições consecutivas do vetor (HONORATO, 2013). Dessa forma, quanto maior for o vetor, maior será o tempo de execução de cada iteração, pois será preciso percorre-lo por inteiro fazendo as comparações. Além disso, o número de iterações desse método será fixo e sempre n-1, sendo n o tamanho do vetor. Consequentemente, o algoritmo continua funcionando até completar as iterações, mesmo que o vetor já esteja ordenado e por isso acaba gastando mais tempo e memória (PUGA & RISSETTI, 2009).

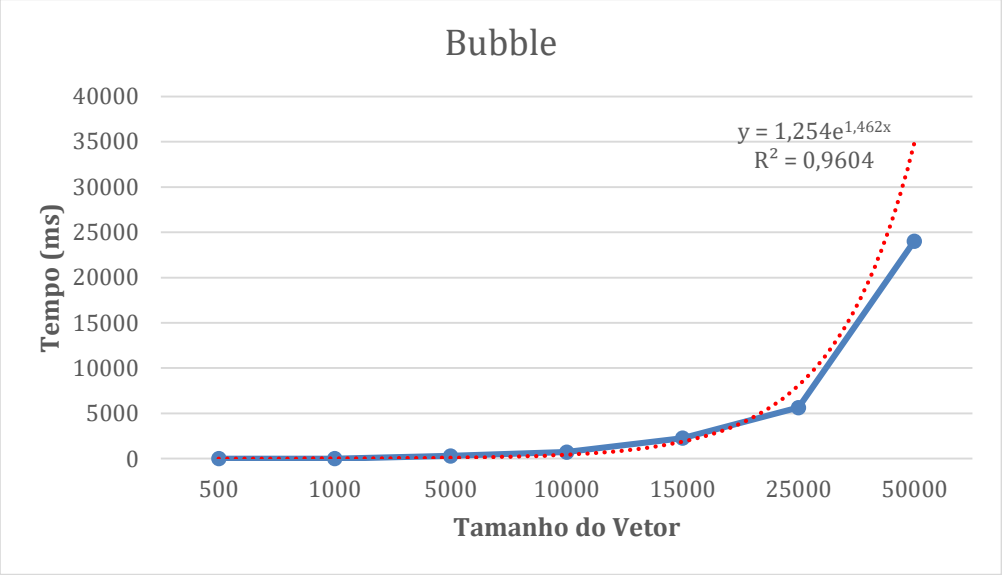


Figura 4 - Resultados dos testes a partir do vetor de 500 algarismos para o método de ordenação Bubble Sort, em milissegundos (ms) e sua linha de tendência exponencial.

Como esse método emprega dois laços de repetição aninhados (BORIN, 2020), a quantidade de comparações realizadas pelo método Bubble Sort será $n^2 - n$, sendo n o tamanho do vetor. Como dito anteriormente, qualquer que seja o vetor que está

sendo ordenado, o algoritmo se comportará da mesma maneira e realizará as comparações mesmo que desnecessárias (ASCENCIO & ARAÚJO, 2010).

O Insertion Sort, é um método que guarda um valor chave e compara-o com os valores à sua esquerda, caso esse valor seja maior que o valor chave, ocorre a cópia desse valor para a posição do valor chave. O valor chave só é posicionado de volta no vetor quando não encontrar valores maiores à sua esquerda (VIANA, 2017). Nesse caso, diferentemente do Bubble Sort, o Insertion Sort não percorre o vetor inteiro até a primeira posição fazendo as comparações consecutivas, pois a partir do momento que encontra um valor menor à esquerda do valor chave, a iteração é finalizada, assim elegendo um novo valor chave (próximo valor à direita) (ASCENCIO & ARAÚJO, 2010).

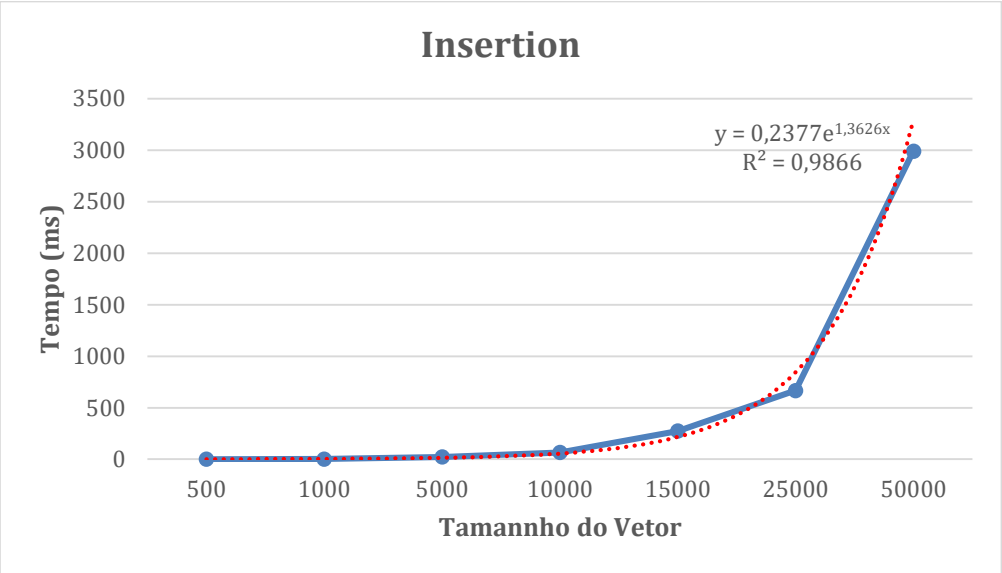


Figura 5 - Resultados dos testes a partir do vetor de 500 algarismos para o método de ordenação Insertion Sort, em milissegundos (ms) e sua linha de tendência exponencial.

Apesar de realizar um grande número de trocas entre os elementos, pois vai passando os números da esquerda para a direita um de cada vez conforme as comparações vão acontecendo, o Insertion Sort acaba fazendo menos trocas que o Bubble Sort. Isso ocorre, pois, as iterações do Bubble Sort ocorrem sempre da última posição do vetor até a segunda posição, enquanto que no Insertion Sort a iteração vai aumentando aos poucos, pois ocorre da posição do valor chave até a primeira posição (HONORATO, 2013).

Segundo ASCENCIO & ARAÚJO (2010), a complexidade e a eficiência desse método vão depender da situação do vetor inicial. Se o vetor inicial possui elementos na ordem inversa, o número de comparações e trocas serão muito maiores, e consequentemente o tempo de execução também será alto. Caso o vetor inicial já esteja ordenado, o número de comparações será menor e não será necessária nenhuma troca, assim resultando num baixo tempo de processamento.

O Merge Sort, segundo os resultados desse estudo, é o método de ordenação mais eficiente dentre os três escolhidos. Esse método se baseia na divisão do vetor em vetores com a metade do original de forma recursiva. Dessa forma, o vetor vai sendo dividido em partes menores até chegar em um tamanho indivisível e a partir daí a função recursiva começa a ser finalizada para cada um dos subvetores, ordenando-os em ordem crescente (ASCENCIO & ARAÚJO, 2010).

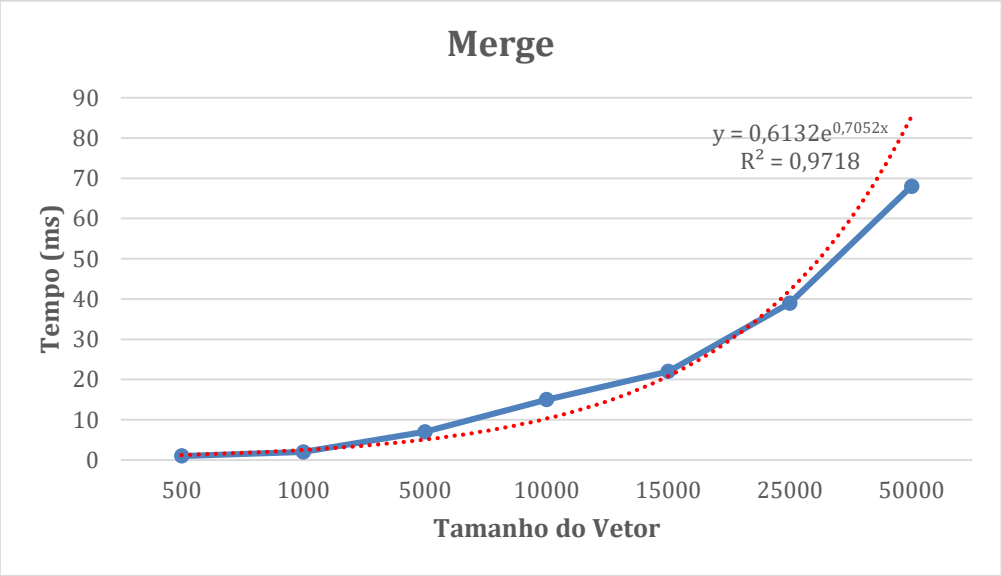


Figura 6 - Resultados dos testes a partir do vetor de 500 algarismos para o método de ordenação Merge Sort, em milissegundos (ms) e sua linha de tendência exponencial.

Como o vetor é separado em diversos subvetores de tamanho 1, para depois realizar a ordenação de cada uma dessas pequenas partes, o número de comparações acaba sendo menor. As comparações irão ocorrer apenas no momento de intercalação/união dos subvetores e serão feitas por partes. Como ambos os subvetores já estão ordenados, basta ir comparando o início de cada um deles e

intercalando seus valores de forma que a junção dos dois também esteja ordenada (BORIN, 2020).

Apesar de ser um método muito eficiente, o Merge Sort acaba sendo um algoritmo bem complexo já que necessita do uso da recursividade de funções. Como o algoritmo cria uma cópia do vetor para cada nível da recursividade, o gasto de memória pode ser grande.

6. CONSIDERAÇÕES FINAIS

O método Bubble Sort, é um algoritmo simples, mas acaba não sendo muito eficiente. Como o vetor é percorrido por i iterações e realiza todas as comparações independente da necessidade, o algoritmo se torna ineficiente para vetores muito grandes. Dentre os métodos escolhidos esse método se mostrou o mais ineficiente principalmente em vetores de maiores tamanhos.

O método Insertion Sort é também um algoritmo simples e acaba sendo mais eficiente do que o Bubble Sort, principalmente por não fazer tantas comparações e não precisar realizar todas as iterações dependendo do vetor que está sendo ordenado. Esse algoritmo se mostrou bem eficiente para vetores menores, mas a sua eficiência diminui drasticamente para vetores grandes como o de 50.000.

O método Merge Sort, diferente dos outros é um algoritmo mais complexo, pois se utiliza de funções recursivas. Apesar de gastar uma maior memória para armazenar os subvetores e suas junções, se mostrou o algoritmo mais rápido dos três, devido principalmente pelo menor número de comparações realizadas. Esse algoritmo se mostrou bem eficiente para vetores muito grandes.

Com isso, podemos concluir que o custo do algoritmo aumenta com o tamanho n do vetor. O tamanho n fornece uma medida da dificuldade para resolver o problema, assim, o tempo necessário aumenta quando o n cresce. O número de comparações para ordenar um vetor aumenta com o tamanho n. A escolha do algoritmo não é um problema quando n é pequeno. O problema é quando n cresce. Por isso, é importante analisar as funções de custo quando n é grande.

7. REFERÊNCIAS BIBLIOGRÁFICAS

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.

BORIN, V. P. **Estrutura de Dados**. 1. ed. Curitiba: Contentus, 2020.

BRAGA, H. **Algoritmos de Ordenação: Insertion Sort**. 2018. Disponível em <https://henriquebraga92.medium.com/algoritmos-de-ordena%C3%A7%C3%A3o-iii-insertion-sort-bfade66c6bf1>. Acesso em 10 de outubro de 2021.

BRUNET, J. A. **Ordenação por Comparação: Insertion Sort**. 2019. Disponível em: <https://joaoarthurbm.github.io/eda/posts/insertion-sort/>. Acesso em 18 de outubro de 2021.

DEVFURIA. **Introdução ao algoritmo de ordenação Insertion Sort**. 2012. Disponível em: <http://devfuria.com.br/logica-de-programacao/introducao-ao-algoritmo-de-ordenacao-insertion-sort/>. Acesso em: 16 de outubro de 2021.

FARIAS, R. **Estrutura de Dados e Algoritmos**. 2014. Disponível em: <https://www.cos.ufrj.br/~rfarias/cos121/aula_07.html>. Acesso em 19 de outubro de 2021.

FELIPE, H. **Merge Sort**. 2018. Disponível em: <https://www.blogcyberini.com/2018/07/merge-sort.html>. Acesso em 22 de outubro de 2021.

FEOFILOFF, P. **Heapsort**. 2018. Disponível em: < https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>. Acesso em: 09 de outubro de 2021.

GATTO, E. C. **Algoritmos de Ordenação: Bubble Sort**. 2017. Disponível em: < https://www.embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/>. Acesso em: 20 de outubro de 2021.

HONORATO, B. A. **Algoritmos de ordenação: análise e comparação**. 2013. Disponível em: <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>. Acesso em: 28 de setembro de 2021.

PUGA, S.; RISSETTI, G. **Lógica de Programação e Estrutura de Dados:** com aplicações em Java. 2. ed. São Paulo: Pearson Prentice, 2009.

VIANA, D. **Conheça os principais algoritmos de ordenação**. 2017. Disponível em: <https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>. Acesso em: 03 de outubro de 2021.

CÓDIGO FONTE

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include "main.h"

int main() {
    // Acentuação
    setlocale(LC_ALL, "Portuguese");

    int acabou = 0;

    abertura(1);

    do {
        // main.h {
        VETOR v;
        // }

        int escolha, escolhaCompararVal;

        // vetor.c {
        inicializaVetor(&escolha, &v);
        organizaVetor(&escolha, &v);
        comparaVetores(&escolhaCompararVal, &escolha, &v);
        // }

        printf("\nVocê deseja usar outro vetor?\n1) Sim\n2) Não\n\nEscolha: ");
        scanf(" %d", &acabou);
        acabou--;

        // vetor.c {
        liberaMem(&v);
        // }

    } while (!acabou);

    if (acabou == 1) {
        printf("\nTudo bem, até mais! :3\n\n");
    }
    else {
        printf("\nValor invalido!\n\n");
    }

    return 0;
}
```

main.h

```
// Vetor
void abertura();
struct vetor { int* vetor[3], tamanhoDoVetor; };
typedef struct vetor VETOR;
void qualTamanhoDoVetor(int* tamanhoDoVetor);
void geraAleatorio(int tamanhoDoVetor, int vetor[]);
void leDados(int tamanhoDoVetor, int vetor[]);
void printVetor(int vetor[], int tamanhoDoVetor);
void printMerge(int vetor[], int tamanhoDoVetor);
void executaBubbleSort(int vetor[], int tamanhoDoVetor);
void executaInsertionSort(int vetor[], int tamanhoDoVetor);
void executaMergeSort(int vetor[], int tamanhoDoVetor);
void inicializaVetor(int* escolha, VETOR* v);
void organizaVetor(int* escolha, VETOR* v);
void comparaVetores(int* escolhaCompararVal, int* escolha, VETOR* v);
void liberaMem(VETOR* v);

// Cronômetro
void printTempo(int escolha);
void printTempos();
void contagemCronometro(int escolha, unsigned long inicio, unsigned long
fim);

// BubbleSort
void bubbleSort(int* vetor, int tamanhoDoVetor);
void troca(int* a, int* b);

// Insertion Sort
void insertionSort(int vetor[], int tamanhoDoVetor);

// Merge Sort
void merge(int vetor[], int comeco, int meio, int fim);
void mergeSort(int vetor[], int comeco, int fim);
```

vetor.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "main.h"

// Recebe qual será o tamanho do vetor
void qualTamanhoDoVetor(int* tamanhoDoVetor) {
    printf("\nInsira a quantidade de números dentro do vetor:\n");
    scanf("%d", tamanhoDoVetor);
}

void geraAleatorio(int tamanhoDoVetor, int vetor[]) {
    int limiteCasasAleatorios = 100; // 100 = de 0 a 99.
    srand(time(0)); // Função para gerar aleatório com base no tempo para
não repetir os valores
    for (int i = 0; i < tamanhoDoVetor; i++) {
        vetor[i] = rand() % limiteCasasAleatorios;
    }
}

// Caso for selecionada a opção de escolher os valores do vetor,
// a função irá receber os valores que o usuário fornecerá
void leDados(int tamanhoDoVetor, int vetor[]) {
    printf("\nInsira os valores dentro do vetor:\n");
    for (int i = 0; i < tamanhoDoVetor; i++) {
        scanf("%d", &vetor[i]);
    }
}

// Print padrão de vetores
void printVetor(int vetor[], int tamanhoDoVetor) {
    for (int i = 0; i < tamanhoDoVetor; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}

// O print do merge sort precisa ser diferente, pois a função sempre gera
um número 0 no final do vetor
// e este valor é ordenado junto com os outros, ou seja, toda vez que
printamos um merge usando o print
// de cima, é mostrado um valor 0 no início e o ultimo valor não é exibido.
// Para corrigir, iniciamos a variável do loop em 1 para pular o primeiro
valor e dizemos que o loop só vai
// finalizar quando i <= que o tamanhoDoVetor
void printMerge(int vetor[], int tamanhoDoVetor) {
    for (int i = 1; i <= tamanhoDoVetor; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}
```

```c
void executaBubbleSort(int vetor[], int tamanhoDoVetor) {
    clock_t inicioCronometro = clock();
    bubbleSort(vetor, tamanhoDoVetor - 1); // (função em BubbleSort.c)
    clock_t fimCronometro = clock();
    contagemCronometro(0, inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}

void executaInsertionSort(int vetor[], int tamanhoDoVetor) {
    clock_t inicioCronometro = clock();
    insertionSort(vetor, tamanhoDoVetor); // (Função em InsertionSort.c)
    clock_t fimCronometro = clock();
    contagemCronometro(1, inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}

void executaMergeSort(int vetor[], int tamanhoDoVetor) {
    clock_t inicioCronometro = clock();
    mergeSort(vetor, 0, tamanhoDoVetor); // (Função em MergeSort.c)
    clock_t fimCronometro = clock();
    contagemCronometro(2, inicioCronometro, fimCronometro); // (Função em
cronometro.c)
}


void inicializaVetor(int* escolha, VETOR* v) {

    qualTamanhoDoVetor(&v->tamanhoDoVetor);

    // Alocação dinâmica de memória
    for (int i = 0; i < sizeof(v->vetor) / sizeof(v->vetor[0]); i++) {
        v->vetor[i] = (int*)malloc(sizeof(int) * v->tamanhoDoVetor + 1);
    }

    printf("\nVocê deseja usar números aleatórios ou escolher os valores a
serem ordenados?\n1) Aleatórios\n2) Escolher os valores\n\nDigite: ");
    scanf("%d", escolha);

    switch (*escolha) {
    case 1:
        geraAleatorio(v->tamanhoDoVetor, v->vetor[0]);
        break;
    case 2:
        leDados(v->tamanhoDoVetor, v->vetor[0]);
        break;
    default:
        printf("Erro! Valor inválido!\n");
        exit(1);
    }

    // Aqui, irá haver uma cópia do vetor desordenado para os outros 2
vetores. Por exemplo:
```

```
    // o vetor foi definido para ter o tamanho de 5 valores e esses valores
foram gerados aleatóriamente
    // (Lembrando que começa em [0] e o valor do último [5] é um "\0").
    // Então, na memória, o primeiro vetor ficará:
    // vetor[0][5] = {21, 23, 43, 2, 5}
    // o vetor[0][5] será copiado para os vetores vetor[1][5] e vetor[2][5]


    // A sintaxe do vetor é mais ou menos assim:
    // int vetor[0][tamanhoDoVetor];

    // - O vetor[0] sempre será ordenado pelo BubbleSort
    // - O vetor[1] sempre será ordenado pelo InsertionSort
    // - O vetor[2] sempre será ordenado pelo MergeSort
    for (int i = 0; i <= v->tamanhoDoVetor; i++) {
        v->vetor[2][i] = v->vetor[1][i] = v->vetor[0][i];
    }
}

void organizaVetor(int* escolha, VETOR* v) {
    printf("\nQual algorítmo de ordenação você deseja usar?\n1) Bubble
Sort\n2) Insertion Sort\n3) Merge Sort\n\nDigite: ");
    scanf("%d", escolha);

    if (*escolha == 1) {
        printf("\n\n-- BUBBLE SORT --\n");
    }
    else if (*escolha == 2) {
        printf("\n\n-- INSERTION SORT --\n");
    }
    else if (*escolha == 3) {
        printf("\n\n-- MERGE SORT --\n");
    }
    else {
        printf("\nErro! Opção Inválida!\n");
        exit(1);
    };

    printf("\n\033[1;31mVetor antes da ordenação:\n");
    printVetor(v->vetor[0], v->tamanhoDoVetor);

    printf("\n\033[0;32mVetor depois da ordenação:\n");

    switch (*escolha) {
    case 1:
        executaBubbleSort(v->vetor[0], v->tamanhoDoVetor);
        printVetor(v->vetor[0], v->tamanhoDoVetor);
        printTempo(*escolha); // (Função em cronometro.c)
        break;
    case 2:
        executaInsertionSort(v->vetor[1], v->tamanhoDoVetor);
        printVetor(v->vetor[1], v->tamanhoDoVetor);
        printTempo(*escolha); // (Função em cronometro.c)
        break;
```

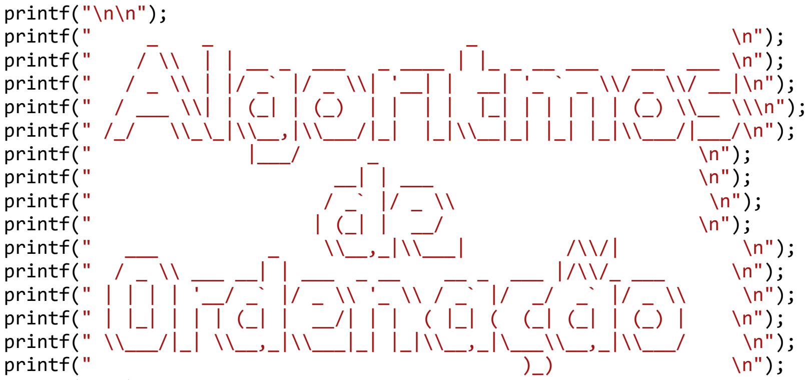
```
    case 3:
        executaMergeSort(v->vetor[2], v->tamanhoDoVetor);
        printMerge(v->vetor[2], v->tamanhoDoVetor);
        printTempo(*escolha); // (Função em cronometro.c)
        break;
    }
}

void comparaVetores(int* escolhaCompararVal, int* escolha, VETOR* v) {
    printf("Você deseja comparar os tempos de cada algoritmo para o mesmo
vetor?\n1) Sim\n2) Não\n\nEscolha: ");
    scanf("%d", escolhaCompararVal);

    if (*escolhaCompararVal == 1) {
        switch (*escolha) {
            // Se quiser comparar os tempos, será executado a ordenação dos
outros vetores
            // sem os printar, apenas será salvo os tempos no vetor
algoritmos[] do arquivo cronometro.c
            // e serão exibidos os seus tempos
        case 1:
            // Caso tenha escolhido o bubble sort
            executaInsertionSort(v->vetor[1], v->tamanhoDoVetor);
            executaMergeSort(v->vetor[2], v->tamanhoDoVetor);
            printTempos(); // (Função em cronometro.c)
            break;
        case 2:
            // Caso tenha escolhido o insertion sort
            executaBubbleSort(v->vetor[0], v->tamanhoDoVetor);
            executaMergeSort(v->vetor[2], v->tamanhoDoVetor);
            printTempos(); // (Função em cronometro.c)
            break;
        case 3:
            // Caso tenha escolhido o merge sort
            executaBubbleSort(v->vetor[0], v->tamanhoDoVetor);
            executaInsertionSort(v->vetor[1], v->tamanhoDoVetor);
            printTempos(); // (Função em cronometro.c)
            break;
        default:
            printf("\nErro! Opção Inválida!\n");
            exit(1);
        }
    }
}

// Libera a memória alocada dinâmicamente
void liberaMem(VETOR* v) {
    for (int i = 0; i < sizeof(v->vetor) / sizeof(v->vetor[0]); i++) {
        free(v->vetor[i]);
    }
}
```

cronometro.c

```c
#include <stdio.h>
#include <time.h>
#include "main.h"


// - O algoritmos[0] sempre guardará o tempo do Bubble Sort
// - O algoritmos[1] sempre guardará o tempo do Insertion Sort
// - O algoritmos[2] sempre guardará o tempo do Merge Sort
float algoritmos[3];

void printTempo(int escolha) {
    printf("\n\033[0;37mTempo de processamento da ordenação: %.5lfms\n\n",
algoritmos[escolha - 1]);
}

void printTempos() {
    printf("\nTempo de processamento da ordenação Bubble Sort: %.5lfms\n",
algoritmos[0]);
    printf("Tempo de processamento da ordenação Insertion Sort: %.5lfms\n",
algoritmos[1]);
    printf("Tempo de processamento da ordenação Merge Sort: %.5lfms\n",
algoritmos[2]);
}

// Calcula o tempo gasto pela função de ordenação
// com base nos tempos de início e fim
void contagemCronometro(int escolha, unsigned long inicio, unsigned long
fim) {
    algoritmos[escolha] = (double)(fim - inicio) / (CLOCKS_PER_SEC / 1000);
}
```

abertura.c

```c
/*
                            Arte gerada em:
https://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%
20
*/

#include <stdio.h>

void abertura(int escolha) {
    if (escolha == 1) {
      printf("\n\n");
```



```c
      printf("\n");
    }
    else if (escolha == 2) {
      printf("\n\n");
```

```c
      printf("\n");
    }
    else {
        printf("\n-- Algoritmos de ordenação --\n");
    }
}
```

BubbleSort.c

```c
void troca(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int* vetor, int tamanhoDoVetor) {

    if (tamanhoDoVetor < 1) {
        return;
    }

    for (int i = 0; i < tamanhoDoVetor; i++) {
        if (vetor[i] > vetor[i + 1]) {
            troca(&vetor[i], &vetor[i + 1]);
        }
    }
    bubbleSort(vetor, tamanhoDoVetor - 1);
}
```

InsertionSort.c

```c
void insertionSort(int vetor[], int tamanhoDoVetor) {
    int i, j, valorDovetor;
    for (i = 1; i < tamanhoDoVetor; i++) {
        valorDovetor = vetor[i];
        j = i - 1;

        while (j >= 0 && vetor[j] > valorDovetor) { // Enquanto j >= 0 e o
numero j do vetor > valorDovetor, faça:
            vetor[j + 1] = vetor[j];
            j = j - 1;
        }
        vetor[j + 1] = valorDovetor;
    }
}
```

MergeSort.c

```c
#include <stdio.h>
#include <stdlib.h>

void merge(int vetor[], int comeco, int meio, int fim) {
    int com1 = comeco, com2 = meio + 1, comAux = 0, tam = fim - comeco + 1;
    int* vetAux;
    vetAux = (int*)malloc(tam * sizeof(int));

    while (com1 <= meio && com2 <= fim) {
        if (vetor[com1] < vetor[com2]) {
            vetAux[comAux] = vetor[com1];
            com1++;
        }
        else {
            vetAux[comAux] = vetor[com2];
            com2++;
        }
        comAux++;
    }

    while (com1 <= meio) {  //Caso ainda haja elementos na primeira metade
        vetAux[comAux] = vetor[com1];
        comAux++;
        com1++;
    }

    while (com2 <= fim) {   //Caso ainda haja elementos na segunda metade
        vetAux[comAux] = vetor[com2];
        comAux++;
        com2++;
    }

    for (comAux = comeco; comAux <= fim; comAux++) {    //Move os elementos
de volta para o vetor original
        vetor[comAux] = vetAux[comAux - comeco];
    }

    free(vetAux);
}

void mergeSort(int vetor[], int comeco, int fim) {
    if (comeco < fim) {
        int meio = (fim + comeco) / 2;

        mergeSort(vetor, comeco, meio);
        mergeSort(vetor, meio + 1, fim);
        merge(vetor, comeco, meio, fim);
    }
}
```