

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE MATO GROSSO DO SUL
CÂMPUS AQUIDAUANA
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA
INTERNET**

**INÊS EUSTÁQUIO CAETANO RIBEIRO
IURY FEITOSA DA ROCHA
MATHEUS DANIEL CRISTAL COMPAROTTO GOMES**

**MENSURAÇÃO DE DESEMPENHO DE SEIS MÉTODOS DE
ORDENAÇÃO UTILIZANDO A LINGUAGEM DE PROGRAMAÇÃO
TYPESCRIPT**

Estudo dos tempos de execução dos seguintes métodos de ordenação: método da bolha; método de ordenação por seleção; método de ordenação por inserção; método de ordenação por intercalação; método de ordenação por separação e método de ordenação por monte

AQUIDAUANA - MS

2022

INÊS EUSTÁQUIO CAETANO RIBEIRO
IURY FEITOSA DA ROCHA
MATHEUS DANIEL CRISTAL COMPAROTTO GOMES

Mensuração de desempenho de seis métodos de ordenação utilizando a linguagem de programação TypeScript: Estudo dos tempos de execução dos seguintes métodos de ordenação: método da bolha; método de ordenação por seleção; método de ordenação por inserção; método de ordenação por intercalação; método de ordenação por separação e método de ordenação por monte.

Relatório apresentado no Curso Superior de Tecnologia em Sistemas para Internet do Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso do Sul Câmpus Aquidauana como requisito para obtenção da nota parcial das atividades da unidade curricular Estruturas de Dados.

Orientador: Leandro Magalhães de Oliveira

AQUIDAUANA - MS

2022

RESUMO

O seguinte trabalho tem como objetivos analisar os tempos de execução dos métodos de ordenação método da bolha (em inglês: *bubble sort*), método de ordenação por seleção (em inglês: *selection sort*), método de ordenação por inserção (em inglês: *insertion sort*), método de ordenação por intercalação (em inglês: *merge sort*), método de ordenação por separação (em inglês: *quick sort*) e método de ordenação por monte (em inglês: *heap sort*) e identificar para quais entradas determinados métodos de ordenação são eficientes.

Palavras-chave: Ordenação; Eficiência; Vetor; Tempo; Milissegundos.

LISTA DE ILUSTRAÇÕES

FIGURAS

Figura 1 - Primeiras aparições das linguagens de programação.....	8
Figura 2 - Função de ler uma entrada e convertê-la em um vetor.....	13
Figura 3 - Exemplo de uso dos métodos “console.time” e “console.timeEnd”.....	14
Figura 4 - Código-fonte efetivo do arquivo principal.....	15
Figura 5 - Método da bolha.....	17
Figura 6 - Método de ordenação por seleção.....	18
Figura 7 - Método de ordenação por inserção.....	19
Figura 8 - Método de ordenação por intercalação.....	20
Figura 9 - Função que seleciona o elemento <i>pivot</i> do método de ordenação por separação.....	21
Figura 10 - Método de ordenação por separação.....	21
Figura 11 - Classe para instanciar um <i>heap</i> máximo.....	22
Figura 12 - Método de ordenação por monte.....	23
Figura 13 - Novo código-fonte do arquivo principal.....	24
Figura 14 - Cinco primeiras linhas do arquivo de texto contendo os resultados.....	25
Figura 15 - Função de criar planilhas.....	26

GRÁFICOS

Gráfico 1 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de um milhão de números desordenados.....	31
--	----

Gráfico 2 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente.....	31
Gráfico 3 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem decrescente.....	32
Gráfico 4 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente, com exceção do primeiro.....	32

LISTA DE TABELAS

Tabela 1 - Raiz do projeto.....	11
Tabela 2 - Subdiretórios do diretório “src/”	12
Tabela 3 - Dezesete configurações da máquina utilizada para a mensuração de desempenho.....	16
Tabela 4 - Tempo de execução dos métodos de ordenação para um vetor de um milhão de números desordenados.....	27
Tabela 5 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente.....	28
Tabela 6 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem decrescente.....	29
Tabela 7 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente, com exceção do primeiro.....	30

SUMÁRIO

RESUMO.....	3
LISTA DE ILUSTRAÇÕES.....	4
FIGURAS.....	4
GRÁFICOS.....	4
LISTA DE TABELAS.....	6
RECURSIVIDADE.....	7
INTRODUÇÃO.....	8
OBJETIVOS.....	10
OBJETIVOS GERAIS.....	10
OBJETIVOS ESPECÍFICOS.....	10
METODOLOGIA.....	11
CONFIGURAÇÕES	DA
MÁQUINA.....	16
MÉTODO DA BOLHA.....	17
MÉTODO DE ORDENAÇÃO POR SELEÇÃO.....	18
MÉTODO DE ORDENAÇÃO POR INSERÇÃO.....	19
MÉTODO DE ORDENAÇÃO POR INTERCALAÇÃO.....	20
MÉTODO DE ORDENAÇÃO POR SEPARAÇÃO.....	21
MÉTODO DE ORDENAÇÃO POR MONTE.....	22
RESULTADOS E DISCUSSÃO.....	24
CONSIDERAÇÕES FINAIS.....	33
REFERÊNCIAS.....	34

1. INTRODUÇÃO

Por vezes, as pessoas têm de organizar informações, sejam números ou palavras, em uma ordem específica para um determinado fim. Por exemplo: um recepcionista de um consultório médico elenca os pacientes por ordem de chegada a fim de valorizar a pontualidade dos pacientes que chegaram mais cedo; um professor faz a chamada com os nomes dos seus alunos em ordem alfabética e identifica os alunos com dificuldades com as notas da turma em ordem crescente; um investidor do mercado financeiro estuda uma criptomoeda analisando seu valor de mercado por hora; um cliente de um *E-commerce* procura entre várias ofertas de um mesmo produto a que tenha o menor preço *etc.* Nesse contexto, a invenção do computador simplificou o trabalho de organizar informações sequenciadas.

Figura 1 - Primeiras aparições das linguagens de programação.

Programming Languages First Appearances		
Name & Year		Designed / Written By
C	1972	Dennis Ritchie
C++	1985	Bjarne Stroustrup
C#	2000	Microsoft Corporation
GoLang	2009	Rob Pike & Ken Thompson
Java	1995	James Gosling
Javascript	1995	Brendan Eich
Julia	2012	Jeff, Alan, Stefan, Viral
Kotlin	2011	Jet Brains
Matlab	1970	Cleve Moler
PHP	1995	Rasmus Lerdorf
Python	1990	Guido van Rossum
R	1993	Ross Ihaka, Robert Gentleman
Ruby	1995	Yukihiro Matsumoto
Swift	2014	Chris Lattner & Apple
.Net	2001	Andres Hejlesberg

Fonte: Página no Twitter¹.

Como exemplificado pela figura acima, a primeira linguagem de programação, C, foi oficialmente lançada na década de setenta. Essa linguagem de programação deu origem a uma

¹ Disponível em: <https://twitter.com/iphonegalaxymd/status/1315092513821204482/photo/1>. Acesso em: 1 out. 2022.

estrutura de dados — que também é nativa em todas as linguagens de programação, diga-se de passagem — chamada vetor (em inglês: *array*), a qual consiste em uma lista de elementos, sejam eles números ou caracteres.

Com o advento dos vetores, alguns algoritmos vêm sendo escritos a fim de ordenar um vetor em ordem crescente ou decrescente e por isso são conhecidos como métodos de ordenação. A estratégia de cada método de ordenação visa racionar o tempo de execução, o consumo de memória, a quantidade de processos comunicando-se entre si e a quantidade de portas lógicas acessadas independentemente do tamanho do vetor ou da disposição dos seus elementos.

Dado que as linguagens de programação têm tempos de execução diferentes para o mesmo algoritmo, foi selecionada a linguagem de programação TypeScript devido a sua alta aceitação e replicabilidade para mensurar o desempenho de seis métodos de ordenação selecionados pelo orientador, Prof. Me. Leandro Magalhães de Oliveira.

Em algumas linguagens de programação, em especial as linguagens de programação mais recentes, um vetor pode ser homogêneo; ou seja, pode comportar elementos de diferentes tipos primitivos. Isso posto, é digno de nota que o seguinte trabalho tratará apenas de vetores heterogêneos e numéricos.

2. OBJETIVOS

2.1. OBJETIVOS GERAIS

Mensurar o desempenho dos seguintes métodos de ordenação:

- Método da bolha (em inglês: *bubble sort*);
- Método de ordenação por seleção (em inglês: *selection sort*);
- Método de ordenação por inserção (em inglês: *insertion sort*);
- Método de ordenação por intercalação (em inglês: *merge sort*);
- Método de ordenação por separação (em inglês: *quick sort*);
- Método de ordenação por monte (em inglês: *heap sort*).

2.2. OBJETIVOS ESPECÍFICOS

Concluir, com base nas seguintes entradas fornecidas pelo orientador, Prof. Me. Leandro Magalhães de Oliveira, para qual ocasião cada um dos métodos de ordenação supracitados é eficiente:

- Um milhão de números desordenados;
- Quinhentos mil números ordenados em ordem crescente;
- Quinhentos mil números ordenados em ordem decrescente;
- Quinhentos mil números ordenados em ordem crescente, com exceção do primeiro.

3. METODOLOGIA

A priori, foi criado um repositório público na plataforma de hospedagem de código-fonte GitHub, o qual está disponível no endereço eletrônico <https://github.com/mdccg/sorting-algorithms-benchmark>. O repositório faz uso de apenas três dependências de desenvolvimento que atuam em conjunto para compilar códigos-fonte escritos em TypeScript. Além disso, todos os códigos-fonte foram escritos em língua inglesa visando boas práticas de desenvolvimento de *software*. Eis a estrutura do projeto:

Tabela 1 - Raiz do projeto.

Diretório ou arquivo	Papel
docs/	Diretório que conterá o presente relatório para que os visitantes do repositório o consultem.
node_modules/	Diretório contendo todas as dependências utilizadas no projeto.
src/	Diretório contendo todos os códigos-fonte <i>etc.</i> organizados em subdiretórios.
.gitignore	Arquivo de versionamento de <i>software</i> para não subir diretórios e arquivos desnecessários para o repositório on-line, como o diretório node_modules/.
index.ts	Arquivo principal, o qual importa funções exportadas de códigos-fonte contidos no diretório src/, em que é cronometrada a ordenação dos vetores por cada um dos seis métodos de ordenação.
package.json	Arquivo contendo os apelidos para rotinas (nele chamados de “ <i>scripts</i> ”) e dependências do projeto, além de outras informações.
README.md	Arquivo contendo um resumo não formal do trabalho para ser exibido aos visitantes do repositório.
tsconfig.json	Arquivo gerado automaticamente através do comando “npx tsc --init” contendo configurações para compilar o projeto.
yarn.lock	Arquivo contendo as versões atuais das dependências utilizadas no projeto, cujo papel é paralelo ao papel do arquivo package.json.

Tabela 2 - Subdiretórios do diretório “src/”.

Diretório	Papel
data/	Diretório contendo cinco arquivos, quatro arquivos respectivos às quatro entradas mencionadas na subseção de objetivos específicos e um arquivo que se trata da saída do arquivo principal copiada manualmente.
functions/	Diretório contendo oito arquivos, dos quais um arquivo se trata da função para ler uma entrada — utilizando as funções “join” e “readFileSync” dos pacotes nativos “path” e “fs”, respectivamente — e convertê-la em um vetor numérico; seis arquivos se tratam dos seis métodos de ordenação mencionados na subseção de objetivos gerais e um arquivo se trata de uma função criada após a mensuração de desempenho para facilitar a organização dos dados. Cada arquivo referente a um método de ordenação pode conter mais de uma função e somente a função respectiva ao método de ordenação em questão é exportada por padrão.
sheets/	Diretório contendo quatro planilhas, cada uma respectiva a uma entrada e contendo trinta tempos de execução e cinco médias aritméticas, o qual foi criado após a mensuração de desempenho dos seis métodos de ordenação para facilitar a organização dos dados.

No arquivo principal, foram importadas a função de ler uma entrada e convertê-la em um vetor, exemplificada abaixo, e as funções dos métodos de ordenação.

Figura 2 - Função de ler uma entrada e convertê-la em um vetor.

```
1 import { join } from 'path';
2 import { readFileSync } from 'fs';
3
4 const readArray = (fileName: string): number[] => {
5   return readFileSync(
6     join(__dirname, '..', 'data', fileName),
7     'utf-8'
8   )
9     .split('\n')
10    .map((value: string) => Number(value));
11 }
12
13 export default readArray;
```

Fonte: acervo pessoal.

Após as importações, as seguintes instruções foram escritas:

- Foi declarado um vetor contendo as seis funções respectivas aos seis métodos de ordenação;
- Foi declarado o primeiro laço de repetição que percorre o vetor supracitado;
- Foi declarado o segundo laço de repetição, inscrito no primeiro laço de repetição, no qual sua variável de controle vai de um a quatro (inclusive) e foi declarada uma constante chamada “*file name*” (em *camel case*² e do inglês: nome do arquivo) que recebe o prefixo comum aos nomes dos arquivos das entradas, a variável de controle e o formato do arquivo concatenados. O resultado é o nome do arquivo da entrada a ser lida na iteração em questão;
- Foi declarado o terceiro laço de repetição, inscrito no segundo laço de repetição, no qual sua variável de controle vai de um a cinco³ (inclusive) e foram declarados:
 - Uma constante chamada “*label*” (do inglês: etiqueta) que recebe o nome do método de ordenação, o número da entrada e o número da repetição concatenados;
 - O vetor numérico convertido através da função “*read array*” (em *camel case* e do inglês: ler vetor) a ser ordenado na iteração em questão.


² Expressão constituinte do jargão de desenvolvedores de *software*, a qual significa “começar com a primeira letra minúscula e a primeira letra de cada nova palavra subsequente maiúscula.” (ALICE, 2022).

³ A razão pela qual o terceiro laço de repetição itera cinco vezes é o quarto requisito deste trabalho: “[...] Cada algoritmo deve ser executado ao menos 5 vezes para cada entrada e o aluno/grupo deverá considerar a média de tempo de execução do algoritmo para fins estatísticos.” (OLIVEIRA, 2022).

No terceiro laço de repetição, após as declarações supracitadas, foram utilizados dois métodos do objeto “console”: “time” (do inglês: tempo) e “time end” (em *camel case* e do inglês: fim do tempo).

O primeiro método “inicia um cronômetro [...] para monitorar quanto tempo uma operação leva [...] para cada cronômetro um nome único, e deve ter no máximo 10.000 deles sendo executados [...]” (MDN Web Docs, 2022). Já o segundo método “com o mesmo nome [...] mostrará o tempo, em milissegundos, que se passou desde que o cronômetro iniciou.” (MDN Web Docs, 2022). A figura abaixo exemplifica o uso dos dois métodos supracitados.

Figura 3 - Exemplo de uso dos métodos “console.time” e “console.timeEnd”.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains three lines of JavaScript code:

```
1 console.time('Identificador');  
2 console.log('Olá, mundo!');  
3 console.timeEnd('Identificador');
```

Fonte: acervo pessoal.

A saída do algoritmo acima é o texto “Olá, mundo!” seguido do texto “Identificador: 5.29ms”. Quando um algoritmo leva mais de cinquenta e nove segundos ou mais de cinquenta e nove minutos, a saída também especifica o formato de exibição de data e hora, como “(m:ss.mmm)” ou “(h:mm:ss.mmm)”, em que a letra “h” representa as horas, a penúltima ocorrência da letra “m” representa os minutos, a letra “s” representa o segundos e a última ocorrência da letra “m” representa os milissegundos em ambos os formatos.

Vale destacar que apenas a linha de comando que ordena o vetor foi cronometrada. O custo computacional das importações, declarações, laços de repetição e conversão do arquivo da entrada em vetor não faz parte dos resultados obtidos.

Eis o código-fonte do arquivo principal:

Figura 4 - Código-fonte efetivo do arquivo principal.

```
1  const sortingMethods = [  
2    bubbleSort,  
3    selectionSort,  
4    insertionSort,  
5    mergeSort,  
6    quickSort,  
7    heapSort  
8  ];  
9  
10 for (const sortingMethod of sortingMethods) {  
11   for (let i = 1; i ≤ 4; ++i) {  
12     const fileName = `entrada${i}.txt`;  
13  
14     for (let j = 1; j ≤ 5; ++j) {  
15       const label = `${sortingMethod.name}, entry No. ${i}, loop No. ${j}`;  
16       const array = readArray(fileName);  
17  
18       console.time(label);  
19       sortingMethod(array);  
20       console.timeEnd(label);  
21     }  
22   }  
23 }
```

Fonte: acervo pessoal.

3.1. CONFIGURAÇÕES DA MÁQUINA

Para a mensuração de desempenho, apenas o editor de código-fonte Visual Studio Code foi mantido aberto e as opções de economia de energia “Apagar a tela” e “Suspensão automática” foram desativadas. O único incidente ocorrido foi bateria fraca durante a quarta ordenação da primeira entrada pelo método de ordenação por monte, o qual foi resolvido imediatamente conectando o carregador. O mesmo procedimento foi seguido para todos os métodos de ordenação.

Eis algumas configurações tidas vagamente como relevantes da máquina e obtidas através do comando “lscpu”:

Tabela 3 - Dezesete configurações da máquina utilizada para a mensuração de desempenho.

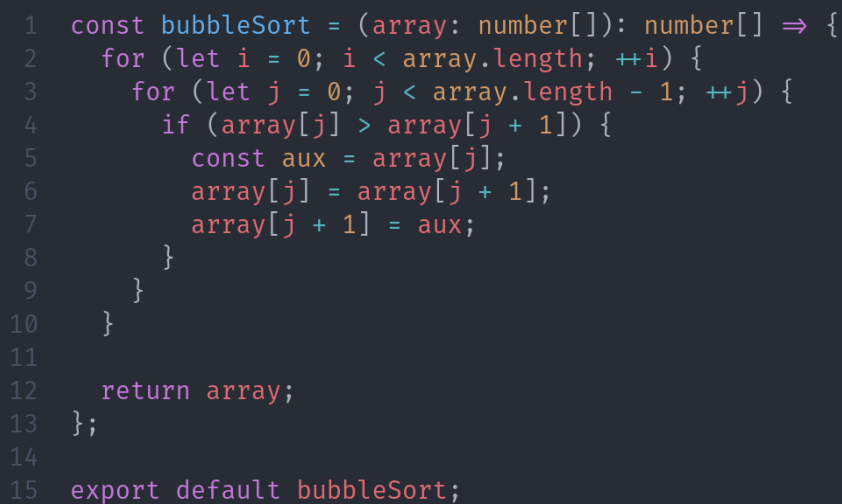
Etiqueta	Configuração
Arquitetura	x86_64
Modo(s) operacional da CPU	32-bit, 64-bit
<i>Address sizes</i>	39 bits <i>physical</i> , 48 bits <i>virtual</i>
Ordem dos bytes	Little Endian
CPU(s)	4
Lista de CPU(s) on-line	0-3
ID de fornecedor	GenuineIntel
Nome do modelo	Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz
Família da CPU	6
Modelo	78
Thread(s) per núcleo	2
Núcleo(s) por soquete	2
Soquete(s)	1
Step	3
CPU MHz máx.	2000,0000
CPU MHz mín.	400,0000
BogoMIPS	3999.93

3.2. MÉTODO DA BOLHA

O método da bolha (em inglês: *bubble sort*) consiste em comparar dois elementos consecutivos de um vetor e trocá-los de posição caso o elemento anterior seja maior que o elemento posterior, de modo a fazer flutuar os maiores elementos para o final do vetor a cada iteração do segundo laço de repetição.

Eis o código-fonte do método da bolha implementado:

Figura 5 - Método da bolha.



```
1  const bubbleSort = (array: number[]): number[] => {
2    for (let i = 0; i < array.length; ++i) {
3      for (let j = 0; j < array.length - 1; ++j) {
4        if (array[j] > array[j + 1]) {
5          const aux = array[j];
6          array[j] = array[j + 1];
7          array[j + 1] = aux;
8        }
9      }
10   }
11
12   return array;
13 };
14
15 export default bubbleSort;
```

Fonte: acervo pessoal.

3.3. MÉTODO DE ORDENAÇÃO POR SELEÇÃO

O método de ordenação por seleção (em inglês: *selection sort*) consiste em dividir o vetor em dois vetores menores, um vetor menor ordenado e um vetor menor não ordenado. O vetor menor ordenado está localizado no início do vetor e todos os elementos à direita do último elemento ordenado são considerados não ordenados^[3].

Inicialmente, o vetor menor ordenado está vazio, enquanto o restante do vetor não está ordenado. O método de ordenação por seleção percorre o vetor menor não ordenado e encontra o menor ou o maior elemento. O elemento é então trocado com o elemento mais à esquerda do vetor menor não ordenado. Em seguida, o vetor menor ordenado é expandido para incluir esse elemento.

Eis o código-fonte do método de ordenação por seleção implementado:

Figura 6 - Método de ordenação por seleção.



```
1  const selectionSort = (array: number[]): number[] => {
2    for (let i = 0; i < array.length; ++i) {
3      let min = i;
4
5      for (let j = i + 1; j < array.length; ++j) {
6        if (array[j] < array[min]) {
7          min = j;
8        }
9      }
10
11     if (min !== i) {
12       const aux = array[i];
13       array[i] = array[min];
14       array[min] = aux;
15     }
16   }
17
18   return array;
19 };
20
21 export default selectionSort;
22
```

Fonte: acervo pessoal.

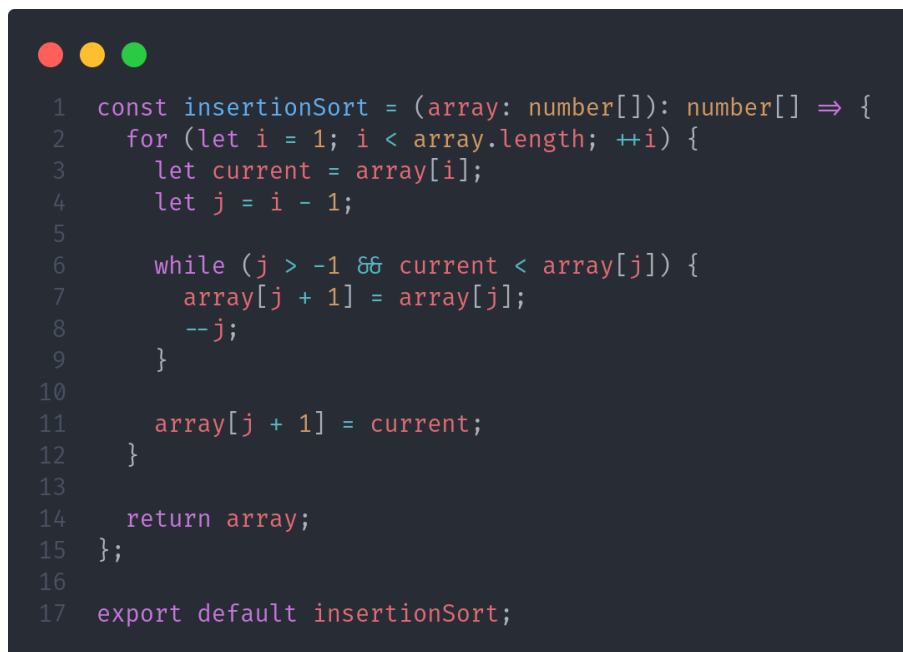
3.4. MÉTODO DE ORDENAÇÃO POR INSERÇÃO

A ideia por trás do método de ordenação por inserção (em inglês: *insertion sort*) é frequentemente comparada à maneira como as pessoas classificam uma mão de cartas enquanto jogam Mexe-mexe. Nesse jogo de cartas, o *dealer* distribui cartas para cada jogador. Em seguida, os jogadores pegam as cartas que lhes são dadas uma a uma, classificando-as em sua mão em ordem crescente, inserindo cada carta em seu lugar^[4].

Durante todo esse processo, os jogadores seguram uma lista de cartas ordenadas em suas mãos, enquanto a lista não ordenada da qual tiram novas cartas está à sua frente. Uma propriedade muito útil do método de ordenação por inserção é o fato de que ele não precisa conhecer o vetor inteiro antecipadamente para que ele seja ordenado, ele apenas insere os elementos dados um por um. Isso é útil quando se deseja adicionar mais elementos em um vetor já ordenado, pois o método de ordenação por inserção adicionará os novos elementos em seus devidos lugares sem recorrer ao vetor inteiro.

Eis o código-fonte do método de ordenação por inserção implementado:

Figura 7 - Método de ordenação por inserção.



```
1  const insertionSort = (array: number[]): number[] => {
2    for (let i = 1; i < array.length; ++i) {
3      let current = array[i];
4      let j = i - 1;
5
6      while (j > -1 && current < array[j]) {
7        array[j + 1] = array[j];
8        --j;
9      }
10
11      array[j + 1] = current;
12    }
13
14    return array;
15  };
16
17  export default insertionSort;
```

Fonte: acervo pessoal.

3.5. MÉTODO DE ORDENAÇÃO POR INTERCALAÇÃO

O método de ordenação por intercalação (em inglês: *merge sort*), também conhecido por método ordenação por mistura, usa o conceito de dividir e conquistar para ordenar o vetor fornecido. Ele divide o problema em subproblemas menores até que eles se tornem simples o suficiente para serem resolvidos diretamente^[5].

Eis as etapas do método de ordenação por intercalação:

1. Dividir o vetor dado em duas metades (metades aproximadamente iguais no caso de uma lista com um número ímpar de elementos);
2. Dividir recursivamente os vetores menores da mesma maneira até que se obtenha um vetor menor com um único elemento;
3. Mesclar os vetores menores com um único elemento para que cada vetor menor mesclado seja ordenado;
4. Repetir a terceira etapa até que todo o vetor esteja ordenado.

Eis o código-fonte do método de ordenação por intercalação implementado:

Figura 8 - Método de ordenação por intercalação.

```
1  const merge = (left: number[], right: number[]): number[] => {
2    let array: number[] = [];
3
4    while (left.length && right.length) {
5      if (left[0] < right[0]) {
6        array.push(left.shift() as number);
7      } else {
8        array.push(right.shift() as number);
9      }
10   }
11
12   return [ ...array, ...left, ...right];
13 };
14
15 const mergeSort = (array: number[]): number[] => {
16   const half = array.length / 2;
17
18   if (array.length < 2) {
19     return array;
20   }
21
22   const left = array.splice(0, half);
23   return merge(mergeSort(left), mergeSort(array));
24 };
25
26 export default mergeSort;
```

Fonte: acervo pessoal.

3.6. MÉTODO DE ORDENAÇÃO POR SEPARAÇÃO

O método de ordenação por separação (em inglês: *quick sort*), tal como o método de ordenação por intercalação, também usa o conceito de dividir e conquistar. Ele divide os elementos em vetores menores com base em alguma condição e realiza as operações de ordenação nesses vetores menores divididos. Portanto, funciona bem para grandes conjuntos de dados^[6].

Eis as etapas do método de ordenação por separação:

1. Um elemento é selecionado para ser o elemento *pivot*;
2. Todos os elementos do vetor são comparados com o elemento *pivot* e organizados de tal forma que os elementos menores que o elemento *pivot* estejam à esquerda e os elementos maiores que o *pivot* estejam à direita;
3. Finalmente, as mesmas operações nos elementos do lado esquerdo e direito do elemento *pivot* são executadas.

Eis os códigos-fonte do método de ordenação por separação implementado:

Figura 9 - Função que seleciona o elemento *pivot* do método de ordenação por separação.

```
1 const partition = (array: number[], left: number, right: number) => {
2   const pivot = array[Math.floor((right + left) / 2)];
3   let i = left;
4   let j = right;
5
6   while (i <= j) {
7     while (array[i] < pivot) {
8       ++i;
9     }
10
11    while (array[j] > pivot) {
12      --j;
13    }
14
15    if (i <= j) {
16      const aux = array[i];
17      array[i] = array[j];
18      array[j] = aux;
19      ++i;
20      --j;
21    }
22  }
23  return i;
24 }
25 }
```

Fonte: acervo pessoal.

Figura 10 - Método de ordenação por separação.

```
1 const quickSort = (
2   array: number[],
3   left: number = 0,
4   right: number = array.length - 1
5 ): number[] => {
6   if (array.length > 1) {
7     const index = partition(array, left, right);
8
9     if (left < index - 1) {
10      quickSort(array, left, index - 1);
11    }
12
13    if (index < right) {
14      quickSort(array, index, right);
15    }
16  }
17
18  return array;
19 };
20
21 export default quickSort;
```

Fonte: acervo pessoal.

3.7. MÉTODO DE ORDENAÇÃO POR MONTE

O método de ordenação por monte (em inglês: *heap sort*) consiste em converter um vetor não ordenado em um *heap*. Um *heap*, por sua vez, é uma estrutura de dados semelhante a uma árvore. Há duas características distintas de um *heap*:^[7]

1. Um *heap* deve ser completo, o que significa que cada nível da árvore deve ser preenchido da esquerda para a direita, e não é permitido criar outro nível da árvore sem preencher todos os nós possíveis restantes no último nível;
2. Cada nó deve conter um valor maior ou igual (no caso de um *heap* mínimo, menor ou igual) ao valor de cada um de seus descendentes. Isso é chamado de "condição de *heap*".

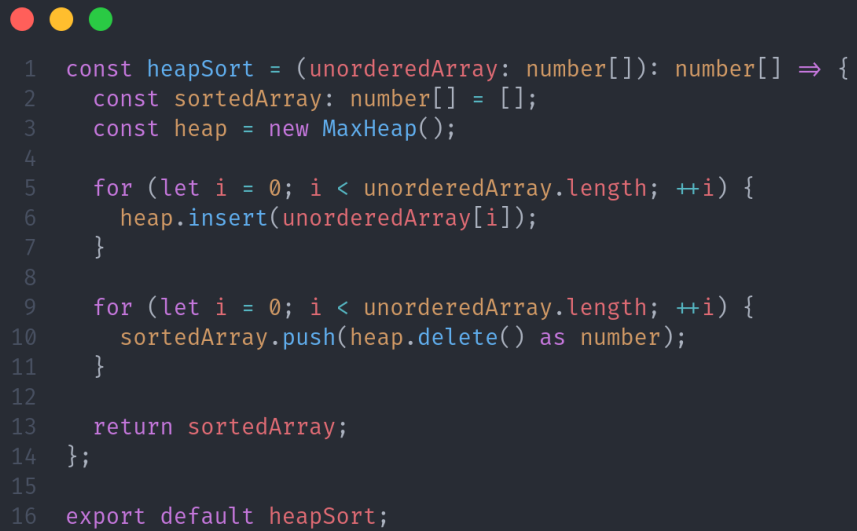
Eis os códigos-fonte do método de ordenação por monte implementado:

Figura 11 - Classe para instanciar um *heap* máximo.

```
1 class MaxHeap {
2   private _heap: number[] = [];
3
4   constructor() {
5     this._heap = [];
6   }
7
8   public get heap(): number[] {
9     return this._heap;
10  }
11
12  public set heap(value: number[]) {
13    this._heap = value;
14  }
15
16  parentIndex(index: number) {
17    return Math.floor((index - 1) / 2);
18  }
19
20  leftChildIndex(index: number) {
21    return 2 * index + 1;
22  }
23
24  rightChildIndex(index: number) {
25    return 2 * index + 2;
26  }
27
28  swap(a: number, b: number) {
29    const aux = this._heap[a];
30    this._heap[a] = this._heap[b];
31    this._heap[b] = aux;
32  }
33
34  insert(item: number) {
35    this._heap.push(item);
36
37    let index = this._heap.length - 1;
38    let parent = this.parentIndex(index);
39
40    while (this._heap[parent] < this._heap[index]) {
41      this.swap(parent, index);
42      index = this.parentIndex(index);
43      parent = this.parentIndex(index);
44    }
45  }
46
47  delete() {
48    const item = this._heap.shift();
49    this._heap.unshift(this._heap.pop() as number);
50
51    let index = 0;
52    let leftChild = this.leftChildIndex(index);
53    let rightChild = this.rightChildIndex(index);
54
55    while (
56      (this._heap[leftChild] > this._heap[index]) ||
57      (this._heap[rightChild] > this._heap[index])
58    ) {
59      let max = leftChild;
60
61      if (this._heap[rightChild] > this._heap[max]) {
62        max = rightChild;
63      }
64
65      this.swap(max, index);
66
67      index = max;
68      leftChild = this.leftChildIndex(index);
69      rightChild = this.rightChildIndex(index);
70    }
71
72    return item;
73  }
74 }
75
76 }
```

Fonte: acervo pessoal.

Figura 12 - Método de ordenação por monte.



```
1  const heapSort = (unorderedArray: number[]): number[] => {
2    const sortedArray: number[] = [];
3    const heap = new MaxHeap();
4
5    for (let i = 0; i < unorderedArray.length; ++i) {
6      heap.insert(unorderedArray[i]);
7    }
8
9    for (let i = 0; i < unorderedArray.length; ++i) {
10     sortedArray.push(heap.delete() as number);
11   }
12
13   return sortedArray;
14 };
15
16 export default heapSort;
```

Fonte: acervo pessoal.

4. RESULTADOS E DISCUSSÃO

A mensuração de desempenho começou no domingo do dia 09/10/2022, às 05:48, 4h atrasadas em relação ao Tempo Universal Coordenado, e terminou por volta das 09:00 do dia seguinte.

O código-fonte efetivo do arquivo principal, vide figura 4, foi projetado para executar todos os testes de uma só vez. Entretanto, após o método da bolha ordenar a primeira entrada pela quinta vez, ou seja, com $\frac{5}{120}$ ou 4,16% dos testes concluídos, já haviam decorrido seis horas, vinte e três minutos e vinte e cinco segundos, o que, juntamente com a temperatura elevada das partes internas da máquina, poderia desgastar os seus componentes eletrônicos. Isso posto, foi feita uma modificação no código-fonte do arquivo principal para interromper a mensuração de desempenho e retomá-la a partir de um teste específico.

Eis o novo código-fonte do arquivo principal implementado:

Figura 13 - Novo código-fonte do arquivo principal.



```
1  const sortingMethods = [
2    // bubbleSort,
3    // selectionSort,
4    insertionSort,
5    mergeSort,
6    quickSort,
7    heapSort
8  ];
9
10 let entry: number | undefined = 3;
11 let loop: number | undefined = 3;
12
13 for (const sortingMethod of sortingMethods) {
14   for (let i = entry || 1; i ≤ 4; ++i) {
15     const fileName = `entrada${i}.txt`;
16
17     for (let j = loop || 1; j ≤ 5; ++j) {
18       const label = `${sortingMethod.name}, entry No. ${i}, loop No. ${j}`;
19       const array = readArray(fileName);
20
21       console.time(label);
22       sortingMethod(array);
23       console.timeEnd(label);
24       console.log(new Date().toLocaleTimeString());
25       console.log('-'.repeat(label.length));
26     }
27
28     loop = undefined;
29   }
30
31   entry = undefined;
32 }
```

Fonte: acervo pessoal.

O novo código-fonte do arquivo principal segue as seguintes instruções:

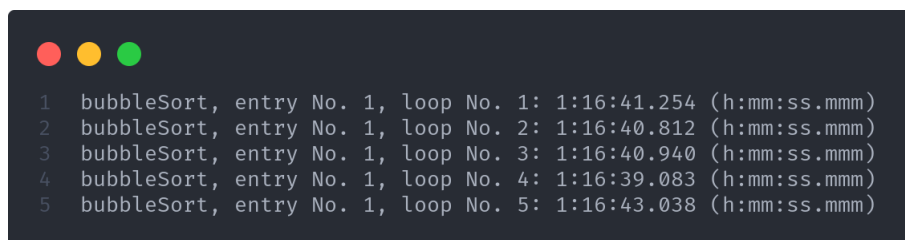
- Os métodos de ordenação já testados foram comentados no vetor dos métodos de ordenação;

- Foram declaradas as variáveis “*entry*” (do inglês: entrada) e “*loop*” (do inglês: iteração) antes do primeiro laço de repetição, as quais seguem as seguintes instruções:
 - Ambas as variáveis podem receber um valor numérico ou indefinido;
 - Quando há valor numérico, a expressão lógica na inicialização do segundo e terceiro laço de repetição, nas linhas quatorze e dezessete da figura 13, respectivamente, selecionará o valor à esquerda do operador lógico OR (do inglês: OU), ou seja, o valor da variável em questão;
 - Quando não há valor numérico, a expressão lógica selecionará o valor à direita do operador lógico OR, ou seja, o valor numérico definido por padrão.
- Após a conclusão de cada teste, são impressos a hora local e um tracejado para separar os testes um do outro.

No exemplo do código-fonte da figura 13, o último teste concluído foi a segunda ordenação da terceira entrada pelo método de ordenação por inserção. Logo, o próximo teste será a terceira ordenação da terceira entrada pelo método de ordenação por inserção. Quando a mesma entrada for ordenada cinco vezes e quando o método de ordenação em questão tiver ordenado todas as entradas, as variáveis “*entry*” e “*loop*” recebem o valor “*undefined*” (do inglês: indefinido), de modo a resetar a contagem de entradas e iterações.

Como não há como sobrescrever o método “*console.timeEnd*” de modo a armazenar automaticamente sua saída em um arquivo, cada teste foi copiado do terminal e colado em um arquivo de texto criado no diretório “*data/*”. Tal arquivo foi modificado durante as pausas da mensuração de desempenho, apenas.

Figura 14 - Cinco primeiras linhas do arquivo de texto contendo os resultados.



```

1 bubbleSort, entry No. 1, loop No. 1: 1:16:41.254 (h:mm:ss.mmm)
2 bubbleSort, entry No. 1, loop No. 2: 1:16:40.812 (h:mm:ss.mmm)
3 bubbleSort, entry No. 1, loop No. 3: 1:16:40.940 (h:mm:ss.mmm)
4 bubbleSort, entry No. 1, loop No. 4: 1:16:39.083 (h:mm:ss.mmm)
5 bubbleSort, entry No. 1, loop No. 5: 1:16:43.038 (h:mm:ss.mmm)

```

Fonte: acervo pessoal.

Após a conclusão dos cento e vinte testes, foi desenvolvida a função “*create sheets*” (em *camel case* e do inglês: criar planilhas), a qual lê o arquivo de texto contendo os resultados linha a linha, organiza os dados por método de ordenação, entrada e iteração; converte todos os resultados numéricos em milissegundos, calcula a média aritmética dos tempos de execução de cada método de ordenação por entrada ordenada e cria quatro planilhas respectivas às quatro entradas.

Figura 15 - Função de criar planilhas.

```
1 import { join } from 'path';
2 import { readFileSync, writeFileSync } from 'fs';
3
4 type Label = {
5   sortingMethod: string;
6   entry: number;
7   loop: number;
8 };
9
10 const sortingMethods = {
11   bubbleSort: 'Método da bolha',
12   selectionSort: 'Método de ordenação por seleção',
13   insertionSort: 'Método de ordenação por inserção',
14   mergeSort: 'Método de ordenação por intercalação',
15   quickSort: 'Método de ordenação por separação',
16   heapSort: 'Método de ordenação por monte'
17 };
18
19 const createSheets = () => {
20   let path = join(__dirname, '..', 'data', 'result.txt');
21   const runTimes = readFileSync(path, 'utf-8').split('\n');
22   const header = [
23     'Método de ordenação',
24     ...['primeira', 'segunda', 'terceira', 'quarta', 'quinta'].map((value: string) => `Tempo da ${value} execução (ms)`),
25     'Média aritmética'
26   ].map((value: string) => `${value}`).join(',') + '\n';
27   const map = new Map<string, number>();
28   runTimes.forEach((runtime: string) => {
29     const runtimeArray = runtime.split(' ');
30     const sortingMethod = runtimeArray[0].slice(0, -1);
31     const entry = Number(runtimeArray[3].slice(0, -1));
32     const loop = Number(runtimeArray[6].slice(0, -1));
33     const label = { sortingMethod, entry, loop } as Label;
34     let value: number = 0;
35     if (runtimeArray.length === 9) {
36       const [milliseconds, seconds, minutes, hours] = runtimeArray[7].split(/[:\.]/).reverse().map((value: string) => Number(value));
37       if (hours !== undefined) value += hours * 60 * 60 * 1000;
38       value += minutes * 60 * 1000;
39       value += seconds * 1000;
40       value += milliseconds;
41     } else {
42       value = Number(runtimeArray[7].slice(0, -2));
43     }
44     map.set(JSON.stringify(label), value);
45   });
46   const sheets: string[] = Array(4).fill(header);
47   for (let i = 0; i < 4; ++i) {
48     for (const rawSortingMethod of Object.keys(sortingMethods)) {
49       const sortingMethod = sortingMethods[rawSortingMethod as keyof typeof sortingMethods];
50       let average: number = 0;
51       sheets[i] += `${sortingMethod},"`;
52       for (let j = 1; j ≤ 5; ++j) {
53         const label = { sortingMethod: rawSortingMethod, entry: (i + 1), loop: j } as Label;
54         const value = map.get(JSON.stringify(label)) as number;
55         sheets[i] += `${value.toLocaleString()},"`;
56         average += value;
57       }
58       average /= 5;
59       sheets[i] += `${Number(average.toFixed(3)).toLocaleString()}\n`;
60     }
61     path = join(__dirname, '..', 'sheets', `results-over-entry-${(i + 1)}.csv`);
62     writeFileSync(path, sheets[i]);
63   }
64 }
65
66 createSheets();
```

Fonte: acervo pessoal.

Eis os resultados organizados:

Tabela 4 - Tempo de execução dos métodos de ordenação para um vetor de um milhão de números desordenados.

Método de ordenação	Tempo da primeira execução (ms)	Tempo da segunda execução (ms)	Tempo da terceira execução (ms)	Tempo da quarta execução (ms)	Tempo da quinta execução (ms)	Média aritmética
Método da bolha	4.601.254	4.600.812	4.600.940	4.599.083	4.603.038	4.601.025,4
Método de ordenação por seleção	1.066.599	1.068.000	1.066.845	1.068.035	1.066.382	1.067.172,2
Método de ordenação por inserção	646.485	646.683	647.536	646.732	647.291	646.945,4
Método de ordenação por intercalação	246.502	246.210	244.661	247.705	247.646	246.544,8
Método de ordenação por separação	233,39	225	223,11	223,39	223,45	225,67
Método de ordenação por monte	868.813	868.313	871.832	949.342	886.257	888.911,4

Tabela 5 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente.

Método de ordenação	Tempo da primeira execução (ms)	Tempo da segunda execução (ms)	Tempo da terceira execução (ms)	Tempo da quarta execução (ms)	Tempo da quinta execução (ms)	Média aritmética
Método da bolha	651.630	652.658	652.454	651.178	652.328	652.049,6
Método de ordenação por seleção	257.027	257.861	257.975	258.251	257.338	257.690,4
Método de ordenação por inserção	2,595	2,938	2,593	2,602	2,62	2,67
Método de ordenação por intercalação	21,33	21,5	21,25	21,16	21,3	21,308
Método de ordenação por separação	50,639	51,041	72,414	50,804	50,648	55,109
Método de ordenação por monte	104.786	107.411	108.519	105.602	103.629	105.989,4

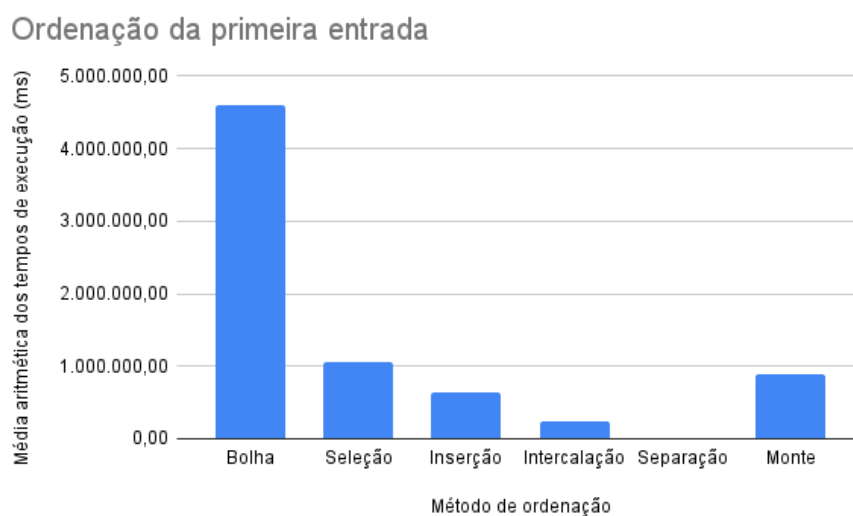
Tabela 6 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem decrescente.

Método de ordenação	Tempo da primeira execução (ms)	Tempo da segunda execução (ms)	Tempo da terceira execução (ms)	Tempo da quarta execução (ms)	Tempo da quinta execução (ms)	Média aritmética
Método da bolha	714.828	713.988	713.839	713.647	713.568	713.974
Método de ordenação por seleção	564.926	564.691	574.777	575.399	572.410	570.440,6
Método de ordenação por inserção	320.337	320.736	320.482	320.681	320.203	320.487,8
Método de ordenação por intercalação	21,19	21,37	21,48	21,3	21,49	21,366
Método de ordenação por separação	51,59	51,631	80,041	51,659	51,63	57,31
Método de ordenação por monte	105.620	104.168	105.914	108.442	105.606	105.950

Tabela 7 - Tempo de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente, com exceção do primeiro.

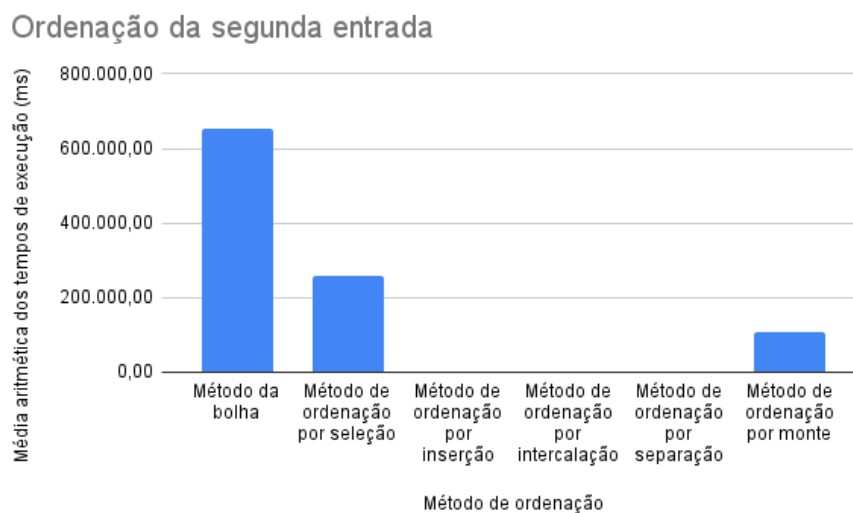
Método de ordenação	Tempo da primeira execução (ms)	Tempo da segunda execução (ms)	Tempo da terceira execução (ms)	Tempo da quarta execução (ms)	Tempo da quinta execução (ms)	Média aritmética
Método da bolha	652.394	653.430	653.469	652.355	653.110	652.951,6
Método de ordenação por seleção	257.518	257.492	257.507	258.726	258.029	257.854,4
Método de ordenação por inserção	289,083	295,184	287,708	287,529	288,324	289,566
Método de ordenação por intercalação	21,72	21,64	21,55	21,66	21,5	21,614
Método de ordenação por separação	64,228	68,137	78,706	63,302	63,156	67,506
Método de ordenação por monte	105.786	104.424	105.573	104.835	104.960	105.115,6

Gráfico 1 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de um milhão de números desordenados.



Fonte: acervo pessoal.

Gráfico 2 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente.



Fonte: acervo pessoal.

Gráfico 3 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem decrescente.



Fonte: acervo pessoal.

Gráfico 4 - Média aritmética dos tempos de execução dos métodos de ordenação para um vetor de quinhentos mil números ordenados em ordem crescente, com exceção do primeiro.



Fonte: acervo pessoal.

Ratificado pelos gráficos comprobatórios acima, o método de ordenação por separação se sobressaiu em todas as entradas, levando no pior cenário proposto pelo trabalho apenas 225,67 milissegundos para ordenar o vetor dado em ordem crescente. Além disso, a performance do método de ordenação por inserção foi superior à performance do método de ordenação por separação na segunda entrada, na qual o vetor já está previamente ordenado em ordem crescente.

5. CONSIDERAÇÕES FINAIS

Conclui-se que o método de ordenação mais eficiente para se implementar sob quaisquer circunstâncias, especialmente quando não se tem detalhes da entrada, é o método de ordenação por separação (em inglês: *quick sort*).

6. REFERÊNCIAS

1. ALICE, Akemi. **Convenções de nomenclatura: Camel, Pascal, Kebab e Snake case.** [S. l.]: Alura, 9 mar. 2022. Disponível em: <https://www.alura.com.br/artigos/convencoes-nomenclatura-camel-pascal-kebab-snake-case>. Acesso em: 7 out. 2022;
2. APIS da Web: Console.time(). [S. l.]: MDN Web Docs, 26 set. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/console/time>. Acesso em: 9 out. 2022;
3. LUKIC, Mila. **Selection Sort in JavaScript.** [S. l.]: Stack Abuse, 13 nov. 2020. Disponível em: <https://stackabuse.com/selection-sort-in-javascript>. Acesso em: 9 out. 2022;
4. LUKIC, Mila. **Insertion Sort in JavaScript.** [S. l.]: Stack Abuse, 27 mai. 2020. Disponível em: <https://stackabuse.com/insertion-sort-in-javascript>. Acesso em: 10 out. 2022;
5. KAKUMANU, Abhilash. **Merge Sort in JavaScript.** [S. l.]: Stack Abuse, 12 out. 2020. Disponível em: <https://stackabuse.com/merge-sort-in-javascript>. Acesso em: 10 out. 2022;
6. HARTMAN, James. **QuickSort Algorithm in JavaScript.** [S. l.]: Guru99, 25 ago. 2022. Disponível em: <https://www.guru99.com/quicksort-in-javascript.html>. Acesso em: 10 out. 2022;
7. GÜLER, Cansın. **Heap Sort in JavaScript.** [S. l.]: Stack Abuse, 16 nov. 2021. Disponível em: <https://stackabuse.com/heap-sort-in-javascript>. Acesso em: 10 out. 2022.