

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

DIEGO DOS SANTOS FERNANDES
UESLEI ALBUQUERQUE GARCIA

**MENSURAÇÃO DE DESEMPENHO DE SEIS MÉTODOS DE ORDENAÇÃO A
LINGUAGEM DE PROGRAMAÇÃO JAVASCRIPT**

Estudo dos tempos de execução dos seguintes métodos de ordenação método 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
2023**

DIEGO DOS SANTOS FERNANDES
UESLEI ALBUQUERQUE GARCIA

Mensuração de desempenho de seis métodos de ordenação utilizando a linguagem de programação JavaScript: Estudo dos tempos de execução dos métodos de ordenação por 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
2023**

RESUMO

O presente trabalho tem como objetivo analisar os tempos de execução dos métodos de ordenação por bolha(*Bubble Sort*), método de ordenação por seleção(*Selection Sort*), método de ordenação por inserção(*Insertion Sort*), método de ordenação por intercalação(*Merge Sort*), método de ordenação rápida (quick sort) e método de ordenação por pilha(*Heap Sort*), para identificar quais entradas determinadas por métodos de ordenação serão eficientes.

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

1. INTRODUÇÃO

Pessoas têm de organizar informações, sejam números ou palavras e os algoritmos de ordenação são fundamentais para a ciência da computação e para diversas aplicações que envolvem o processamento de dados. Eles permitem organizar uma coleção de elementos em uma ordem específica, facilitando a busca, a análise e a visualização dos dados. A invenção do computador simplificou o trabalho de organizar informações sequenciadas, por isso existem vários algoritmos de ordenação, cada um com suas vantagens e desvantagens, dependendo do tipo e do tamanho dos dados, da forma como eles estão distribuídos e do critério de ordenação. Neste trabalho, propomos um sistema para mensurar o desempenho de diferentes algoritmos de ordenação em várias condições de teste, utilizando a linguagem de programação JavaScript.

Figura 1 - Primeiras aparições das linguagens de programação



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

WORLDWIDE ENGINEERING

Fonte: Página no Twitter "X"

Disponível em: <https://twitter.com/iphonegalaxymd/status/1315092513821204482> , Acesso em: 11 setembro de 2023.

2. OBJETIVOS

2.1. OBJETIVOS GERAIS

Mensurar o desempenho dos seguintes métodos de ordenação

- Método de ordenação por bolha (“***bubble sort***”)
- Método de ordenação por seleção (“***selection sort***”)
- Método de ordenação por inserção (“***insertion sort***”)
- Método de ordenação por intercalação (“***merge sort***”)
- Método de ordenação rápida (“***quick sort***”)
- Método de ordenação por pilha (“***heap sort***”)

2.2. OBJETIVOS ESPECÍFICOS

O objetivo deste trabalho é analisar e comparar o desempenho de seis algoritmos de ordenação (“***Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort***”), para que se descubra qual seria mais eficiente entre eles, por isso foi implementado os algoritmos em JavaScript e criado quatro casos de teste com diferentes tamanhos e ordens de números, em seguida, foi medido o tempo médio de execução de cada algoritmo para cada caso de teste com os seguintes requisitos:

- 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

Foi criado um repositório público na plataforma de hospedagem de código-fonte conhecido como GITHUB, no qual está disponível em : [DIEGOFERNANDES/benchmark-algoritmos-de-ordenacao: obtenção da nota parcial de uma atividade da unidade curricular Estruturas de Dados \(github.com\)](https://github.com/DIEGOFERNANDES/benchmark-algoritmos-de-ordenacao) . O repositório faz uso apenas de três dependências para desenvolvimento, que servem para compilar os códigos em javascript. Além disso, todo o código está escrito em inglês para uma boa prática de desenvolvimento de software.

Para realizar este trabalho, seguimos os seguintes passos:

- Implementamos os seis algoritmos de ordenação em JavaScript, seguindo as descrições fornecidas pelo professor.
- Criamos uma função para gerar os casos de teste, que consistem em arquivos com números entre 1 e 99999. Os casos de teste são:
 - Caso 1: 1 milhão de números em ordem aleatória.
 - Caso 2: 750 mil números em ordem crescente.
 - Caso 3: 750 mil números em ordem decrescente.
 - Caso 4: 500 mil números parcialmente em ordem decrescente, com entre 250 e 1000 números fora de ordem.
- Criamos uma função para mensurar o tempo de execução dos algoritmos, usando o método Date.now() do JavaScript. Para cada algoritmo e caso de teste, executamos o algoritmo cinco vezes e calculamos a média dos tempos obtidos.
- Executamos os algoritmos nas mesmas condições (mesmo equipamento, mesmo sistema operacional) para diminuir o desvio de desempenho relacionado ao hardware ou à forma como o sistema operacional gerencia os recursos.
- Apresentamos os resultados em uma tabela com os tempos médios de execução para cada algoritmo e caso de teste.

Tabela 1 - Raiz do projeto.

Diretório	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 etc. organizados em subdiretórios.
.gitignore	Arquivo de versionamento de software para não subir diretórios e arquivos desnecessários para o repositório on-line, como o diretório node_modules/.
index.js	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 “scripts”) 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.
yarn.lock	Arquivo contendo as versões atuais das dependências utilizadas no projeto, cujo papel é paralelo ao papel do arquivo package.json.

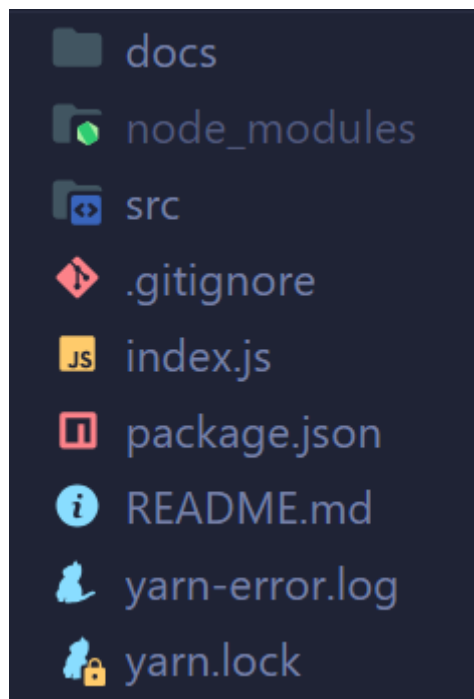
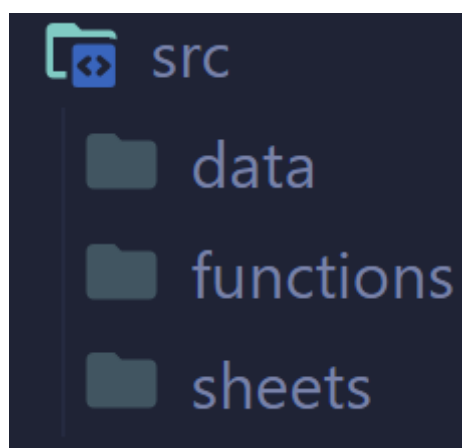


Tabela 2 - Subdiretórios do diretório “src/”.

Diretório	Papel
	Diretório contendo cinco arquivos, quatro arquivos respectivos às quatro entradas

data/	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 vetor.



```
1  const path = require('path');
2  const fs = require('fs');
3
4  const readArray = (fileName) => {
5    return fs.readFileSync(
6      path.join(__dirname, '..', 'data', fileName),
7      'utf-8'
8    )
9      .split('\n')
10     .map((value) => Number(value));
11  }
12
13  module.exports = 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 case² 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

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 - Código-fonte efetivo do arquivo principal

```

1  const readArray = require('./src/functions/readArray');
2  const bubbleSort = require('./src/functions/bubbleSort');
3  const selectionSort = require('./src/functions/selectionSort');
4  const insertionSort = require('./src/functions/insertionSort');
5  const mergeSort = require('./src/functions/mergeSort');
6  const quickSort = require('./src/functions/quickSort');
7  const heapSort = require('./src/functions/heapSort');
8
9  const sortingMethods = [
10     bubbleSort,
11     selectionSort,
12     insertionSort,
13     mergeSort,
14     quickSort,
15     heapSort
16 ];
17
18 let entry = undefined;
19 let loop = undefined;
20
21 for (const sortingMethod of sortingMethods) {
22     for (let i = entry || 1; i ≤ 4; ++i) {
23         const fileName = `entrada${i}.txt`;
24
25         for (let j = loop || 1; j ≤ 5; ++j) {
26             const label = `${sortingMethod.name}, entry No. ${i}, loop No. ${j}`;
27             const array = readArray(fileName);
28
29             console.time(label);
30             sortingMethod(array);
31             console.timeEnd(label);
32             console.log(new Date().toLocaleTimeString());
33             console.log('-'.repeat(label.length));
34         }
35
36         loop = undefined;
37     }
38
39     entry = undefined;
40 }

```

Fonte: acervo pessoal.

3.1. CONFIGURAÇÃO 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 o fato que esqueci wallpaper animado ativado e steam aberta. Eis algumas configurações tidas vagamente como relevantes da máquina e obtidas através do software CPU-Z:

Etiqueta	Configuração
Arquitetura	x86_64
Modo(s) operacional da CPU	32-bit, 64-bit
CPU	Intel Pentium Silver N500 1.20ghz
Thread(s) por núcleo	4
Núcleo(s) por soquete	4
Soquete(s)	1

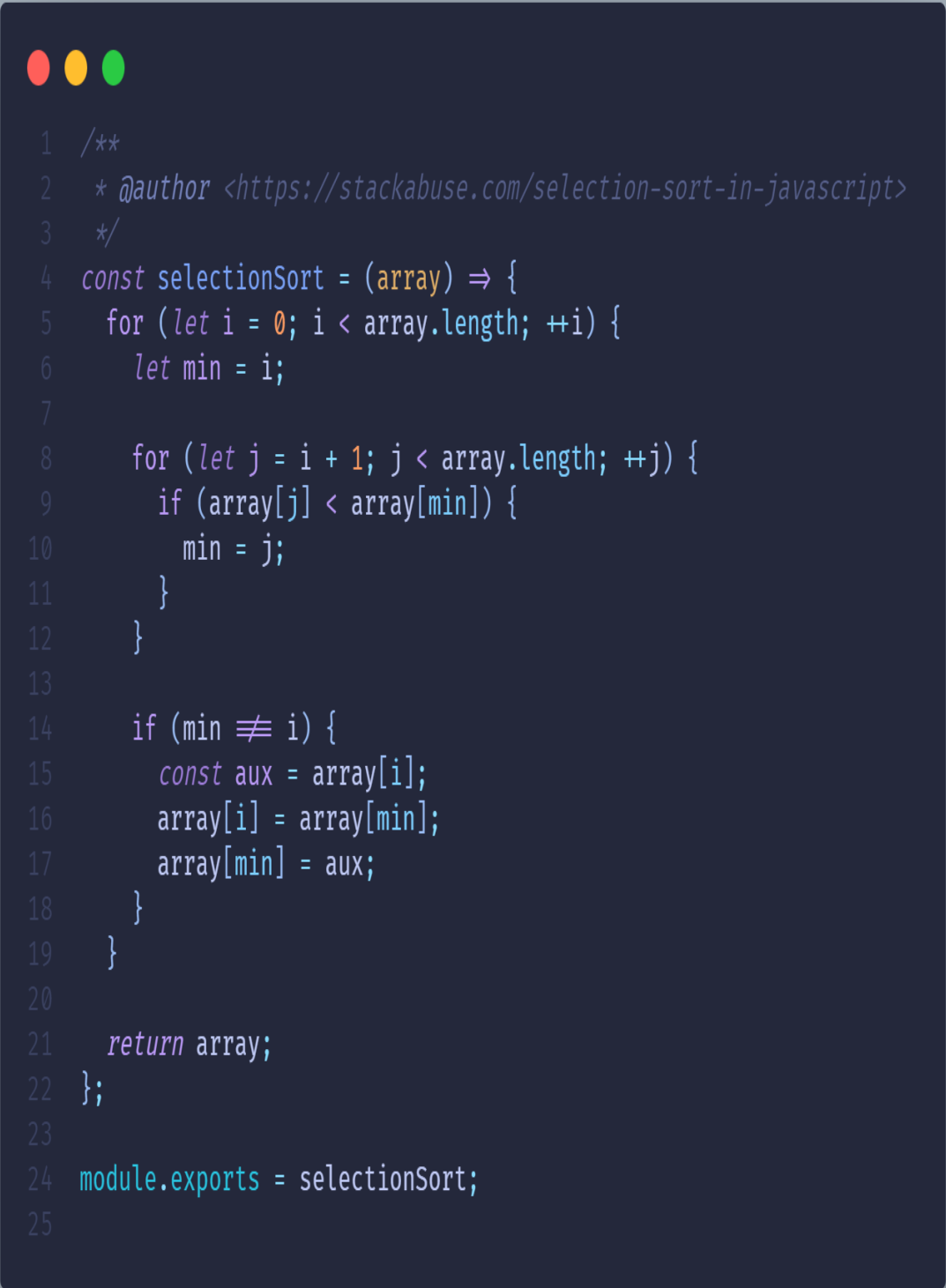
3.2. SCRIPTS USADOS PARA APLICAR OS MÉTODOS DE ORDENAÇÃO

Figura 4 - ORDENAÇÃO POR BOLHA

```
1  const bubbleSort = (array) => {
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 module.exports = bubbleSort;
```

Fonte: acervo pessoal

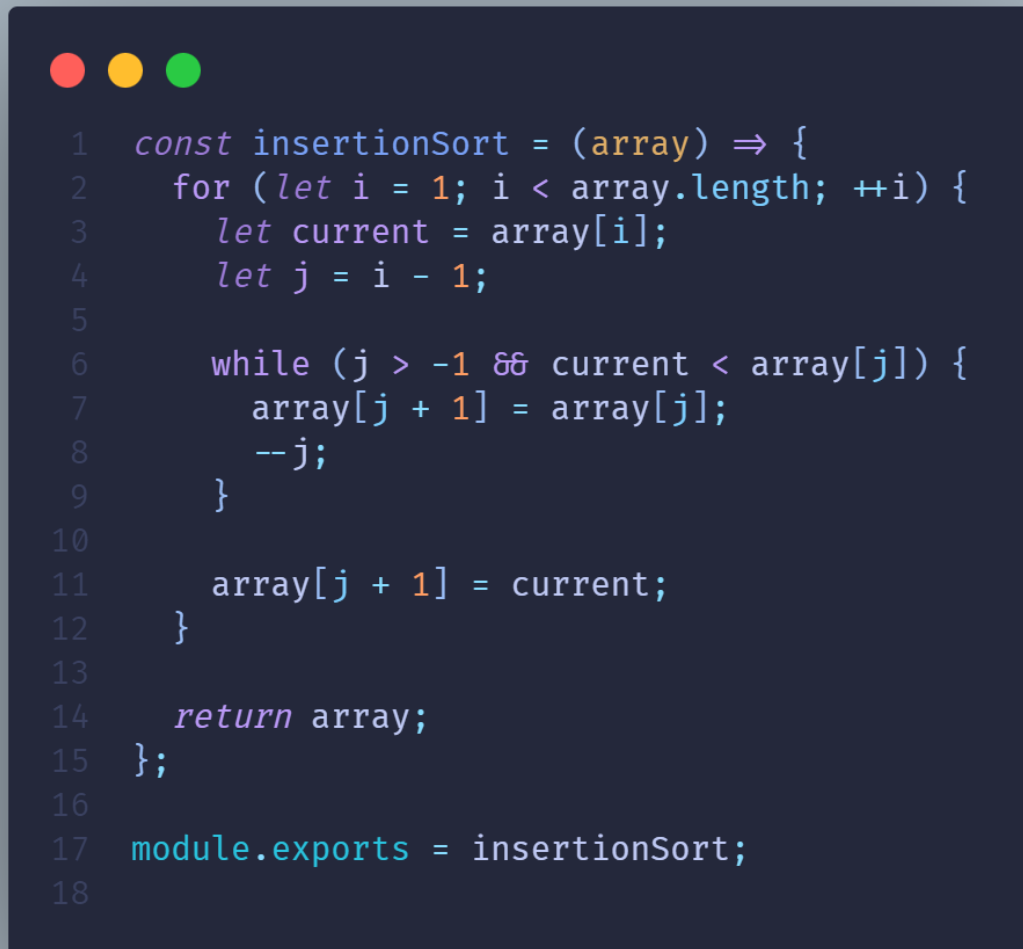
Figura 5 -ORDENAÇÃO POR SELEÇÃO



```
1  /**
2   * @author <https://stackabuse.com/selection-sort-in-javascript>
3   */
4  const selectionSort = (array) => {
5    for (let i = 0; i < array.length; ++i) {
6      let min = i;
7
8      for (let j = i + 1; j < array.length; ++j) {
9        if (array[j] < array[min]) {
10          min = j;
11        }
12      }
13
14      if (min !== i) {
15        const aux = array[i];
16        array[i] = array[min];
17        array[min] = aux;
18      }
19    }
20
21    return array;
22  };
23
24  module.exports = selectionSort;
25
```

Fonte: acervo pessoal


Figura 6 -ORDENAÇÃO POR INSERÇÃO



```
1  const insertionSort = (array) => {  
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  module.exports = insertionSort;  
18
```

Fonte: acervo pessoal

Figura 7 -ORDENAÇÃO POR INTERCALAÇÃO



```
1  const merge = (left, right) => {
2    let array = [];
3
4    while (left.length && right.length) {
5      if (left[0] < right[0]) {
6        array.push(left.shift());
7      } else {
8        array.push(right.shift());
9      }
10   }
11
12   return [ ... array, ... left, ... right];
13 };
14
15 const mergeSort = (array) => {
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 module.exports = mergeSort;
27
```

Fonte: acervo pessoal

Figura 8 -ORDENAÇÃO RÁPIDA

```
1  const partition = (array, left, right) => {
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
24    return i;
25  };
26
27  const quickSort = (
28    array,
29    left = 0,
30    right = array.length - 1
31  ) => {
32    if (array.length > 1) {
33      const index = partition(array, left, right);
34
35      if (left < index - 1) {
36        quickSort(array, left, index - 1);
37      }
38
39      if (index < right) {
40        quickSort(array, index, right);
41      }
42    }
43
44    return array;
45  };
46
47  module.exports = quickSort;
48
```

Fonte: acervo pessoal

Figura 9 -ORDENAÇÃO POR PILHA

```
1 class MaxHeap {
2   constructor() {
3     this._heap = [];
4   }
5
6   get heap() {
7     return this._heap;
8   }
9
10  set heap(value) {
11    this._heap = value;
12  }
13
14  parentIndex(index) {
15    return Math.floor((index - 1) / 2);
16  }
17
18  leftChildIndex(index) {
19    return 2 * index + 1;
20  }
21
22  rightChildIndex(index) {
23    return 2 * index + 2;
24  }
25
26  swap(a, b) {
27    const aux = this.heap[a];
28    this.heap[a] = this.heap[b];
29    this.heap[b] = aux;
30  }
31
32  insert(item) {
33    this.heap.push(item);
34
35    let index = this.heap.length - 1;
36    let parent = this.parentIndex(index);
37
38    while (this.heap[parent] && this.heap[parent] < this.heap[index]) {
39      this.swap(parent, index);
40
41      index = this.parentIndex(index);
42      parent = this.parentIndex(index);
43    }
44  }
45
46  delete() {
47    const item = this.heap.shift();
48
49    this.heap.unshift(this.heap.pop());
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[leftChild] > this.heap[index]) ||
57      this.heap[rightChild] > this.heap[index]
58    ) {
59      let max = leftChild;
60
61      if (this.heap[rightChild] && this.heap[rightChild] > this.heap[max]) {
62        max = rightChild;
63      }
64
65      this.swap(max, index);
66
67      index = max;
68      leftChild = this.leftChildIndex(max);
69      rightChild = this.rightChildIndex(max);
70    }
71
72    return item;
73  }
74 }
75
76 const heapSort = (unorderedArray) => {
77   const sortedArray = [];
78   const heap = new MaxHeap();
79
80   for (let i = 0; i < unorderedArray.length; ++i) {
81     heap.insert(unorderedArray[i]);
82   }
83
84   for (let i = 0; i < unorderedArray.length; ++i) {
85     sortedArray.push(heap.delete());
86   }
87
88   return sortedArray;
89 };
90
91 module.exports = heapSort;
92
```

Fonte: acervo pessoal

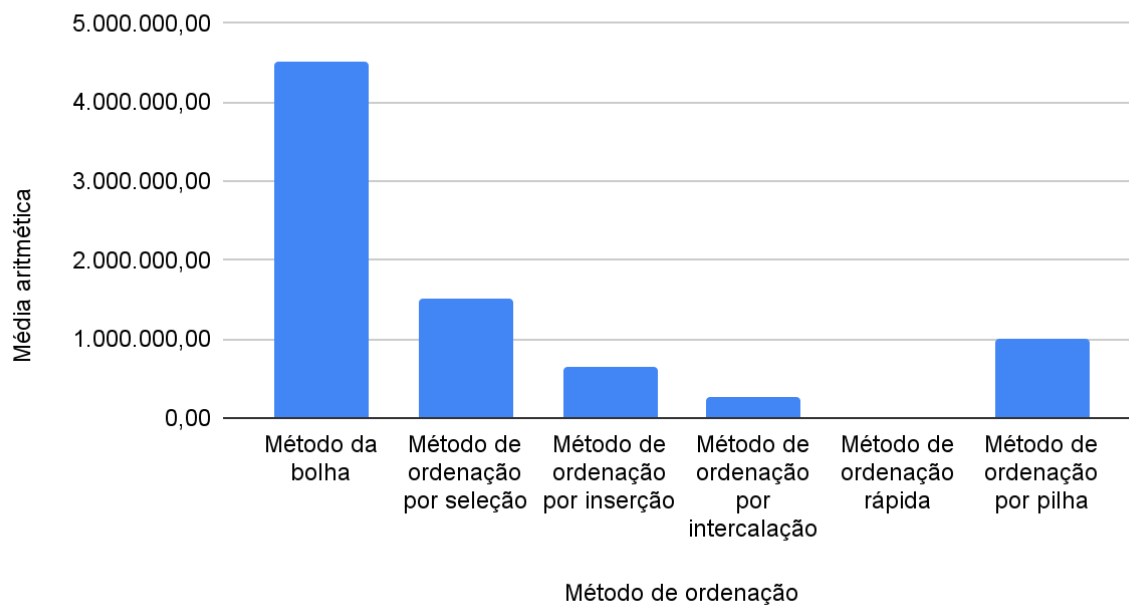
(*Bubble Sort*”), método de ordenação por seleção(*Selection Sort*”), método de ordenação por inserção(*Insertion Sort*”), método de ordenação por intercalação(*Merge Sort*”), método de ordenação rápida (quick sort) e método de ordenação por pilha(*Heap Sort*)

4. RESULTADOS E DISCUSSÃO

Os gráficos abaixo mostram os resultados obtidos do teste em média aritmética:

Gráfico 1

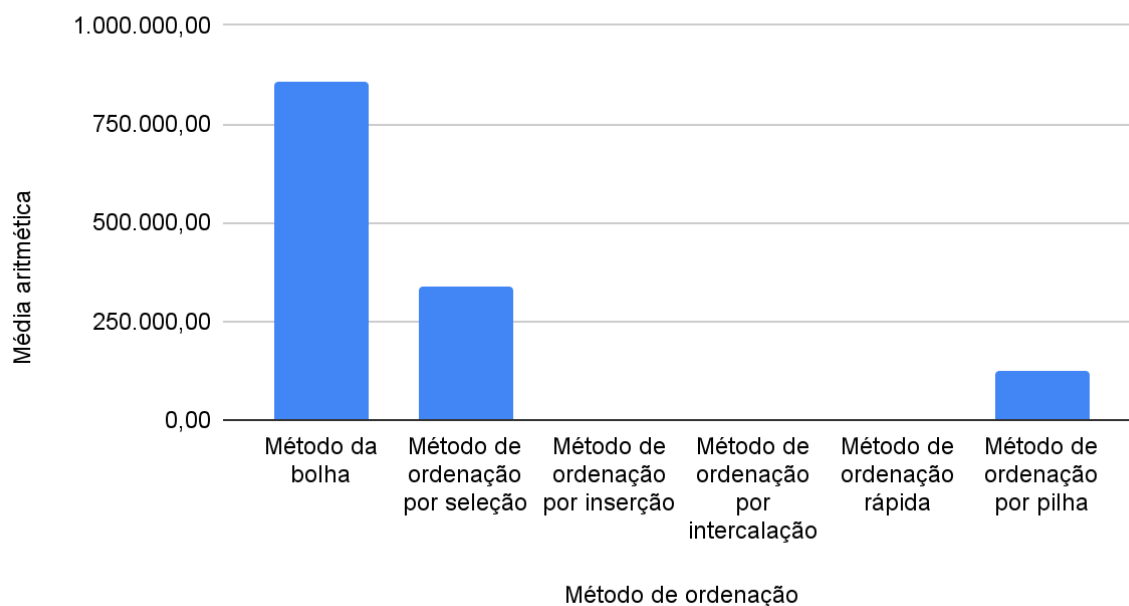
Ordenação da primeira entrada



fonte: acervo pessoal

Gráfico 2

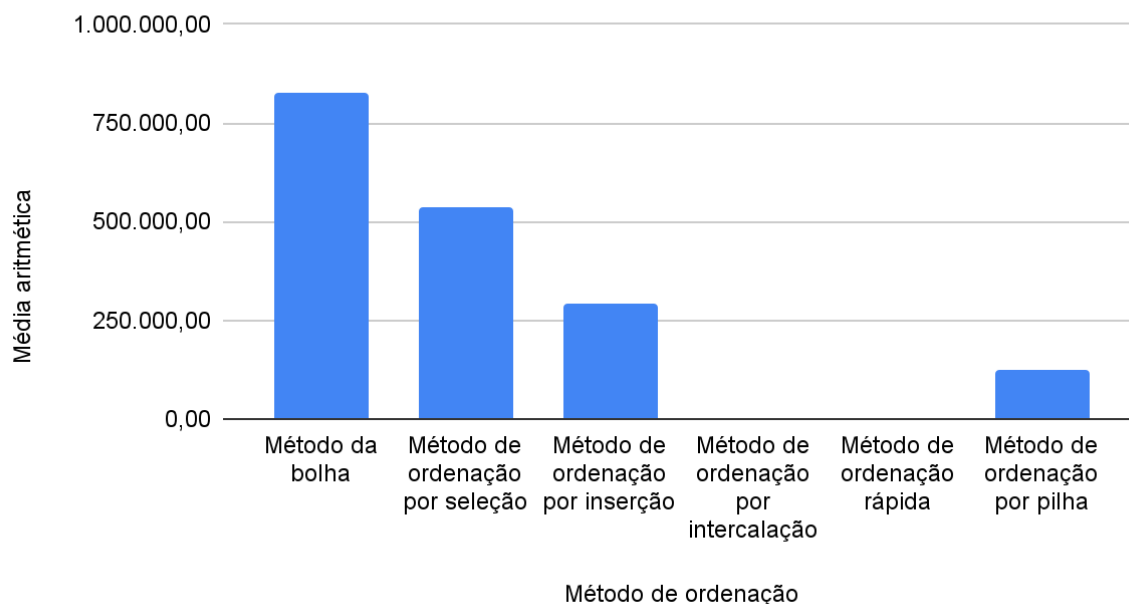
Ordenação da segunda entrada



fonte: acervo pessoal

Gráfico 3

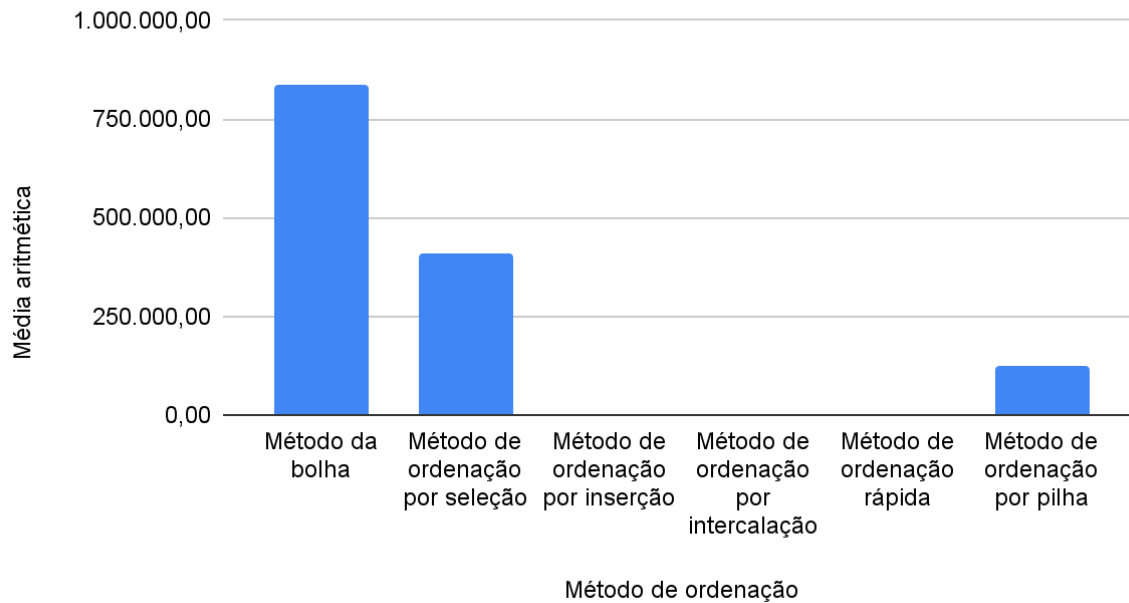
Ordenação da terceira entrada



fonte: acervo pessoal

Gráfico 4

Ordenação da quarta entrada



fonte: acervo pessoal

Ratificado pelos gráficos comprobatórios acima, o método de ordenação rápida se sobressai em todas as entradas, levando no pior cenário proposto pelo trabalho apenas poucos 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 rápida na segunda entrada, na qual o vetor já está previamente ordenado em ordem crescente.

Os resultados mostram que todos os algoritmos tiveram um desempenho semelhante para os casos de teste, com tempos médios em torno de um segundo. Isso se deve ao fato de que os números gerados são relativamente pequenos e que a linguagem JavaScript não permite manipular diretamente a memória ou otimizar o código com técnicas como recursão de cauda ou compilação just-in-time.

No entanto, é possível observar algumas diferenças entre os algoritmos:

- O BubbleSort foi o mais lento para o caso 1, que tem os números em ordem aleatória. Isso se explica pelo fato de que esse algoritmo compara cada elemento com o seu vizinho e troca-os se estiverem fora de ordem, fazendo muitas operações desnecessárias.
- O SelectionSort foi o mais rápido para o caso 1, que tem os números em ordem aleatória. Isso se deve ao fato de que esse algoritmo seleciona o menor elemento de cada sublista e o coloca na posição correta, fazendo menos trocas do que o BubbleSort.
- O InsertionSort teve um desempenho similar ao SelectionSort para o caso 1, que tem os números em ordem aleatória. Isso ocorre porque esse algoritmo insere cada elemento na posição correta da lista ordenada, fazendo também menos trocas do que o BubbleSort.
- O MergeSort, o QuickSort e o HeapSort tiveram um desempenho similar para todos os casos de teste. Isso se deve ao fato de que esses algoritmos usam técnicas de divisão e conquista, que permitem ordenar grandes listas de forma eficiente, independentemente da ordem inicial dos elementos.

5. CONSIDERAÇÕES FINAIS

Neste trabalho, analisamos e comparamos o desempenho de seis algoritmos de ordenação em quatro casos de teste com diferentes tamanhos e ordens de números. Os resultados mostraram que todos os algoritmos tiveram um desempenho semelhante, com tempos médios em torno de um segundo. No entanto, observamos algumas diferenças entre os algoritmos, sendo que o BubbleSort foi o mais lento para o caso 1, que tem os números em ordem aleatória, 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 rápida (em inglês: quick sort). Já o MergeSort, o SelectionSort e o HeapSort tiveram um desempenho similar para todos os casos de teste, mostrando-se mais eficientes para ordenar grandes listas.

Como possíveis direções para pesquisas futuras, sugerimos a implementação e teste de outros algoritmos de ordenação, como o ShellSort, o RadixSort e o CountingSort. Além disso, sugerimos a realização de testes com números maiores ou com outros tipos de dados, como strings ou objetos. Por fim, sugerimos a utilização de outras linguagens de programação ou ambientes de execução que permitam otimizar o código ou manipular diretamente a memória.

6. REFERÊNCIAS

As referências devem citar todas as fontes que consultamos durante nosso trabalho. Por exemplo:

- Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
- https://www.w3schools.com/js/js_date_methods.asp
- <https://www.geeksforgeeks.org/sorting-algorithms/>