Algoritmos e Estruturas de Dados: Algoritmos de Ordenação

Ludmila Carina da Silva

Bacharelado em Sistemas de Informação
Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais – Ouro Branco

***Resumo**: Este estudo tem como objetivo avaliar os dados gerados por meio de experimentação com os algoritmos de ordenação Selection Sort, Insertion Sort, Bubble Sort, Merge Sort e Quick Sort, bem como analisar sua complexidade e eficiência. Foram realizados testes em coleções ordenadas de forma crescente, decrescente e aleatória, utilizando sete vetores de diferentes tamanhos para cada algoritmo, e os dados coletados foram posteriormente comparados.*

***Abstract**: This study aims to evaluate the data generated through experimentation with the sorting algorithms Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort, as well as to analyze their complexity and efficiency. Tests were conducted on collections sorted in ascending, descending, and random order, using seven arrays of different sizes for each algorithm, and the collected data was subsequently compared.*

1. Introdução

Os algoritmos de ordenação são um conjunto de instruções que organizam uma coleção de itens em uma ordem específica, comumente numérica ou alfabética. No entanto, a aplicação desses algoritmos não se limita apenas a esses domínios, estendendo-se a uma variedade de campos, incluindo algoritmos de busca, bancos de dados, métodos de divisão e conquista, estruturas de dados e muito mais.

Ao escolher um algoritmo de ordenação, é importante considerar vários fatores como o tamanho da coleção a ser ordenada, a disponibilidade de memória e as restrições específicas do sistema. Essa escolha estratégica se adapta ao contexto do sistema, permitindo o uso eficiente dos recursos disponíveis.

O estudo se concentra na avaliação e comparação do desempenho dos algoritmos de ordenação, incluindo o Selection Sort, Insertion Sort, Bubble Sort, Merge Sort e Quick Sort, em quatro cenários distintos. Foram realizados testes em coleções ordenadas de forma crescente, decrescente e aleatória, em diferentes tamanhos de vetores.

2. Algoritmos de Ordenação

2.1. Selection Sort

```
public class AlgoritmoSelectionSort {
   public static void main(String[] args) {

      int tamanhoVetor = 500000;
      int vetor[] = new int[tamanhoVetor];
      int intervaloMin = 1; // VALOR MINIMO DOS NUMEROS ALEATORIOS
      int intervaloMax = 500000; // VALOR MAXIMO DOS NUMEROS ALEATORIOS
      int indice;
      int ponteiro; // INICIALIZA MARCADOR DE POSICOES
      int min, aux;

      Random random = new Random(12345);

      for (int i = 0; i < tamanhoVetor; i++) {
         // GERA UM NUMERO ALEATORIO ENTRE INTERVALO MIN E MAX (inclusive)
         vetor[i] = random.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
      }

      // TEMPO DE EXECUCAO INICIA AQUI
      long startTime = System.nanoTime();

      for (indice = 0; indice < (tamanhoVetor - 1); indice++) {
         min = indice;
         for (ponteiro = (indice + 1); ponteiro < tamanhoVetor; ponteiro++) {
            if (vetor[ponteiro] < vetor[min]) {
               min = ponteiro;
            }
         }
         if (vetor[indice] != vetor[min]) {
            aux = vetor[indice];
            vetor[indice] = vetor[min];
            vetor[min] = aux;
         }
      }

      long endTime = System.nanoTime();
      long duration = (endTime - startTime);  // TEMPO DE EXECUCAO EM NANOSSEGUNDOS
      double milliseconds = (double) duration / 1_000_000.0; // CONVERTE EM MILISEGUNDOS
      System.out.println("Tempo de execucao: " + milliseconds + " milisegundos.");

      // IMPRIME O VETOR ORDENADO
      for (int i = 0; i < tamanhoVetor; i++) {
         System.out.print(vetor[i] + " ");
      }
   }
}
```

2.2. Insertion Sort

```
public class AlgoritmoInsertionSort {

   public static void main(String[] args) {

      int tamanhoVetor = 500000;
      int vetor[] = new int[tamanhoVetor];
      int intervaloMin = 1; // VALOR MINIMO DOS NUMEROS ALEATORIOS
      int intervaloMax = 500000; // VALOR MAXIMO DOS NUMEROS ALEATORIOS
```

```
        int ponteiro = 1; // INICIALIZA MARCADOR DE POSICoES
        int indice = 0;

        Random random = new Random(12345);

        for (int i = 0; i < tamanhoVetor; i++) {
            // GERA UM NUMERO ALEATORIO ENTRE INTERVALO MIN E MAX (inclusive)
            vetor[i] = random.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
        }
        // TEMPO DE EXECUCAO INICIA AQUI
        long startTime = System.nanoTime();
        while (ponteiro < tamanhoVetor) {
            int aux = vetor[ponteiro];  // ARMAZENA VALOR ATUAL EM AXILIAR
            indice = ponteiro - 1;  // INICIALIZA O INDICE COM O VALOR DO MARCADOR DE POSICOES COM DECREMENT

            while ((indice >= 0) && (vetor[indice] > aux)) {
                vetor[indice + 1] = vetor[indice];  // MOVE ELEMENTOS MAIORES PARA A DIREITA
                indice = indice - 1;   // DECREMENTA O INDICE
            }
            vetor[indice + 1] = aux;  // INSERE O VALOR AUX NA POSICAO CORRETA
            ponteiro = ponteiro + 1;
        }
        long endTime = System.nanoTime();
        long duration = (endTime - startTime);  // TEMPO DE EXECUCAO EM NANOSSEGUNDOS
        double milliseconds = (double) duration / 1_000_000.0; // CONVERTE EM MILISEGUNDOS
        System.out.println("Tempo de execucao: " + milliseconds + " milisegundos.");

        for (int i = 0; i < tamanhoVetor; i++) {
            System.out.print(vetor[i] + " ");   // Imprime o vetor ordenado
        }
    }
}
```

2.3. Merge Sort

```
public class AlgoritmoMergeSort {
    public static void main(String[] args) {
        int tamanhoVetor = 500000;
        int vetor[] = new int[tamanhoVetor];
        int intervaloMin = 1; // VALOR MINIMO DOS NUMEROS ALEATORIOS
        int intervaloMax = 500000; // VALOR MAXIMO DOS NUMEROS ALEATORIOS

        Random random = new Random(12345);

        for (int i = 0; i < tamanhoVetor; i++) {
            // GERA UM NUMERO ALEATORIO ENTRE INTERVALO MIN E MAX (inclusive)
            vetor[i] = random.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
        }

        // TEMPO DE EXECUCAO INICIA AQUI
        long startTime = System.nanoTime();

        // CHAMA A FUNCAO MERGESORT PARA ORDENAR O VETOR
        mergeSort(vetor, 0, tamanhoVetor - 1);

        long endTime = System.nanoTime();
        long duration = (endTime - startTime);  // TEMPO DE EXECUCAO EM NANOSSEGUNDOS
        double milliseconds = (double) duration / 1_000_000.0; // CONVERTE EM MILISEGUNDOS
        System.out.println("Tempo de execucao: " + milliseconds + " milisegundos.");
```

```
        // IMPRIME O VETOR ORDENADO
        for (int i = 0; i < tamanhoVetor; i++) {
            System.out.print(vetor[i] + " ");
        }
    }

    public static void mergeSort(int[] vetor, int inicio, int fim) {
        if (inicio < fim) {
            int meio = (inicio + fim) / 2;
            // CHAMA O MERGESORT RECURSIVAMENTE PARA A METADE ESQUERDA E DIREITA DO VETOR
            mergeSort(vetor, inicio, meio);
            mergeSort(vetor, meio + 1, fim);

            // CHAMA A FUNCAO MERGE PARA COMBINAR E ORDENAR AS DUAS METADES
            merge(vetor, inicio, meio, fim);
        }
    }

    public static void merge(int[] vetor, int inicio, int meio, int fim) {
        int indiceEsquerdo = inicio, indiceDireito = meio + 1, indiceAuxiliar = 0;
        // CRIA UM VETOR AUXILIAR PARA ARMAZENAR OS VALORES ORDENADOS
        int[] vetorAuxiliar = new int[fim - inicio + 1];

        while (indiceEsquerdo <= meio && indiceDireito <= fim) {
            if (vetor[indiceEsquerdo] <= vetor[indiceDireito]) {
                vetorAuxiliar[indiceAuxiliar] = vetor[indiceEsquerdo];
                indiceEsquerdo++;
            } else {
                vetorAuxiliar[indiceAuxiliar] = vetor[indiceDireito];
                indiceDireito++;
            }
            indiceAuxiliar++;
        }

        // COPIA OS ELEMENTOS RESTANTES DA PRIMEIRA METADE (SE HOUVER)
        while (indiceEsquerdo <= meio) {
            vetorAuxiliar[indiceAuxiliar] = vetor[indiceEsquerdo];
            indiceAuxiliar++;
            indiceEsquerdo++;
        }

        // COPIA OS ELEMENTOS RESTANTES DA SEGUNDA METADE (SE HOUVER)
        while (indiceDireito <= fim) {
            vetorAuxiliar[indiceAuxiliar] = vetor[indiceDireito];
            indiceAuxiliar++;
            indiceDireito++;
        }

        // MOVE OS ELEMENTOS ORDENADOS DE VOLTA PARA O VETOR ORIGINAL
        for (indiceAuxiliar = inicio; indiceAuxiliar <= fim; indiceAuxiliar++) {
            vetor[indiceAuxiliar] = vetorAuxiliar[indiceAuxiliar - inicio];
        }
    }
}
```

2.4. Quick Sort

```
public class AlgoritmoQuickSort {
   public static void main(String[] args) {
      int tamanhoVetor = 500000;
      int vetor[] = new int[tamanhoVetor];
      int intervaloMin = 1; // VALOR MINIMO DOS NUMEROS ALEATORIOS
      int intervaloMax = 500000; // VALOR MAXIMO DOS NUMEROS ALEATORIOS

      Random random = new Random(12345);

      for (int i = 0; i < tamanhoVetor; i++) {
       //GERA UM NUMERO ALEATORIO ENTRE INTERVALO MIN E MAX (inclusive)
         vetor[i] = random.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
      }
      // TEMPO DE EXECUCAO INICIA AQUI
      long startTime = System.nanoTime();

      // CHAMA A FUNCAO QUICKSORT PARA ORDENAR O VETOR
      quickSort(vetor, 0, vetor.length - 1);
      long endTime = System.nanoTime();
      long duration = (endTime - startTime);  // TEMPO DE EXECUCAO EM NANOSSEGUNDOS
      double milliseconds = (double) duration / 1_000_000.0; // CONVERTE EM MILISEGUNDOS
      System.out.println("Tempo de execucao: " + milliseconds + " milisegundos.");

      // IMPRIME O VETOR ORDENADO
      for (int i = 0; i < tamanhoVetor; i++) {
         System.out.print(vetor[i] + " ");
      }
   }

   private static void quickSort(int[] vetor, int inicio, int fim) {
      if (inicio < fim) {
         // SEPARA O VETOR EM DUAS PARTES E REORNA A POSICAO DO PIVO
         int posicaoPivo = separar(vetor, inicio, fim);
         // ORDENA A PRIMEIRA PARTE DO VETOR (ANTES DO PIVO)
         quickSort(vetor, inicio, posicaoPivo - 1);
         // ORDENA A SEGUNDA PARTE DO VETOR (DEPOIS DO PIVO)
         quickSort(vetor, posicaoPivo + 1, fim);
      }
   }

   // METODO SEPARAR: DIVIDE O VETOR EM DUAS PARTES COM BASE NO VALOR DO PIVO
   private static int separar(int[] vetor, int inicio, int fim) {
      int pivo = vetor[inicio];
      int indicePercorrer = inicio + 1;
      while (indicePercorrer <= fim) {
         if (vetor[indicePercorrer] <= pivo) {
            indicePercorrer++;
         } else if (pivo < vetor[fim]) {
            fim--;
         } else {
            int troca = vetor[indicePercorrer];
            vetor[indicePercorrer] = vetor[fim];
            vetor[fim] = troca;
            indicePercorrer++;
            fim--;
         }
      }
```

```
        // COLOCA PIVO NA POSICAO CORRETA
        vetor[inicio] = vetor[fim];
        vetor[fim] = pivo;
        return fim;
    }
}
```

2.5. Bubble Sort

```
public class AlgoritmoBubbleSort {
    public static void main(String[] args) {

        int tamanhoVetor = 400000;
        int vetor[] = new int[tamanhoVetor];
        int intervaloMin = 1; // VALOR MINIMO DOS NUMEROS ALEATORIOS
        int intervaloMax = 400000; // VALOR MAXIMO DOS NUMEROS ALEATORIOS

        Random random = new Random(12345);

        for (int i = 0; i < tamanhoVetor; i++) {
            //GERA UM NUMERO ALEATORIO ENTRE INTERVALO MIN E MAX (inclusive)
            vetor[i] = random.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
        }

        int indice, fim, aux;
        // TEMPO DE EXECUCAO INICIA AQUI
        long startTime = System.nanoTime();
        for (fim = tamanhoVetor - 1; fim > 0; --fim) {
            for (indice = 0; indice < fim; ++indice) {
                if (vetor[indice] > vetor[indice + 1]) {
                    aux = vetor[indice];
                    vetor[indice] = vetor[indice + 1];
                    vetor[indice + 1] = aux;
                }
            }
        }

        long endTime = System.nanoTime();
        long duration = (endTime - startTime);  // TEMPO DE EXECUCAO EM NANOSSEGUNDOS
        double milliseconds = (double) duration / 1_000_000.0; // CONVERTE EM MILISEGUNDOS
        System.out.println("Tempo de execucao: " + milliseconds + " milisegundos.");

        for (int i = 0; i < tamanhoVetor; i++) {
            System.out.print(vetor[i] + " ");   // IMPRIME O VETOR ORDENADO
        }
    }
}
```

3. Metodologia

Os testes iniciam com um vetor de 100 elementos, é avaliado o tempo de execução dos algoritmos nos seguintes cenários:
a) Vetor pré-ordenado na forma crescente.
b) Vetor pré-ordenado na forma decrescente.
c) Vetor aleatório organizado em ordem crescente.
d) Vetor aleatório organizado em ordem decrescente.

Esse procedimento é repetido em vetores de tamanhos 1.000, 100.000, 200.000, 300.000, 400.000 e 500.000. Com base nos resultados, é possível analisar as vantagens e desvantagens de um algoritmo em comparação ao outro.
Para os resultados dos testes realizados com os algoritmos de ordenação foi utilizado o Java como linguagem de programação e a IDE Apache NetBeans. Em relação ao hardware, os testes foram realizados em um computador com as seguintes especificações:

3.1. Configurações do Hardware

Fabricante: Dell
Modelo: Inspiron 15
Processador: AMD Ryzen 5
Frequência: 2.10 GHz
Memória RAM: 8,00 GB
Sistema Operacional: Windows 11 (64 bits)
Seed fixada nos vetores aleatórios: (new Random(12345))

3.2. Teste de Vetores Aleatórios

Os resultados dos testes em vetores aleatórios estão apresentados nas Tabelas 1 e 2, acompanhados pelos Gráficos 1 e 2. Os vetores utilizados possuem tamanhos de v[100], v[1.000], v[100.000], v[200.000], v[300.000], v[400.000] e v[500.000] elementos.

Tabela 1. Desempenho dos algoritmos com elementos aleatórios ordenados na forma crescente

ALEATÓRIO PARA ORDEM CRESCENTE – (TEMPO EM MILISEGUNDOS)					
TAMANHO	BUBBLE SORT	INSERTION SORT	MERGE SORT	SELECTION SORT	QUICK SORT
v[100]	0,17	0,09	0,88	0,10	0,03
v[1000]	7,19	5,44	1,15	4,37	0,42
v[100000]	15.777,56	742,61	25,11	4.343,95	16,65
v[200000]	68.392,67	3.137,12	53,50	17.697,51	24,25
v[300000]	154.388,03	7.066,35	74,50	39.900,56	33,75
v[400000]	275.729,03	12.759,19	87,56	70.893,64	46,69
v[500000]	427.987,77	19.172,45	99,34	109.171,04	60,93

Gráfico 1. Desempenho dos algoritmos com elementos aleatórios ordenados na forma crescente

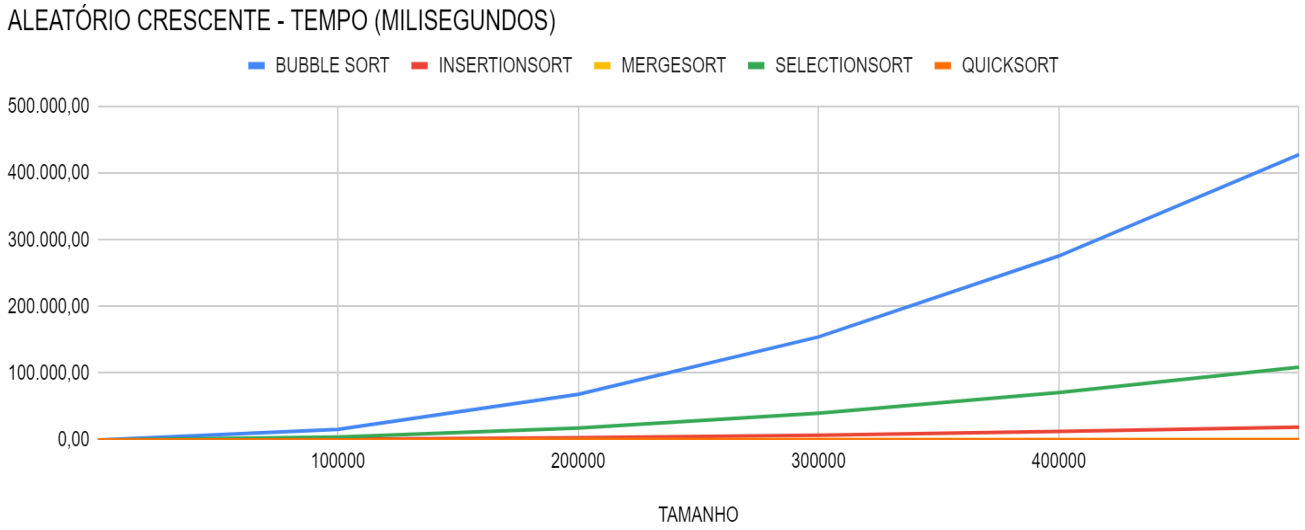
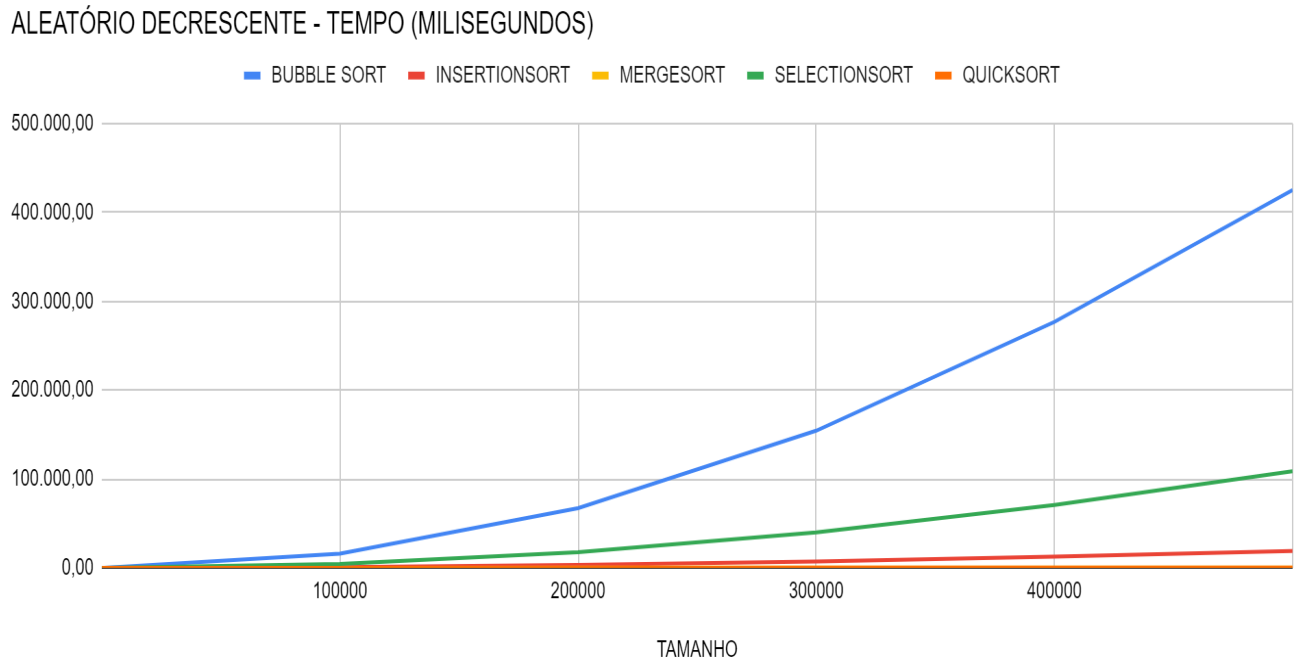


Tabela 2. Desempenho dos algoritmos com elementos aleatórios ordenados na forma decrescente

ALEATÓRIO PARA ORDEM DECRESCENTE – (TEMPO EM MILISEGUNDOS)					
TAMANHO	BUBBLE SORT	INSERTION SORT	MERGE SORT	SELECTION SORT	QUICK SORT
v[100]	0,17	0,08	0,08	0,11	0,03
v[1000]	7,70	5,05	1,21	4,39	0,44
v[100000]	15.790,54	745,31	25,22	4.344,86	14,78
v[200000]	66.947,58	3.185,51	52,50	17.621,71	26,33
v[300000]	154.166,35	7.178,54	75,26	39.678,58	38,55
v[400000]	276.541,24	12.602,32	86,23	70.516,14	44,73
v[500000]	424.852,79	18.926,45	98,92	108.630,73	58,43

Gráfico 2. Desempenho dos algoritmos com elementos aleatórios ordenados para forma decrescente



3.3. Teste de Vetores Pré-Ordenados

Nas Tabelas 3 e 4, assim como nas tabelas anteriores, são apresentados os resultados dos testes realizados com os diferentes tamanhos de vetores para os cinco algoritmos, em listas pré-ordenadas de forma crescente e decrescente.

Tabela 3. Desempenho dos algoritmos com elementos pré-ordenados em ordem crescente

ORDEM CRESCENTE – (TEMPO EM MILISEGUNDOS)					
TAMANHO	BUBBLE SORT	INSERTION SORT	MERGE SORT	SELECTION SORT	QUICK SORT
v[100]	0,10	0,01	0,08	0,14	0,14
v[1000]	3,04	0,04	0,86	2,74	3,89
v[100000]	1.118,45	0,43	25,69	955,74	23,86
v[200000]	4.482,50	0,84	37,12	3.526,80	43,85
v[300000]	10.136,51	1,23	54,01	8.400,94	53,49
v[400000]	17.978,25	1,67	66,42	14.018,88	61,87
v[500000]	387.014.537	2,98	91,67	31.278,49	71,45

Gráfico 3. Desempenho dos algoritmos com elementos pré-ordenados em ordem crescente

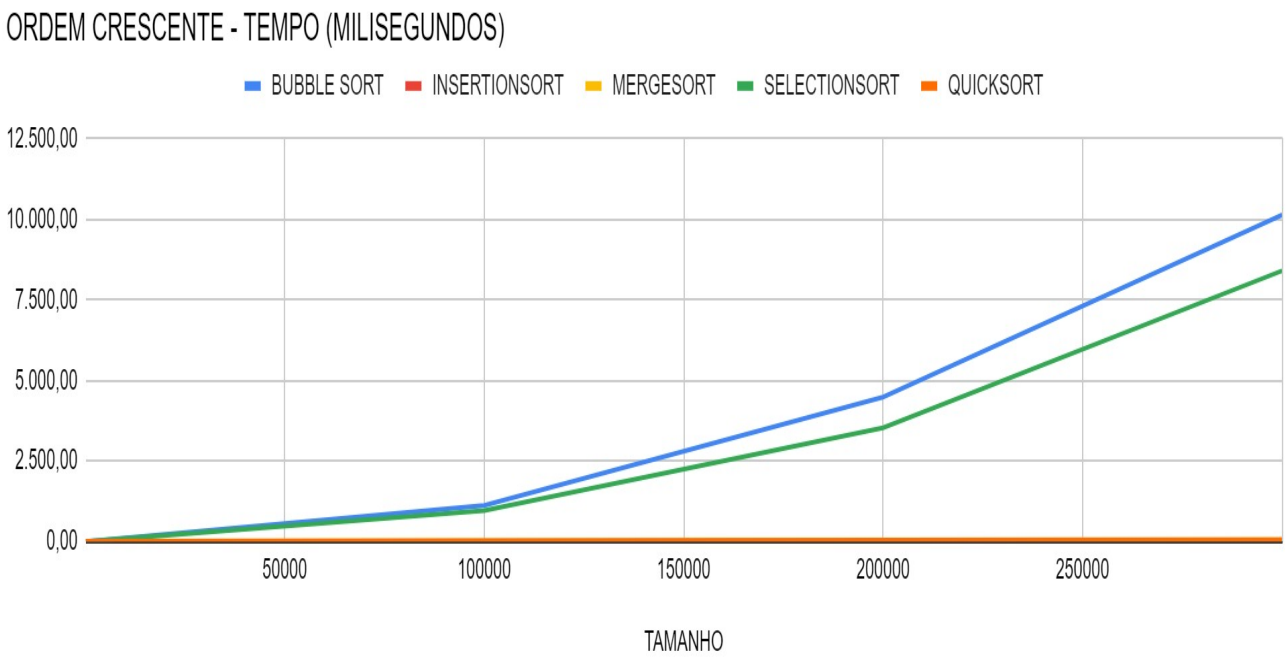
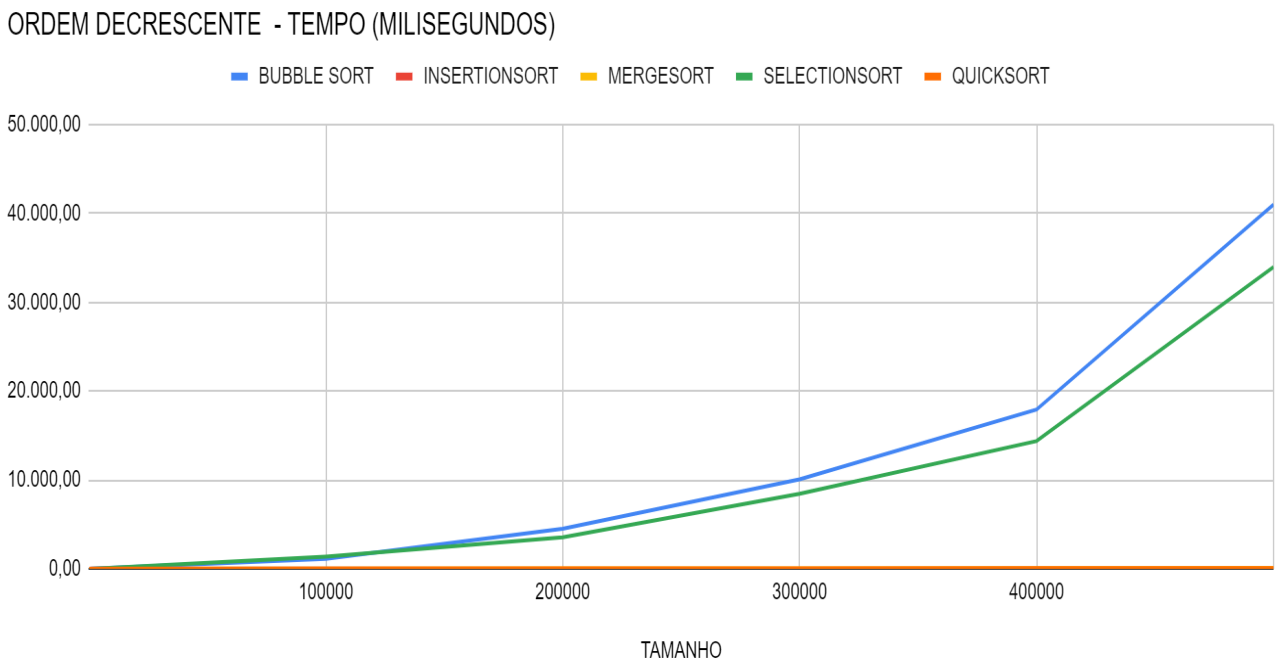


Tabela 4. Desempenho dos algoritmos com elementos pré-ordenados em ordem decrescente

ORDEM DECRESCENTE – TEMPO (MILISEGUNDOS)					
TAMANHO	BUBBLE SORT	INSERTION SORT	MERGE SORT	SELECTION SORT	QUICK SORT
v[100]	0,10	0,01	0,07	0,11	0,13
v[1000]	2,99	0,05	0,84	2,73	3,68
v[100000]	1.111,56	5,41	29,63	1.376,45	23,61
v[200000]	4.494,125	0,8184	39,35	3.549,95	44,93
v[300000]	10.069,11	1,2269	57,66	8.441,32	51,74
v[400000]	17.917,62	1,75	60,36	14.363,65	62,95
v[500000]	40.968,34	2,04	93,08	33.945,94	70,45

Gráfico 4. Desempenho dos algoritmos com elementos pré-ordenados em ordem decrescente



4. Resultado e Discussão

Com base na análise e na visualização dos gráficos, observamos que cada algoritmo de ordenação apresenta características e particularidades que podem afetar sua eficiência e complexidade, dependendo do tamanho do vetor, da disposição dos elementos (melhor e pior caso) e da organização prévia ou aleatória dos dados.

O algoritmo Bubble Sort, apesar de ser de fácil implementação, não produziu resultados satisfatórios. Ele possui uma complexidade média e pior caso de $O(n^2)$, o que significa que, em média, é necessário percorrer o vetor de N elementos N vezes (N-K vezes) para realizar a ordenação ($N*N = N^2$). Isso torna o Bubble Sort menos eficiente, especialmente quando o número de elementos é grande. O Selection Sort também apresenta uma complexidade $O(n^2)$, o que resulta em alto consumo de recursos de processamento, levando a lentidão em máquinas com recursos limitados.

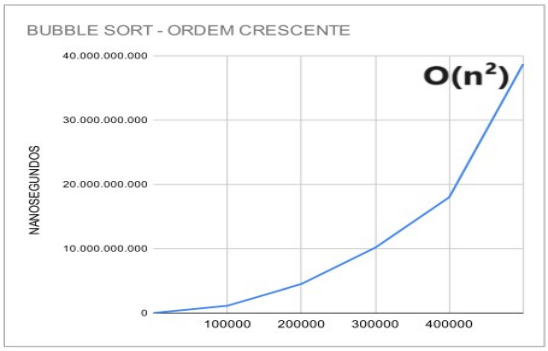
O consumo significativo de recursos de processamento é definido como complexidade espacial, e, nesse contexto, tanto o Bubble Sort quanto o Selection Sort têm complexidade O(1), o que significa que não requerem espaço adicional significativo além do usado para armazenar a entrada, devido ao fato de serem algoritmos de ordenação in-place.

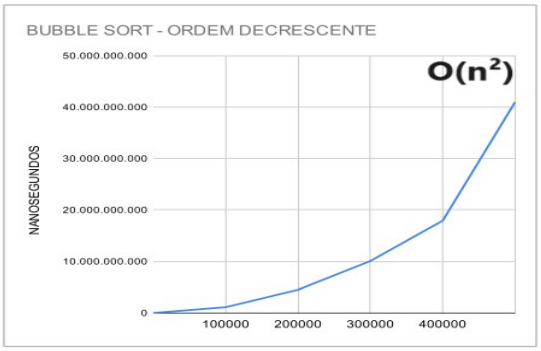
O Insertion Sort, apesar de possuir complexidade média e pior caso de $O(n^2)$, demonstrou ser mais eficiente em estruturas lineares com poucos elementos, especialmente quando os dados de entrada já estão parcialmente ordenados. Nesse caso, pode atingir uma complexidade de tempo de O(n) no melhor cenário. Complexidade espacial: O(1).

Quanto aos algoritmos Merge Sort e Quick Sort, ambos se destacam ao particionar o vetor em subvetores menores, resultando em maior eficiência. O Merge Sort possui uma complexidade de tempo de O(n log n) para todos os casos, enquanto o Quick Sort, embora eficiente em média, pode degradar para $O(n^2)$ no pior caso, quando os dados de entrada estão completamente ordenados ou

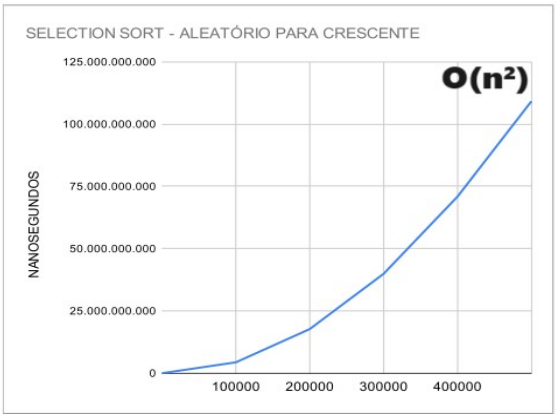
inversamente ordenados.
Ambos os algoritmos apresentam complexidade espacial de O(n), e não são algoritmos in-place, requerendo espaço adicional para armazenar a lista auxiliar usada durante a ordenação.

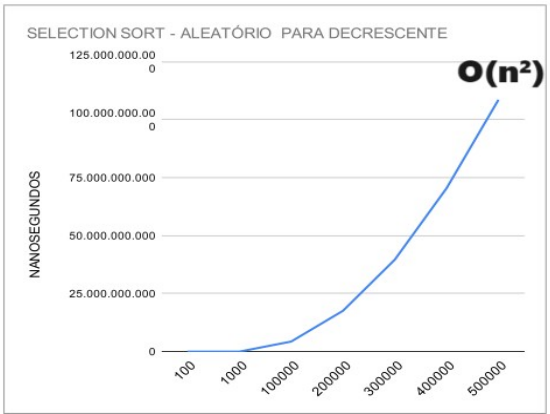
Complexidade Bubble Sort:

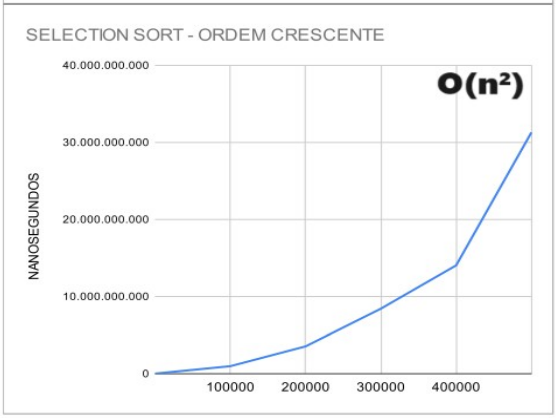


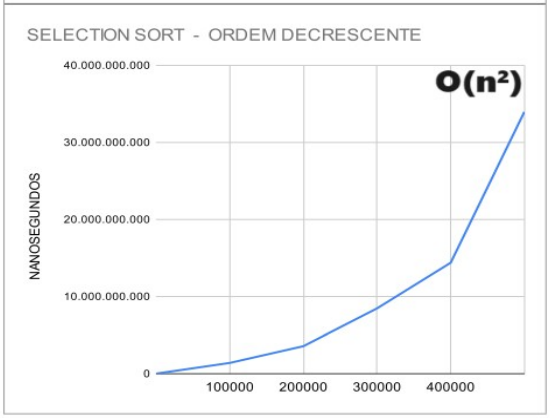


Complexidade Selection Sort:

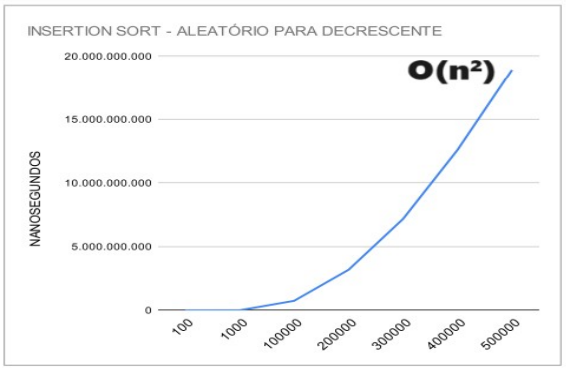


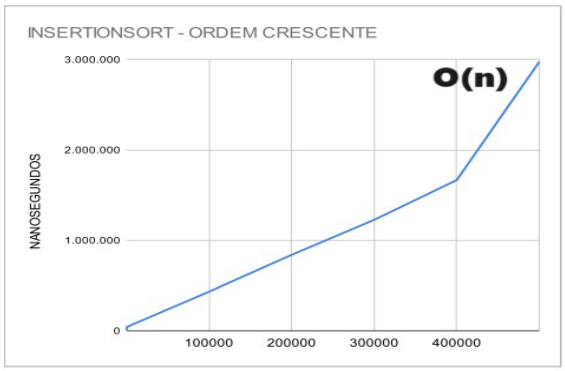


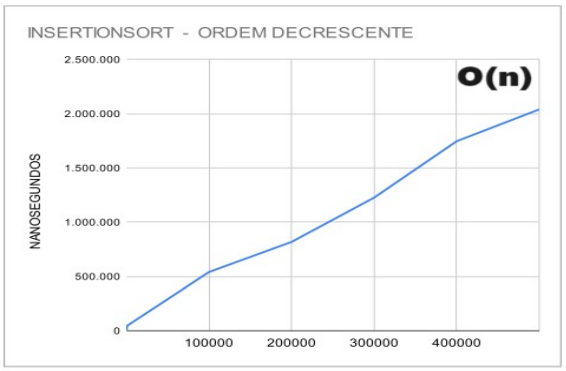




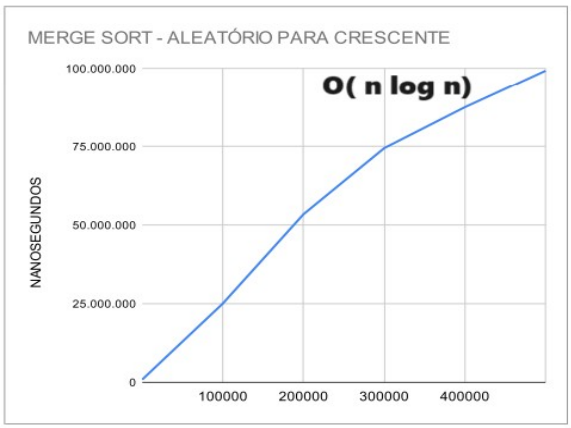
Complexidade Insertion Sort:

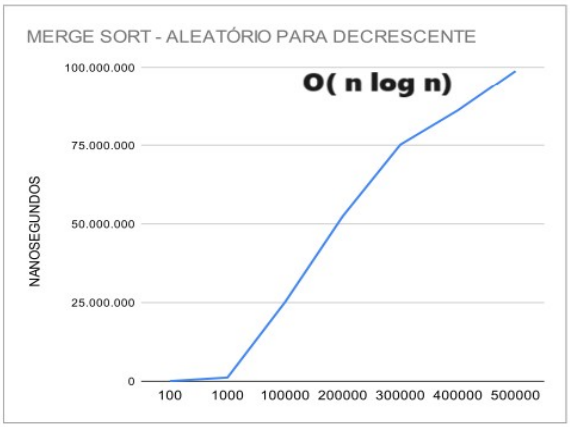


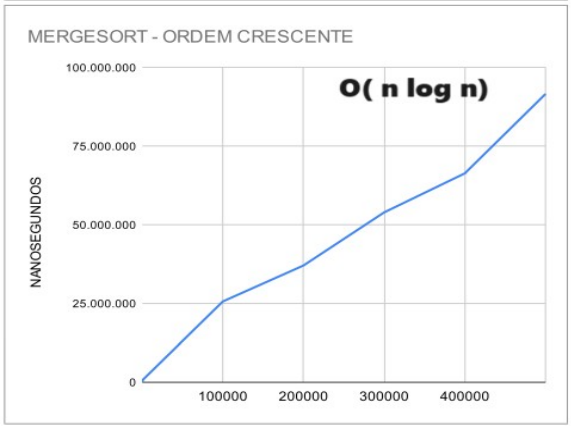


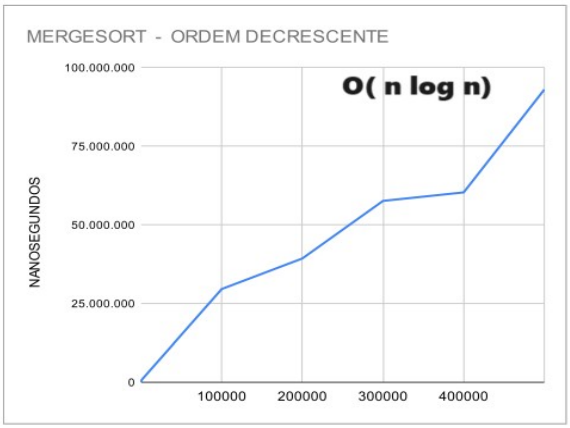


Complexidade Merge Sort:







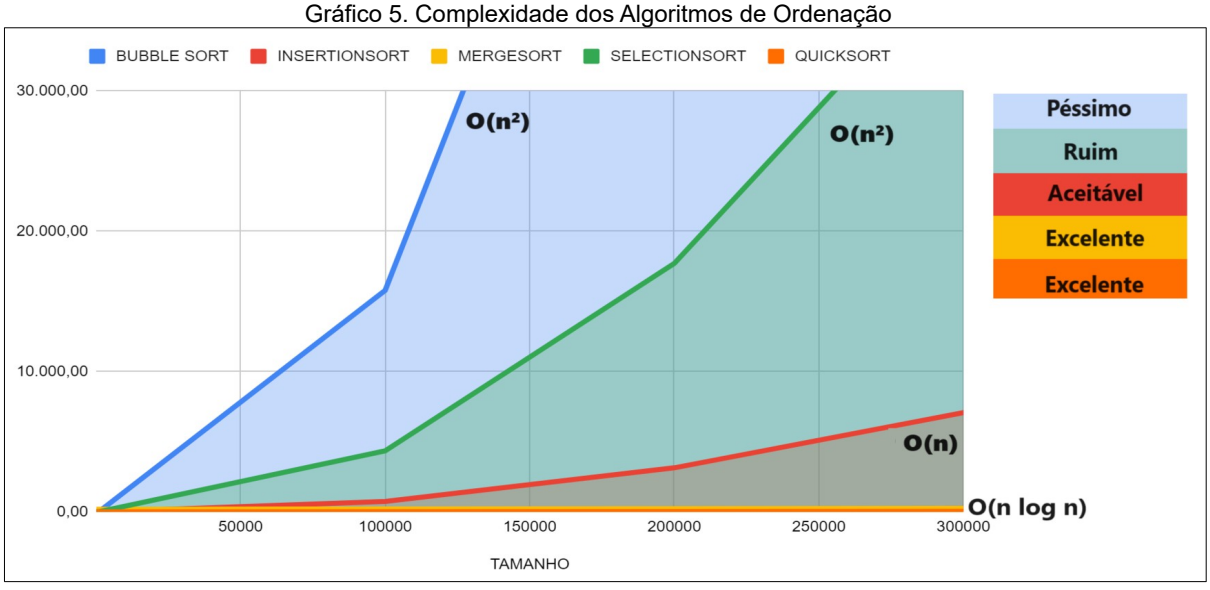


Complexidade Quick Sort:

Tabela 5. Complexidade dos Algoritmos de Ordenação

ALGORITMO	COMPLEXIDADE			ESPAÇO
	MELHOR	MÉDIO	PIOR	
BUBBLE SORT	$O(n^2)$			O(1)
SELECTIONSORT	$O(n^2)$			O(1)
INSERTIONSORT	O(n)		$O(n^2)$	O(1)
MERGESORT	O(n log n)			O(n)
QUICKSORT	O(n log n)		$O(n^2)$	O(n)

Gráfico 5. Complexidade dos Algoritmos de Ordenação



5. Considerações Finais

5.1. Aplicações Reais

Ao considerar as aplicações reais dos algoritmos de ordenação e como eles se comparam entre si, é fundamental levar em conta suas capacidades e limitações em diversos cenários práticos. Cada algoritmo de ordenação apresenta características específicas que o tornam mais adequado para determinados contextos.

- **Ordenação em tempo real:** Quando a necessidade de classificação ocorre em tempo real, o algoritmo Quick Sort emerge como uma escolha sólida. Sua complexidade média de tempo de O(n log n) faz com que seja uma opção eficiente em muitos casos práticos.

- **Ordenação de grandes volumes de dados:** Para situações em que é preciso classificar conjuntos de dados extensos que não podem ser totalmente alocados na memória principal, o Merge Sort, um algoritmo externo, se destaca. Minimizando o acesso ao disco, o Merge Sort se torna uma opção adequada para lidar com grandes volumes de dados de maneira eficiente.

- **Ordenação estável:** A estabilidade da ordenação pode ser crucial em certos aplicativos nos quais é fundamental preservar a ordem relativa dos elementos com chaves iguais. Nesse contexto, o Merge Sort se destaca como um algoritmo estável, assegurando que a ordem original dos elementos seja mantida. Isso é particularmente valioso em cenários nos quais a ordem relativa dos itens é relevante.

5.2 Limitações dos algoritmos de Ordenação

Além das aplicações vantajosas, é fundamental reconhecer as limitações de cada algoritmo de ordenação, visto que sua eficácia pode variar dependendo do cenário e das necessidades específicas.

- **Espaço de Memória:** Algoritmos como Merge Sort e Quick Sort consomem mais memória, pois requerem espaço adicional para armazenar listas auxiliares. Portanto, você deve considerar as restrições de memória do sistema.

- **Desempenho em casos específicos:** Alguns algoritmos, como o Bubble Sort e o Selection Sort, apresentam desempenho insatisfatório em praticamente todos os cenários. Portanto, a menos que haja circunstâncias excepcionais, é aconselhável evitar seu uso na maioria das situações.

- **Complexidade de implementação:** Implementar corretamente certos algoritmos, como o Quick Sort, pode ser uma tarefa complexa. Essa complexidade de implementação aumenta o risco de erros, exigindo uma atenção especial durante o desenvolvimento.

- **Tempo de execução de pior caso:** O Quick Sort, embora eficiente em média, pode ter um tempo de execução no pior caso de $O(n^2)$ quando os dados de entrada estão pré-ordenados ou inversamente ordenados. Portanto, é importante exercer cautela ao utilizá-lo em cenários nos quais o pior caso é uma ocorrência comum, ou explorar estratégias de otimização para mitigar esse problema.

Essas considerações práticas são fundamentais para a seleção do algoritmo de ordenação mais apropriado para um determinado caso, permitindo um equilíbrio entre desempenho e limitações específicas.

6. Referências

COELHO, Hebert; FÉLIX, Nádia. “Métodos de Ordenação: Selection, Insertion, Bubble, Merge (Sort)”, 2009. Disponível em: https://ww2.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf Acesso em: 28/10/2023.

HONORATO, B. “Algoritmos de ordenação: análise e comparação”, 2013. Disponível em: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261. Acesso em: 28/10/2023.

VIANA, Daniel. “Conheça os principais algoritmos de ordenação”, 2016. Disponível em: https://periodicos.ufersa.edu.br/ecop/article/view/7082/6540. Acesso em: 28/10/2023.

SOUZA, Jackson É. G.; RICARTE, João V. G.; LIMA, Náthalee C. A. “Algoritmos de Ordenação: Um Estudo Comparativo”, 2017. Disponível em: https://periodicos.ufersa.edu.br/ecop/article/view/7082/6540. Acesso em: 01/11/2023

SILVA, Arthur V. Romualdo. “Análise de desempenho e complexidade dos Algoritmos de ordenação”, 2022.Disponível em: https://medium.com/@romualdo.v/an%C3%A1lise-de-desempenho-e-complexidade-dos-algoritmos-de-ordena%C3%A7%C3%A3o-f47449e93b33. Acesso em: 01/11/2023.