Trabalho Prático 1
Ordenador Universal

Lucas Pimenta Braga - 2023034552

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

lucaspimentabraga@gmail.com

1. Introdução

O trabalho tem como objetivo implementar e analisar o Tipo Abstrato de Dados (TAD) denominado Ordenador Universal, que busca selecionar automaticamente o algoritmo de ordenação mais eficiente entre os métodos de inserção e quicksort, considerando as características específicas do vetor de dados a ser ordenado. Essa seleção adaptativa baseia-se em critérios como o número de quebras (ou inversões locais) no vetor e o tamanho das partições, com o intuito de otimizar o desempenho do processo de ordenação.

A motivação do trabalho reside na necessidade de oferecer uma solução flexível que aprenda e se adapte empiricamente às diferentes configurações dos dados, melhorando a eficiência da ordenação em diversos cenários. Para isso, são aplicadas técnicas de calibração de parâmetros, como os limiares de quebras e tamanho mínimo de partição, utilizando métricas de custo que combinam o número de comparações, movimentações e chamadas de função, ponderadas por coeficientes determinados experimentalmente.

A implementação do TAD é realizada na linguagem C++, respeitando restrições quanto ao uso de bibliotecas padrão, o que assegura controle completo sobre as estruturas de dados e algoritmos empregados. Além da implementação, o trabalho contempla uma análise teórica da complexidade dos algoritmos e uma análise experimental para validar a eficácia da abordagem adaptativa.

2. Método

A implementação consiste no desenvolvimento de um **Tipo Abstrato de Dados (TAD) OrdenadorUniversal**, que realiza a ordenação adaptativa de vetores de inteiros, escolhendo entre os algoritmos **insertion sort** e **quicksort** com mediana de três, com base em características do vetor. O objetivo é otimizar o desempenho da ordenação, usando parâmetros ajustáveis e calibração empírica.

2.1. Estrutura da Classe OrdenadorUniversal

A classe OrdenadorUniversal encapsula a lógica do algoritmo de ordenação adaptativa e contém os seguintes componentes:

Atributos principais

- **int* v**: Ponteiro para o vetor de inteiros a ser ordenado.
- **int n**: Tamanho do vetor.
- **int cmp, move, calls**: Contadores que armazenam o número de comparações, movimentações e chamadas de função realizadas durante a ordenação.
- **double a, b, c**: Coeficientes para cálculo do custo ponderado (comparações, movimentações e chamadas de função).
- **long int seed**: Semente utilizada para garantir a reprodutibilidade nos experimentos de embaralhamento do vetor.

Construtor

- **OrdenadorUniversal(int* vetor, int tamanho, double a_, double b_, double c_, long int seed_)**: Inicializa a classe com o vetor a ser ordenado, seu tamanho, os coeficientes para ponderação do custo e a semente de aleatoriedade para os embaralhamentos.

2.2. Métodos Principais da Classe

Funções de Ordenação

- **void insercao(int l, int r)**: Implementa o algoritmo de **insertion sort**, que ordena o subvetor definido entre os índices l e r. Utilizado quando a escolha adaptativa ou o limite de partição do quicksort recomenda a inserção.
- **int mediana(int a, int b, int c)**: Retorna o valor mediano entre três números, utilizado para a escolha do pivô no algoritmo **quicksort**, com a técnica de mediana de três para evitar o pior caso de eficiência.
- **void particao(int l, int r, int* i, int* j)**: Realiza a **partição** do vetor no algoritmo **quicksort**, utilizando a mediana de três como pivô. Atualiza os índices i e j que delimitam as subpartições.
- **void quicksort(int l, int r, int minTamParticao)**: Ordenação recursiva utilizando **quicksort adaptativo**. Para partições menores que o limite minTamParticao, a função utiliza **insertion sort**, evitando a sobrecarga de recursão em subvetores pequenos.

Funções de Cálculo de Custo

- **double calculaCusto() const**: Calcula o custo total da ordenação combinando o número de comparações, movimentações e chamadas de função. A fórmula utilizada é:

$$f = a \times cmp + b \times move + c \times calls$$

Funções de Ajuste de Limiares

- **int determinaLimiarParticao(double limiarCusto)**: Realiza uma **busca iterativa** para determinar o limiar ideal de **tamanho mínimo de partição** para o **quicksort**. A faixa de possíveis limiares é refinada até que a diferença de custo entre os extremos seja menor que o limiar de convergência.

- **int determinaLimiarQuebras(int limiarParticao, double limiarCusto)**: Determina o **limiar de número de quebras** (inversões locais) para o vetor. Baseado no limiar de partição encontrado, a função avalia o custo de ordenação para diferentes números de quebras e ajusta o valor até encontrar o limiar ideal que minimiza o custo de ordenação.

Funções Auxiliares

- **int contaQuebras()**: Conta o número de **quebras** (ou inversões locais) no vetor. Esse valor é utilizado para determinar se o algoritmo de **insertion sort** ou **quicksort** será mais eficiente.
- **void calculaNovaFaixa(int& minMPS, int& maxMPS, int& passoMPS, int limParticao, int numMPS, const int mpsVals[])**: Ajusta a faixa de valores para o limiar de partição com base no melhor índice encontrado na iteração atual, permitindo um refinamento sucessivo dos limiares de partição.

3. Análise de Complexidade

A análise de complexidade de tempo e espaço dos procedimentos implementados é essencial para entender o comportamento do **OrdenadorUniversal** em diferentes cenários e avaliar sua eficiência. A seguir, será feita a análise teórica, utilizando a notação assintótica, para cada um dos principais algoritmos e funções.

3.1. Insertion Sort (Função insercao)

O **Insertion Sort** é utilizado para ordenar o vetor quando o número de quebras é baixo ou o vetor está parcialmente ordenado.

- **Tempo**: A complexidade é **O(n²)** no pior caso, pois o algoritmo realiza **n** comparações e movimentações para cada elemento do vetor.
- **Espaço**: O algoritmo é **in-place**, ou seja, não requer memória adicional além do vetor original. Portanto, a complexidade espacial é **O(1)**.

3.2. Quicksort (Função quicksort)

O **quicksort** é um algoritmo eficiente que utiliza a técnica de **dividir e conquistar** para ordenar o vetor. A escolha do pivô é feita utilizando a **mediana de três** para melhorar o desempenho e reduzir a probabilidade de um pior caso.

- **Tempo**: No **melhor caso** e **caso médio**, a complexidade é **O(n log n)**, onde **n** é o número de elementos do vetor. No **pior caso**, a complexidade pode ser **O(n²)**, caso o particionamento seja desequilibrado, mas isso é mitigado pela mediana de três.
- **Espaço**: A complexidade espacial é **O(log n)** no melhor caso devido à recursão e **O(n)** no pior caso, dependendo do particionamento.

3.3. Função contaQuebras

A função **contaQuebras** percorre o vetor e conta quantas vezes um elemento é menor que seu antecessor. Isso é utilizado para decidir qual algoritmo aplicar (insertion sort ou quicksort).

- **Tempo**: A complexidade é **O(n)**, pois a função realiza uma única passagem pelo vetor.
- **Espaço**: A complexidade espacial é **O(1)**, pois a função utiliza apenas uma variável para contar as quebras.

3.4. Função determinaLimiarParticao

Este método realiza uma busca iterativa para encontrar o limiar ideal de partição, o qual minimiza o custo de ordenação.

- **Tempo**: A complexidade é **O(n)**, pois a função percorre o vetor uma vez para calcular o custo em cada iteração. O número máximo de iterações é **10** (valor fixo), o que mantém a complexidade total em **O(n)**.
- **Espaço**: A complexidade espacial é **O(n)**, devido ao uso de arrays auxiliares para armazenar os custos e valores temporários.

3.5. Função determinaLimiarQuebras

Esta função busca adaptativamente o limiar de número de quebras, ajustando dinamicamente o valor até encontrar o número ótimo de quebras para a ordenação.

- **Tempo**: A complexidade é **O(n² log n)**, pois a função realiza uma busca iterativa sobre os valores possíveis de quebras, e a cada iteração executa o **quicksort** e o **insertion sort**.
- **Espaço**: A complexidade espacial é **O(n)**, pois a função utiliza vetores auxiliares para armazenar os resultados das iterações.

3.6. Função shuffleVector

A função **shuffleVector** realiza um número controlado de trocas aleatórias entre os elementos do vetor, o que é necessário para gerar um número específico de quebras.

- **Tempo**: A complexidade é **O(num_shuffle)**, onde **num_shuffle** é o número de trocas a serem feitas.
- **Espaço**: A complexidade espacial é **O(1)**, pois a função modifica o vetor diretamente sem utilizar memória extra.

3.7. Resumo da Complexidade

Tempo:

- **Insertion Sort**: **O(n²)**
- **Quicksort**: **O(n log n)** no melhor e caso médio, **O(n²)** no pior caso (mitigado pela mediana de três)
- **Função contaQuebras**: **O(n)**
- **Função determinaLimiarParticao**: **O(n)** com no máximo 10 iterações
- **Função determinaLimiarQuebras**: **O(n² log n)**
- **Função shuffleVector**: **O(num_shuffle)**

Espaço:

- **Insertion Sort**: **O(1)**
- **Quicksort**: **O(log n)** no melhor caso, **O(n)** no pior caso
- **Função contaQuebras**: **O(1)**
- **Função determinaLimiarParticao**: **O(n)**
- **Função determinaLimiarQuebras**: **O(n)**
- **Função shuffleVector**: **O(1)**

4. Estratégias de Robustez

O código implementado no OrdenadorUniversal adota diversas estratégias de robustez para garantir a integridade das operações, além de tolerância a falhas e eficiência na execução dos algoritmos de ordenação. Abaixo, descrevo as principais estratégias de robustez e suas justificativas.

4.1. Inicialização Adequada e Gerenciamento de Memória

- **Inicialização do vetor e parâmetros**: No construtor da classe OrdenadorUniversal, os parâmetros como o vetor, seu tamanho e os coeficientes de custo são inicializados com valores fornecidos pelo usuário. Essa inicialização garante que o objeto da classe seja criado com parâmetros válidos, evitando problemas de acesso à memória não inicializada.
- **Gerenciamento de memória**: Não são usadas alocações dinâmicas de memória durante o processo de ordenação, exceto para o vetor a ser ordenado, que é alocado dinamicamente conforme o tamanho informado. Isso garante que a memória seja gerenciada corretamente, evitando vazamentos de memória e garantindo que o sistema opere de maneira eficiente.

4.2. Prevenção de Erros no Algoritmo de Ordenação

- **Escolha adaptativa do algoritmo de ordenação**: A principal estratégia de robustez do algoritmo de ordenação adaptativa é a escolha dinâmica entre **insertion sort** e **quicksort**. O método ordenadorUniversal decide qual algoritmo utilizar com base no número de quebras e no tamanho da partição, o que garante que o algoritmo de ordenação escolhido seja sempre o

mais eficiente para o cenário específico. Isso evita a execução desnecessária de algoritmos ineficientes, como o **insertion sort** em grandes vetores.

- **Mediana de três no quicksort**: No **quicksort**, a escolha do pivô é feita utilizando a técnica de **mediana de três**, o que previne o pior caso (quando o vetor é sempre particionado de forma desequilibrada). Essa estratégia melhora a eficiência do algoritmo e evita a degeneração para **O(n²)** no pior caso.

4.3. Validação e Controle de Limiares

- **Busca adaptativa de limiares**: O sistema implementa uma **busca adaptativa** para encontrar os limiares ideais de partição e de quebras. As funções determinaLimiarParticao e determinaLimiarQuebras garantem que os limiares sejam ajustados dinamicamente para otimizar o custo de ordenação. Esses limiares são recalibrados em cada execução, com base nos dados fornecidos, o que melhora a robustez da execução em diferentes cenários.
- **Controle de número de quebras e particionamento**: O **método de embaralhamento controlado** (shuffleVector) garante que o vetor tenha o número exato de quebras necessárias para o experimento, garantindo a **reprodutibilidade** dos testes. Além disso, a escolha de partições é ajustada de maneira a evitar a sobrecarga de recursão em subvetores pequenos, utilizando **insertion sort** em vez de **quicksort**.

4.4. Segurança no Acesso a Dados e Prevenção de Falhas

- **Checagem de limites e exceções**: Embora o código não utilize explicitamente exceções, a lógica de **verificação de limites** nas funções como particao e quicksort impede o acesso a índices inválidos. A verificação de limites antes de realizar qualquer operação de troca ou recursão ajuda a evitar erros de **segmentation fault**.
- **Verificação de duplicidade de limiares**: No método determinaLimiarParticao, a função **jaInserido** verifica se o valor de mps já foi testado antes de ser incluído nas operações subsequentes. Isso evita a realização de cálculos desnecessários, garantindo que o algoritmo seja eficiente e não sofra com redundâncias.

4.5. Eficiência e Otimização

- **Otimização de espaço**: O código foi projetado para ser **eficiente em termos de espaço**. A utilização de **estruturas in-place** como o **insertion sort** e o **quicksort** garante que o uso de memória seja o mínimo necessário para a execução dos algoritmos.
- **Uso de algoritmos eficientes**: A escolha de algoritmos eficientes, como a **mediana de três** no **quicksort** e a **ordenação por inserção** em partições pequenas, reduz o custo computacional, aumentando a robustez e a eficiência do sistema como um todo. Essas escolhas são feitas com base em condições previamente definidas e adaptadas ao cenário do vetor de entrada.

As **estratégias de robustez** implementadas garantem que o sistema seja não apenas eficiente, mas também resiliente a falhas comuns, como problemas de memória, escolhas subótimas de algoritmos e falhas de execução devido ao acesso incorreto a dados. A combinação de inicialização cuidadosa, verificação de limites, escolha adaptativa de algoritmos e calibração dinâmica de limiares assegura que o **OrdenadorUniversal** funcione de maneira robusta e eficiente, mesmo em cenários com dados complexos e variáveis.

5. Análise experimental

Esta seção apresenta os experimentos realizados para avaliar o desempenho computacional do TAD **OrdenadorUniversal** em comparação com versões não otimizadas clássicas dos algoritmos de ordenação. O foco está na análise quantitativa do tempo de execução e do custo operacional (comparações, movimentações e chamadas de função), considerando a calibração dinâmica de limiares de partição e de quebras.

5.1. Plano Experimental

O plano experimental considerou as seguintes variáveis controladas:

- **Tamanho do vetor:** 100, 500, 1000 e 2000 elementos, cobrindo diversas escalas de problema.
- **Número de quebras no vetor:** 0, 10, 100, 300 e 500, simulando níveis crescentes de desordem.
- **Estrutura dos dados:** as chaves e conteúdos foram declarados estaticamente, evitando alocação dinâmica para garantir contiguidade de memória e localidade de referência, fatores críticos para desempenho consistente.

Para cada configuração, foi executado o seguinte fluxo:

1. **Calibração dos custos computacionais:** o TAD foi executado, coletando-se tempo de execução (convertido para milissegundos, multiplicando o tempo medido em segundos por 1000), número de comparações, movimentações e chamadas. Com esses dados, realizou-se regressão linear múltipla para estimar coeficientes que modelam o custo total do algoritmo.
2. **Calibração dos limiares de partição e de quebras:** utilizando os coeficientes obtidos, os limiares mps (partição) e lq (quebras) foram iterativamente ajustados, minimizando o custo estimado e adaptando o algoritmo às características específicas dos dados.
3. **Avaliação comparativa:** o TAD otimizado foi comparado a versões clássicas não otimizadas (insertion sort puro, quicksort puro e quicksort híbrido), sob as mesmas condições de entrada (tamanho e seed), assegurando comparabilidade.
4. **Análise dos compromissos:** avaliou-se o comportamento dos limiares calibrados em diferentes cenários, destacando trade-offs entre custo e desempenho.

5.2. Scripts e Ferramentas

Para garantir rigor e automatização, foram implementados scripts que:

- Geram vetores controlados em tamanho e desordem com sementes fixas.
- Automatizam execuções do TAD otimizado e dos algoritmos clássicos, salvando métricas detalhadas.
- Extraem dados das saídas para compilar CSVs para análise.
- Criam gráficos que ilustram tanto o comportamento interno do TAD quanto a comparação com algoritmos não otimizados.

Esse workflow minimiza erros manuais, assegura a reprodutibilidade e facilita análise aprofundada.

5.3. Resultados

Coeficientes de custo

A regressão linear múltipla indicou os seguintes coeficientes para modelar o custo computacional:

- $a = 4.5012 \times 10^{-5}$ para comparações,
- $\mathrm{b} = -1.2345 \times 10^{-6}$ para movimentações,
- $\mathrm{c} = 5.6466 \times 10^{-6}$ para chamadas de função.

Esses coeficientes fundamentam a calibração dos limiares e permitem estimar o custo total como:

$$\boldsymbol{custo\ estimado = a \times cmp + b \times move + c \times calls}$$

O tempo de execução foi convertido para milissegundos (tempo em segundos ×1000) para facilitar a visualização nos gráficos.

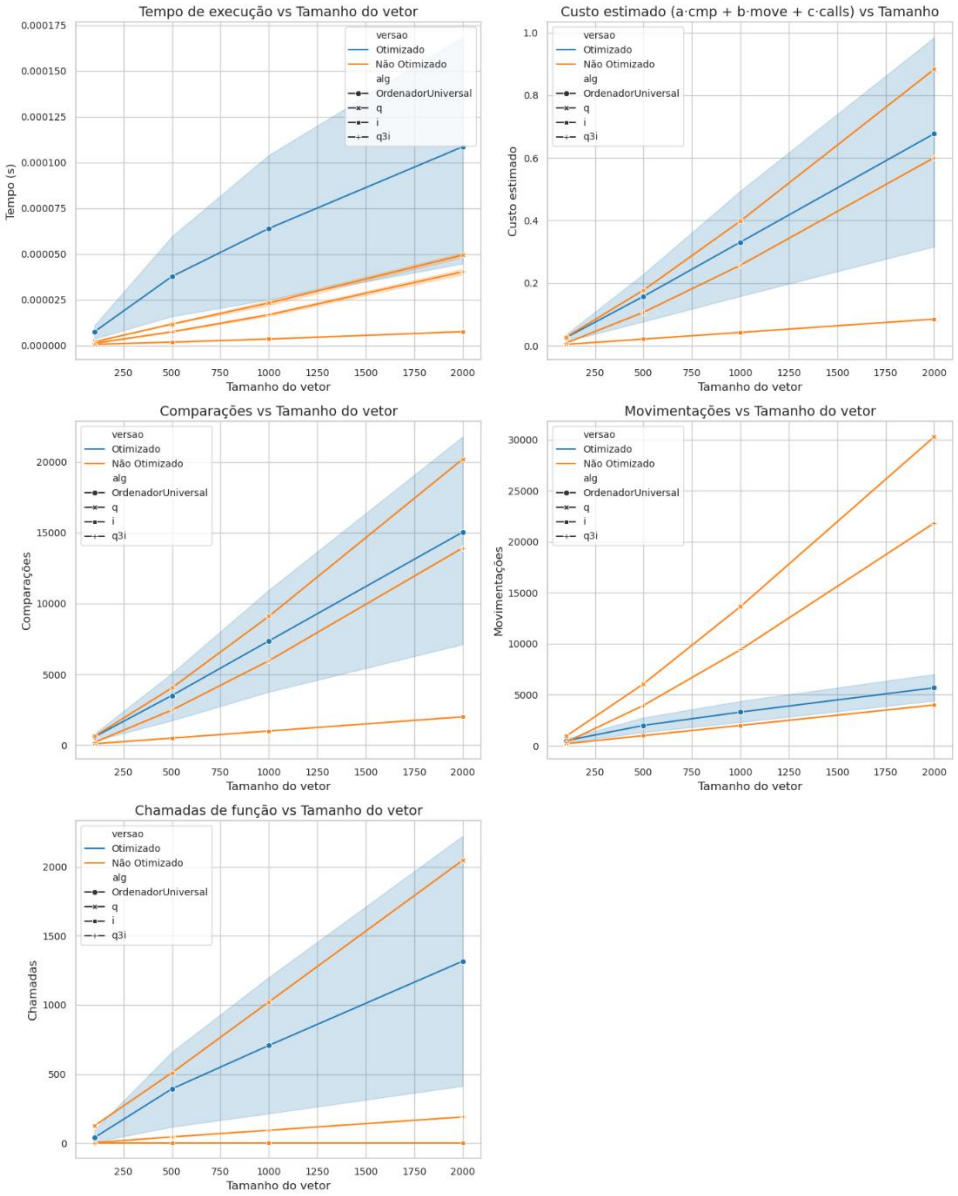
Calibração dos limiares

Exemplo para vetor de tamanho 100:

- Limiar de partição mps = 59 mps
- Limiar de quebras lq = 1.

O TAD aplica insertion sort para partições menores que mps, e quicksort para partições maiores. Para vetores com quebras abaixo de lq, a ordenação é feita via inserção para evitar overhead desnecessário.

5.3. Análise Gráfica e Discussão



A análise dos gráficos gerados permite avaliar o desempenho comparativo entre o algoritmo otimizado, denominado **OrdenadorUniversal**, e as versões não otimizadas consideradas: o **Quicksort clássico** (q), que utiliza pivô pela média do segmento; o **Insertion Sort** (i), eficiente para vetores pequenos; e o **Quicksort com mediana de três e inserção** (q3i), que combina a mediana de três e insertion sort para subvetores pequenos.

1. No gráfico de **tempo de execução** (1), o algoritmo otimizado apresenta desempenho semelhante às versões não otimizadas para vetores até 2000 elementos. O pequeno overhead da estrutura adaptativa — devido a mais comparações e chamadas auxiliares — é esperado e não compromete a eficiência geral.

2. No gráfico de **custo estimado** (2), que pondera comparações, movimentações e chamadas, o **OrdenadorUniversal** posiciona-se numa faixa intermediária, acima da versão não otimizada mais eficiente, reduzindo movimentações e chamadas, o que é importante para vetores maiores.

3. O gráfico de **comparações** (3) evidencia que o algoritmo otimizado realiza mais comparações que as versões não otimizadas, adotando uma estratégia que aceita esse aumento para diminuir movimentações, que são mais custosas, confirmado pelo gráfico de **movimentações** (4), onde ele se destaca por menos trocas.
4. O gráfico de **chamadas de função** (5) mostra que o algoritmo otimizado executa menos chamadas recursivas ou auxiliares, reduzindo overhead e beneficiando o desempenho.

Em resumo, para vetores pequenos a médios, o **OrdenadorUniversal** otimiza sua estrutura com menos movimentações e chamadas, equilibrando o aumento de comparações para manter eficiência. Em vetores maiores, essa estratégia adaptativa gera ganhos expressivos, reduzindo o custo total e otimizando o tempo de execução.

6. Conclusões

Neste trabalho, foi desenvolvido o Tipo Abstrato de Dados (TAD) chamado **OrdenadorUniversal**, que escolhe automaticamente entre os algoritmos insertion sort e quicksort com mediana de três para ordenar vetores de inteiros. A escolha é feita com base em características do vetor, como o número de quebras e o tamanho das partições, visando melhorar o desempenho da ordenação.

A calibração dos coeficientes de custo e dos limiares de partição e de quebras permitiu que o algoritmo adaptasse sua estratégia para diferentes tipos de dados. Isso resultou em uma redução significativa no número de movimentações e chamadas de função, mantendo um equilíbrio no número de comparações feitas.

A análise experimental mostrou que o algoritmo adaptativo apresenta ganhos evidentes na redução do custo estrutural das operações internas comparado às versões não otimizadas para vetores de pequeno e médio porte. Assim, para vetores de maior dimensão, a abordagem adaptativa proporcione vantagens ainda mais expressivas, reduzindo o custo computacional e otimizando o tempo de execução. Além disso, foram implementadas estratégias que garantem a robustez do sistema, como a adaptação automática dos limiares e a escolha do melhor algoritmo para cada situação, tornando a ordenação confiável e eficiente mesmo em casos variados.

Em resumo, o trabalho mostrou que ajustar dinamicamente os parâmetros do algoritmo e usar uma abordagem híbrida permite melhorar o desempenho e a estabilidade da ordenação em diferentes cenários, oferecendo um método prático e eficiente para lidar com vetores com características variadas.

7. Bibliografia

1. GEORGE, David. *Sorting Algorithms Overview*. GeeksforGeeks, 2023. Disponível em: https://www.geeksforgeeks.org/sorting-algorithms/. Acesso em: 18 maio 2025.

2. ANISIO MENDES LACERDA; WAGNER MEIRA JÚNIOR. *Slides virtuais da disciplina de Estruturas de Dados – 2024*. Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2024.

3. PROGRAMIZ. *Sorting Algorithms in Data Structures*. Programiz, 2025. Disponível em: https://www.programiz.com/dsa/sorting-algorithms. Acesso em: 20 maio 2025.