UNIVERSIDADE PAULISTA

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

BRASÍLIA

2024

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

Trabalho de Atividades Práticas Supervisionadas do curso de Ciência da Computação 4° semestre, apresentado a Universidade Paulista – UNIP, como parte do título de bacharel em Ciência da Computação.

Orientador: Prof. Diego Rodrigues

BRASÍLIA

2024

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

Trabalho de Atividades Práticas Supervisionadas do curso de Ciência da Computação 4° semestre, apresentado a Universidade Paulista – UNIP, como parte do título de bacharel em Ciência da Computação.

Orientador: Prof. Diego Rodrigues

Aprovado em:

Prof. Cristiano Silva

Universidade Paulista- UNIP

/ /

RESUMO

**Palavras-chaves:**

ABSTRACT

**Keywords:**

Sumário

[1 OBJETIVO E MOTIVAÇÃO DO TRABALHO 6](#_Toc179998320)

[2 INTRODUÇÃO 7](#_Toc179998321)

[3 REFERENCIA TEÓRICO 8](#_Toc179998322)

[3.1 BubbleSort 8](#_Toc179998323)

[3.2 QuickSort 9](#_Toc179998324)

[3.3 InsertionSort 10](#_Toc179998325)

[3.4 BinaryInsertionSort 10](#_Toc179998326)

[3.5 SelectionSort 10](#_Toc179998327)

[3.6 HeapSort 11](#_Toc179998328)

[3.7 MergeSort 12](#_Toc179998329)

[3.8 BucketSort 14](#_Toc179998330)

[4 PLANO DE DESENVOLVIMENTO DO JOGO 17](#_Toc179998331)

[5 PROJETO 18](#_Toc179998332)

[6 RELATÓRIO COM AS LINHAS DE CÓDIGO DO PROGRAMA 19](#_Toc179998333)

[BIBLIOGRAFIA 20](#_Toc179998334)

[CARÔMETRO 21](#_Toc179998335)

# 1 OBJETIVO E MOTIVAÇÃO DO TRABALHO

# 2 INTRODUÇÃO

# 3 REFERENCIA TEÓRICO

**3.1 Definição de Algoritmos**

Um algoritmo é uma sequência finita e bem definida de instruções ou operações projetadas para resolver um problema específico ou realizar uma tarefa, geralmente de maneira lógica e ordenada. De acordo com Cormen et al. (2009, p. 5), "algoritmos são a essência da computação e descrevem, em sua forma mais elementar, uma sequência de passos para realizar uma determinada tarefa." No contexto da ciência da computação, os algoritmos são fundamentais para o desenvolvimento de programas de computador e de sistemas complexos, sendo utilizados em uma ampla gama de aplicações, desde tarefas simples até processos altamente sofisticados.

#BUSCA(BINARIA E SEQUENCIAL)

#PYTHON

**3.2 BubbleSort**

3.2.1 Introdução ao BubbleSort

O Bubble Sort é um algoritmo simples de ordenação que itera repetidamente sobre uma lista, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. A cada iteração, o maior elemento "flutua" para o topo da lista, como uma bolha, de onde vem o nome do algoritmo. Embora seja fácil de entender e implementar, o Bubble Sort é considerado ineficiente para grandes conjuntos de dados, uma vez que seu tempo de execução é O(n2)O(n^2)O(n2), onde nnn é o número de elementos na lista (CORMEN et al., 2009).

3.2.2 Fundamentos do BubbleSort

O funcionamento do Bubble Sort pode ser descrito da seguinte forma:

1. Percorrer a lista do primeiro ao penúltimo elemento;
2. Para cada par de elementos adjacentes, se o elemento atual for maior que o próximo, os dois são trocados;
3. Após cada iteração, o maior elemento não estará mais na lista não ordenada, e o processo se repete com a parte restante da lista.

Apesar da simplicidade, o Bubble Sort possui uma grande desvantagem em termos de eficiência. Em situações onde se busca otimização, outros algoritmos de ordenação, como o Merge Sort e o QuickSort, tendem a ser preferidos, uma vez que têm complexidade O(nlog⁡n)O(n \log n)O(nlogn) (CORMEN et al., 2009; SEDGEWICK, 2002).

3.2.3 Aplicações do Bubble Sort

O Bubble Sort é geralmente utilizado em contextos educacionais, pois oferece uma introdução intuitiva ao conceito de ordenação e algoritmos iterativos. No entanto, seu uso em sistemas reais é raro devido à sua baixa eficiência em comparação com algoritmos mais avançados. Segundo Menezes (2014), o Bubble Sort pode ser útil em situações onde o número de elementos a ser ordenado é muito pequeno, ou quando já se sabe que a lista está quase ordenada, aproveitando a possibilidade de detecção de uma lista ordenada antes do final do processo.

Em termos práticos, em sistemas computacionais modernos, o Bubble Sort raramente é utilizado diretamente, sendo mais comum em exemplos acadêmicos e didáticos, para explicar a lógica básica de trocas e iterações (NIVENS, 2011).

3.2.4 Vantagens e Desvantagens

**3.3 QuickSort**

3.3.1 Introdução ao QuickSort

O QuickSort, ou ordenação rápida, foi desenvolvido por Tony Hoare em 1960. É um algoritmo de ordenação recursivo que utiliza a técnica de divisão e conquista. Seu princípio básico é dividir o array em duas partes com base em um elemento denominado pivô.

3.3.2 Fundamentos do QuickSort

O QuickSort realiza a ordenação da seguinte maneira (HOARE, 1962):

1. Seleciona um pivô (geralmente o primeiro ou último elemento da lista, ou um elemento escolhido aleatoriamente);
2. Organiza os elementos de modo que todos os elementos menores que o pivô fiquem à esquerda, e todos os elementos maiores fiquem à direita;
3. Recursivamente, aplica o mesmo procedimento às sublistas esquerda e direita.

O tempo de execução do QuickSort, no melhor caso e no caso médio, é O(nlog⁡n)O(n \log n)O(nlogn), sendo, portanto, um dos algoritmos de ordenação mais rápidos para a maioria das entradas. No entanto, no pior caso, quando o pivô escolhido não é ideal, a complexidade pode atingir O(n2)O(n^2)O(n2). Para mitigar esse problema, diversas variações do QuickSort empregam técnicas de escolha de pivô mais sofisticadas, como o "pivô aleatório" ou o "pivô mediano" (SEDGEWICK, 2002).

3.3.3 Aplicações do QuickSort

Devido à sua eficiência e flexibilidade, o QuickSort é amplamente utilizado em contextos de ordenação de grandes volumes de dados. Ele é particularmente eficaz em situações em que os dados podem ser carregados na memória principal e a ordenação precisa ser feita rapidamente. Segundo Cormen et al. (2009), o QuickSort é uma escolha preferida em sistemas de alto desempenho, como bancos de dados, sistemas operacionais e aplicações de ciência de dados, onde a velocidade e a eficiência são críticas.

3.3.4 Vantagens e Desvantagens

O QuickSort se destaca por sua eficiência em média, sua implementação relativamente simples e pelo fato que não requer espaço adicional significativo, uma vez que é um algoritmo in-place. Além disso, ele é amplamente utilizado em várias bibliotecas padrão de linguagens de programação, como o C e o Python, devido à sua performance em entradas diversas (MENEZES, 2014).

No entanto, uma das principais desvantagens do QuickSort é sua instabilidade, já que a ordem relativa de elementos iguais pode não ser preservada após a ordenação. Além disso, como mencionado, no pior caso, seu desempenho pode ser degradado para O(n2)O(n^2)O(n2), mas isso é raramente observado na prática quando implementado com boas heurísticas de escolha de pivô (NIVENS, 2011).

**3.4 InsertionSort**

3.4.1 Introdução ao InsertionSort

3.4.2 Fundamentos do InsertionSort

3.4.3 Aplicações do InsertionSort

3.4.4 Vantagens e Desvantagens

**3.5 BinaryInsertionSort**

3.5.1 Introdução ao BinaryInsertionSort

3.5.2 Fundamentos do BinaryInsertionSort

3.5.3 Aplicações do BinaryInsertionSort

3.5.4 Vantagens e Desvantagens

**3.6 SelectionSort**

3.6.1 Introdução ao SelectionSort

O SelectionSort é um método de ordenação por seleção, onde é feito uma troca de elementos das matrizes. O método é pegar o menor elemento e trocar sua posição com o primeiro, após isso, o próximo menor elemento troca de posição com o segundo e assim continua até que o algoritmo seja ordenado corretamente, podendo ser ordenado de forma crescente ou decrescente.

3.6.2 Fundamentos do SelectionSort

3.6.2.1 Processo de Seleção e Troca

O Selection Sort realiza a ordenação selecionando, a cada passo, o menor elemento da porção não ordenada do array e o trocando com o primeiro elemento não ordenado. Este processo de seleção e troca é repetido até que todos os elementos estejam em suas posições corretas. Weiss (2012, p. 42) explica que "a simplicidade do algoritmo advém de sua estratégia de seleção direta, em que cada elemento é colocado diretamente em sua posição final após ser encontrado."

3.6.2.2 Complexidade Computacional

A complexidade de tempo do Selection Sort é O(n²) para o melhor, o pior e o caso médio. Esse comportamento quadrático ocorre porque o algoritmo percorre toda a lista para encontrar o menor elemento em cada iteração. Knuth (1998, p. 130) descreve que "o Selection Sort é consistentemente O(n²), independentemente da organização inicial dos elementos, o que limita sua aplicabilidade para grandes conjuntos de dados."

3.6.3 Comparações e Trocas

No Selection Sort, a quantidade de comparações realizadas é consistente, já que ele sempre percorre a lista completa para encontrar o menor elemento. No entanto, o número de trocas é menor em comparação com outros algoritmos quadráticos, como o Bubble Sort. Sedgewick e Wayne (2011, p. 78) comentam que "apesar do número elevado de comparações, o Selection Sort tem o benefício de realizar um número limitado de trocas, o que o torna interessante em cenários onde as trocas de elementos são mais custosas que as comparações."

3.6.3 Aplicações do Selection Sort

Embora o Selection Sort não seja o algoritmo de ordenação mais eficiente, ele possui algumas aplicações específicas em que sua simplicidade e baixo número de trocas o tornam útil. Segundo Cormen et al. (2009, p. 103), o Selection Sort "é adequado para listas pequenas e em cenários onde a simplicidade do código e a previsibilidade do desempenho são mais importantes que a eficiência."

3.6.3.1 Ordenação de Pequenos Conjuntos de Dados

O Selection Sort é frequentemente utilizado para ordenar pequenos conjuntos de dados ou em contextos educacionais, onde sua lógica simples ajuda a ilustrar os princípios básicos de ordenação. Weiss (2012, p. 45) destaca que "em conjuntos de dados pequenos, o Selection Sort oferece uma solução direta e fácil de implementar, com desempenho aceitável."

3.6.3.2 Uso em Sistemas com Restrição de Trocas

Em sistemas onde a troca de elementos é uma operação custosa, o Selection Sort pode ser preferível ao Bubble Sort, pois, embora tenha a mesma complexidade, realiza menos trocas. Knuth (1998, p. 132) comenta que "o baixo número de trocas do Selection Sort o torna vantajoso em ambientes onde a movimentação de elementos é restrita ou limitada por custos."

3.6.4 Vantagens e Desvantagens

O SelectionSort é simples e fácil de implementar, porém não muito eficaz para listas grandes, algo também observado por Thomas Cormen, autor do livro Desmitificando Algritmos, de acordo com sua analise “A primeira é que veremos que seu tempo de execução assintótico de Θ (n2) é o pior dos algoritmos de ordenação que examinaremos.” (Desmitificando Algoritmos, p.29).

**3.7 HeapSort**

3.7.1 Introdução ao HeapSort

Diferente do método ‘SelectionSort’, o método ‘HeapSort’ não faz comparações entre o menor elemento com os maiores, mas sim do maior, reorganizando de uma forma que o maior elemento fique na última posição e o próximo maior, caso exista, depois dele ocupando assim a última posição, continuando até que a lista esteja ordenada.

Esse método usa uma estrutura de dados ‘Binary Heap’ que simboliza uma árvore binária, onde o maior elemento é a raiz dessa árvore, os elementos filhos a esquerda são verificados para saber se são maiores que a raiz, os filhos a direita são verificados para saber se são maiores que o maior elemento até agora. A prioridade é levar o maior elemento para esquerda primeiramente.

3.7.2 Fundamentos do HeapSort

3.7.2.1 Estrutura de Dados Heap

O HeapSort depende da estrutura heap, na qual o elemento raiz é sempre o maior (em um max-heap) ou o menor (em um min-heap). A propriedade de ordem do heap permite que o algoritmo selecione repetidamente o maior ou menor elemento, facilitando a construção de uma lista ordenada. Weiss (2012, p. 210) afirma que "a estrutura de heap fornece um método eficaz para manter e acessar o elemento de maior prioridade, fundamental para o HeapSort."

3.7.2.2 Construção do Heap

Para aplicar o HeapSort, o primeiro passo é construir o max-heap a partir do array não ordenado. Esse processo transforma o array em uma estrutura onde o maior elemento se encontra na raiz, o que facilita a ordenação subsequente. De acordo com Knuth (1998, p. 176), "a construção de um max-heap pode ser feita em tempo linear, garantindo a eficiência da primeira etapa do HeapSort."

3.7.2.3 Remoção do Maior Elemento

Após construir o max-heap, o algoritmo remove o maior elemento (a raiz) e o coloca no final do array. Em seguida, o heap é reestruturado para que o próximo maior elemento se torne a nova raiz, e o processo é repetido até que todos os elementos estejam ordenados. Sedgewick e Wayne (2011, p. 235) descrevem que "o HeapSort utiliza a estrutura de heap de forma a garantir a extração ordenada dos maiores elementos sem a necessidade de armazenamento extra."

3.7.3 Aplicações do HeapSort

3.7.3.1 Ordenação em Sistemas de Baixa Memória

O HeapSort é particularmente útil em sistemas com restrições de memória, já que sua ordenação in-place o torna mais eficiente do que algoritmos que requerem armazenamento auxiliar. Segundo Knuth (1998, p. 178), "o HeapSort oferece uma solução de ordenação eficaz para ambientes com memória limitada, onde outras técnicas de ordenação poderiam não ser viáveis."

3.7.3.2 Sistemas de Prioridade

A estrutura de heap permite a implementação eficiente de filas de prioridade, tornando o HeapSort uma base para sistemas que exigem uma classificação rápida e constante dos maiores ou menores elementos. Cormen et al. (2009, p. 166) observam que "a estrutura do heap é ideal para cenários de filas de prioridade, onde o acesso ao elemento de maior prioridade deve ser mantido de forma rápida."

3.7.4 Vantagens e Desvantagens

O Heap Sort tem uma complexidade de tempo de O(n log n) no pior caso, o que o torna eficiente para ordenar listas grandes e utiliza uma estrutura de dados chamada heap, que é útil em várias aplicações, como gerenciamento de prioridades.

Porém, comparado a outros algoritmos como Quick Sort e Bubble Sort, o Heap Sort pode ser menos intuitivo e mais difícil de entender, especialmente na construção e manutenção da estrutura de heap. Embora a complexidade teórica seja O(n log n), na prática, o Heap Sort pode ser mais lento que outros algoritmos de ordenação, como Quick Sort, devido

**3.8 MergeSort**

3.8.1 Introdução ao MergeSort

O MergeSort é um algoritmo de ordenação baseado no paradigma de dividir para conquistar (*divide and conquer*), que foi inicialmente descrito por John von Neumann em 1945. O algoritmo divide repetidamente a lista ou array em duas metades, ordena cada metade de maneira recursiva e, em seguida, combina as duas metades para produzir uma lista final ordenada. Segundo Cormen et al. (2009, p. 34), "MergeSort é um algoritmo eficiente de ordenação com complexidade de tempo O(n log n) em todos os casos, sendo também estável."

3.8.2 Fundamentos do MergeSort

O algoritmo merge sort divide, recursivamente, o array em duas partes até que cada posição dele seja considerada como um array de um único elemento. Em seguida, o algoritmo combina dois arrays de forma a obter um array maior e ordenado. Essa combinação dos arrays é feita intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). Esse processo se repete até que exista apenas um array. (BACKES, André, 2022, p. 34)

3.8.2.1 Divisão e Conquista

O MergeSort exemplifica o paradigma de dividir para conquistar, onde um problema é decomposto em partes menores e mais gerenciáveis. Cada subproblema é resolvido de forma recursiva, e as soluções dessas partes são combinadas. Conforme Sedgewick e Wayne (2011, p. 145), o princípio fundamental do MergeSort está na "divisão da lista original em sublistas até que o problema seja trivial, ou seja, cada sublista contenha apenas um elemento."

3.8.2.2 Recursividade

O uso de recursividade é essencial para a operação do MergeSort. O processo de divisão recursiva continua até que as sublistas tenham apenas um elemento. De acordo com Weiss (2012, p. 78), "a recursividade do MergeSort permite que listas de qualquer tamanho sejam divididas até o ponto onde a fusão seja direta e rápida."

3.8.2.3 Complexidade Computacional

A complexidade de tempo do MergeSort é O(n log n**)** no melhor, pior e caso médio, o que o torna um dos algoritmos mais consistentes. Knuth (1998, p. 172) observa que "a eficiência do MergeSort em tempo O(n log n) se deve à divisão da lista em duas metades, seguida da combinação das duas listas ordenadas em tempo linear."

3.8.2.4 Estabilidade

A estabilidade do MergeSort é uma de suas características mais notáveis. Algoritmos estáveis de ordenação, como o MergeSort, garantem que a ordem relativa de elementos iguais seja mantida após a ordenação. Cormen et al. (2009, p. 39) ressaltam que "a estabilidade do MergeSort o torna útil em situações em que a preservação da ordem original é desejável."

3.8.3 Aplicações do MergeSort

O MergeSort é amplamente utilizado em sistemas que lidam com grandes volumes de dados. Ele é especialmente eficaz em situações onde os dados precisam ser processados em memória externa, como discos rígidos, uma vez que pode lidar eficientemente com grandes conjuntos de dados que não cabem na memória RAM. Knuth (1998, p. 181) afirma que "o MergeSort é particularmente útil em sistemas de arquivos externos e grandes bases de dados, onde a eficiência da memória é crucial."

3.8.4 Vantagens e Desvantagens

3.8.4.1 Vantagens

Entre as principais vantagens do MergeSort, destacam-se a estabilidade e a complexidade de tempo previsível. Cormen et al. (2009, p. 50) apontam que "um dos principais benefícios do MergeSort é a sua estabilidade e a consistência da complexidade O(n log n) em todos os casos."

3.8.4.2 Desvantagens

Um dos pontos negativos do MergeSort é o uso de memória adicional. Durante o processo de fusão, o algoritmo precisa de memória extra para armazenar as sublistas temporárias. Weiss (2012, p. 91) ressalta que "o principal problema do MergeSort é a necessidade de espaço adicional de O(n), o que pode limitar sua aplicabilidade em sistemas com restrições severas de memória.".

**3.9 BucketSort**

3.9.1 Introdução ao BucketSort

O BucketSort é um algoritmo de ordenação que distribui os elementos de uma lista em diferentes baldes (*buckets*), os quais são posteriormente ordenados individualmente e combinados para produzir a lista final ordenada. Esse algoritmo é eficaz quando os elementos são distribuídos de forma relativamente uniforme dentro de um intervalo finito. Segundo Cormen et al. (2009, p. 201), o BucketSort "funciona bem para dados que podem ser uniformemente distribuídos em intervalos, pois aproveita essa propriedade para organizar os elementos em grupos, simplificando o processo de ordenação."

3.9.2 Fundamentos do BucketSort

3.9.2.1 Organização em Baldes

O principal conceito do BucketSort é a distribuição dos elementos em baldes, ou recipientes, que representam intervalos definidos. Cada balde contém elementos que pertencem a um determinado intervalo, facilitando a ordenação local em cada balde. Conforme Sedgewick e Wayne (2011, p. 210), "a eficiência do BucketSort reside na capacidade de dividir os dados em subconjuntos menores, que podem ser ordenados independentemente."

3.9.2.2 Utilização de Algoritmos Auxiliares

Após distribuir os elementos nos baldes, o algoritmo utiliza um método auxiliar, como o InsertionSort ou o QuickSort, para ordenar os elementos em cada balde. De acordo com Weiss (2012, p. 120), "o BucketSort se beneficia de algoritmos de ordenação secundários, que podem ser aplicados aos baldes com poucos elementos, onde algoritmos como o InsertionSort podem ser extremamente eficazes."

3.9.2.3 Complexidade Computacional

A complexidade de tempo do BucketSort depende da distribuição uniforme dos elementos e da eficiência do algoritmo auxiliar usado em cada balde. No melhor caso, a complexidade é **O(n)**, enquanto no pior caso pode ser **O(n²)**, se todos os elementos forem colocados em um único balde. Knuth (1998, p. 256) ressalta que "a complexidade do BucketSort pode ser extremamente favorável em situações onde a distribuição dos dados é homogênea, oferecendo um desempenho linear."

3.9.3 Aplicações do BucketSort

O BucketSort é amplamente utilizado em aplicações onde os dados são distribuídos uniformemente e podem ser mapeados para intervalos finitos. É comumente aplicado em sistemas que lidam com números flutuantes, uma vez que sua divisão em intervalos facilita o processamento. Knuth (1998, p. 259) aponta que "em sistemas de análise estatística e gráficos de dispersão, o BucketSort demonstra seu valor, principalmente ao ordenar grandes conjuntos de dados numéricos."

3.9.4 Vantagens e Desvantagens

3.9.4.1 Vantagens

Entre as principais vantagens do BucketSort, destaca-se a sua eficiência em cenários onde os dados são distribuídos uniformemente. Além disso, ao dividir o problema em subproblemas menores, o algoritmo pode se beneficiar de algoritmos mais simples e eficientes em casos pequenos. Cormen et al. (2009, p. 215) afirmam que "o BucketSort pode ser altamente eficiente em situações onde os dados possuem uma distribuição uniforme, o que possibilita um desempenho linear."

3.9.4.2 Desvantagens

Uma desvantagem significativa do BucketSort é a sua dependência da distribuição uniforme dos dados. Caso os elementos estejam concentrados em um pequeno número de baldes, o algoritmo pode perder sua eficiência e apresentar desempenho semelhante ao do InsertionSort. Weiss (2012, p. 123) alerta que "a eficiência do BucketSort depende fortemente da capacidade de distribuir os elementos de forma equitativa entre os baldes, o que pode não ocorrer em todos os casos práticos."

# 4 PLANO DE DESENVOLVIMENTO DO JOGO

# 5 PROJETO

# 6 RELATÓRIO COM AS LINHAS DE CÓDIGO DO PROGRAMA

**BIBLIOGRAFIA**

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*. 2nd ed. Reading: Addison-Wesley, 1998.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4th ed. Reading: Addison-Wesley, 2011.

WEISS, M. A. *Data Structures and Algorithm Analysis in C*. 4th ed. Boston: Pearson, 2012.

Drozdek, A. Estrutura de Dados e Algoritmos em C++: Tradução da 4ª Edição Norte-Americana. 2ª Edição. São Paulo: Cengage Learning, 2017.

CORMEN, Thomas H. et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2009.

HOARE, Charles Antony Richard. Quicksort. *The Computer Journal*, v. 5, n. 1, p. 10-15, 1962.

MENEZES, Ricardo. Algoritmos de ordenação: uma abordagem didática. São Paulo: Editora XYZ, 2014.

NIVENS, Scott. Sorting Algorithms Explained. Porto Alegre: Novatec, 2011.

SEDGEWICK, Robert. Algoritmos em C. 3. ed. São Paulo: Addison-Wesley, 2002.

Drozdek, A. Estrutura de Dados e Algoritmos em C++: Tradução da 4ª Edição Norte-Americana. 2ª Edição. São Paulo: Cengage Learning, 2017.

# CARÔMETRO

|  |  |  |  |
| --- | --- | --- | --- |
| Samara dos Santos Sales | Victor Hugo de Oliveira Penga | Roberta Teixeira Silva Costa | Rafael Milani |
|  |  |  |  |