UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

LEANDRO MARCOS MENDES ZANETTI

ALGORITMOS DE ORDENAÇÃO EM FOCO: VISUALIZAÇÃO, INTERAÇÃO E AVALIAÇÃO DE DESEMPENHO

Ouro Preto
2026

LEANDRO MARCOS MENDES ZANETTI

ALGORITMOS DE ORDENAÇÃO EM FOCO: VISUALIZAÇÃO, INTERAÇÃO E AVALIAÇÃO DE DESEMPENHO

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Ouro Preto
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Leandro Marcos Mendes Zanetti

Algoritmos de ordenação em foco: visualização, interação e avaliação de desempenho

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 24 de Fevereiro de 2026

Membros da banca

Rafael Alves Bonfim de Queiroz (Orientador) - Doutor - Universidade Federal de Ouro Preto
Guilherme Tavares de Assis (Examinador) - Doutor - Universidade Federal de Ouro Preto
Pedro Henrique Lopes Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto

Rafael Alves Bonfim de Queiroz, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 24/02/2026

Documento assinado eletronicamente por **Rafael Alves Bonfim de Queiroz**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 04/03/2026, às 13:36, conforme horário oficial de Brasília, com fundamento no art. 6°, § 1°, do Decreto n° 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1062458** e o código CRC **A5D8A218**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus pais, pelo amor incondicional e apoio em todas as etapas da minha vida acadêmica, ao meu irmão, que esteve presente e vivenciou comigo todos os altos e baixos durante o processo da minha formação e a todos que, de forma direta ou indireta, me incentivaram a seguir firme no caminho da Ciência da Computação.

Agradecimentos

A Deus, pela força e pela saúde que me sustentaram ao longo desta jornada. Aos meus pais e familiares, pelo carinho, incentivo e compreensão nos momentos de dificuldade. Ao meu orientador, Prof. Dr. Rafael Alves Bonfim de Queiroz, pela orientação dedicada, paciência e pelas valiosas contribuições para a realização deste trabalho. Por fim, agradeço a todos que, de alguma forma, contribuíram para que este trabalho fosse concretizado.

“A educação não transforma o mundo. Educação muda pessoas. Pessoas transformam o mundo.”

— (Paulo Freire, 1996)

Resumo

A ordenação de dados é um tema central na Ciência da Computação, amplamente abordado no ensino de estruturas de dados e algoritmos. Este trabalho, de natureza aplicada, apresenta o desenvolvimento da ferramenta educacional "Estou com Sort", concebida para a visualização, interação e análise de desempenho de algoritmos de ordenação. O objetivo geral consiste em construir uma aplicação didática que auxilie estudantes e professores na compreensão dos principais métodos de ordenação, enquanto os objetivos específicos incluem a geração automática de relatórios descritivos, a demonstração passo a passo da execução e a apresentação de explicações textuais em português. As atividades metodológicas abrangeram a implementação dos algoritmos *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort*, a representação gráfica animada de sua execução e a coleta de métricas como tempo de execução, comparações e trocas. Os resultados experimentais confirmaram o comportamento teórico esperado, evidenciando maior eficiência dos algoritmos de complexidade $O(n \log n)$ em grandes volumes de dados e melhor desempenho do *Smooth Sort* em entradas parcialmente ordenadas. Mesmo em um cenário simulado, a avaliação técnica de desempenho demonstrou a consistência das medições e a estabilidade da ferramenta, enquanto a avaliação heurística de usabilidade indicou boa qualidade da interface e adequação ao contexto educacional, com predominância de problemas não críticos. Desenvolvida em Python com a biblioteca Pygame, a ferramenta apresenta arquitetura modular, interface intuitiva e suporte multiplataforma, constituindo um recurso eficaz para o ensino e análise de algoritmos de ordenação.

Palavras-chave: Algoritmos de ordenação. Visualização. Ensino de computação.

Abstract

Data sorting is a central topic in Computer Science, widely addressed in the teaching of data structures and algorithms. This applied work presents the development of the educational tool "Estou com Sort", designed for the visualization, interaction, and performance analysis of sorting algorithms. The main objective is to build a didactic application that assists students and educators in understanding fundamental sorting methods, while the specific objectives include the automatic generation of descriptive reports, step-by-step execution visualization, and textual explanations in Portuguese. The methodological activities included the implementation of the *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort*, and *Smooth Sort* algorithms, graphical animation of their execution, and the collection of metrics such as execution time, comparisons, and swaps. The experimental results confirmed the expected theoretical behavior, highlighting the higher efficiency of $O(n \log n)$ complexity algorithms for large datasets and the superior performance of *Smooth Sort* in partially sorted inputs. Even in a simulated scenario, the technical performance evaluation demonstrated measurement consistency and tool stability, while the heuristic usability evaluation indicated good interface quality and suitability for educational use, with predominantly non-critical issues. Developed in Python using the Pygame library, the tool features a modular architecture, intuitive interface, and cross-platform support, constituting an effective resource for teaching and analyzing sorting algorithms.

Keywords: Sorting algorithms. Visualization. Computer science education.

Sumário

1 Introdução

A ordenação de dados é uma das operações mais fundamentais na Ciência da Computação, aplicando-se desde algoritmos simples de organização até mecanismos de busca, compressão e aprendizado de máquina. Como conteúdo didático, os algoritmos de ordenação são frequentemente utilizados para introduzir estudantes ao estudo da eficiência computacional, análise de complexidade e estruturas de dados (SKIENA; REVILLA, 2003).

No entanto, o ensino tradicional desses algoritmos, baseado majoritariamente na exposição teórica e no código-fonte, pode dificultar a compreensão de seus comportamentos práticos. Essa barreira pedagógica tem motivado pesquisadores a explorar abordagens mais visuais e interativas para melhorar o aprendizado de conceitos abstratos (LIM et al., 2022; MUKASHEVA; KALKABAYEVA; PUSSYRMANOV, 2023).

Diante desse contexto, este trabalho propôs o desenvolvimento de uma ferramenta educacional denominada "Estou com Sort". O nome constitui um trocadilho com o termo em inglês *sort*, que remete ao processo de ordenação, e a expressão popular “estar com sorte”, conferindo à aplicação uma identidade lúdica e memorável. A escolha do título busca refletir simultaneamente o caráter didático da solução e sua finalidade acadêmica, alinhando criatividade e propósito pedagógico como forma de estimular o engajamento dos estudantes.

A ferramenta foi concebida para auxiliar no ensino e na compreensão de algoritmos de ordenação por meio de recursos visuais e analíticos. Ela permite acompanhar a execução dos algoritmos em tempo real por meio de animações, observar métricas quantitativas de desempenho, como tempo de execução, número de comparações e trocas, além de gerar relatórios explicativos com base nos resultados obtidos. Ademais, sua interface foi projetada com foco na usabilidade e no contexto educacional, possibilitando a experimentação prática e a análise comparativa entre diferentes algoritmos.

1.1 Problema Abordado

A compreensão dos algoritmos de ordenação é dificultada pelo caráter abstrato de suas representações tradicionais, geralmente restritas a pseudocódigos e análises matemáticas de complexidade (LANDAU; SUSAN; TOU, 2006; CORMEN et al., 2009). Embora existam ferramentas de visualização que auxiliam nesse processo, como o *VisuAlgo*, o *Algorithm Visualizer* e outras propostas acadêmicas (LOOI, 2014; PARK; KIM, 2016; GOSWAMI; GUPTA; GUPTA, 2021; MUKASHEVA; KALKABAYEVA; PUSSYRMANOV, 2023), muitas apresentam restrições, seja na quantidade de algoritmos contemplados, na avaliação de desempenho, na personalização das entradas ou na coleta de métricas quantitativas. Dessa forma, permanece um desafio a dispo-

nibilidade de recursos acessíveis e completos que apoiem o estudo de algoritmos de ordenação sob múltiplas perspectivas.

1.2 Justificativa

A relevância do tema se sustenta tanto na importância dos algoritmos de ordenação em aplicações reais quanto na sua presença marcante no ensino de estruturas de dados. Ferramentas visuais vêm sendo defendidas por estudiosos como alternativas eficazes para aumentar o engajamento e a retenção de conteúdo entre estudantes (LIM et al., 2022).

Além disso, aspectos como estabilidade, uso de memória e adaptabilidade a diferentes tipos de entrada são frequentemente discutidos de forma abstrata. Isso pode dificultar a assimilação de conceitos importantes, como a escolha do algoritmo mais adequado para uma situação específica (RIZVI; RAI; JAISWAL, 2024). Uma solução prática e didática seria permitir que o estudante observasse o comportamento dos algoritmos de forma animada, favorecendo um aprendizado mais visual e interativo (LIM et al., 2022; MUKASHEVA; KALKABAYEVA; PUSSYRMANOV, 2023).

1.3 Objetivos

Esta seção apresenta os objetivos que orientam o desenvolvimento deste trabalho, estabelecendo de forma clara o propósito geral da pesquisa e os resultados específicos alcançados. A definição dos objetivos fundamenta as decisões relacionadas ao projeto, à implementação e à avaliação da ferramenta proposta, garantindo alinhamento com a finalidade educacional e com a análise prática do desempenho de algoritmos de ordenação. Para melhor organização, os objetivos são divididos em objetivo geral e objetivos específicos.

1.3.1 Objetivo Geral

Este trabalho propõe o desenvolvimento e a validação de uma ferramenta interativa que visa aproximar a teoria da prática, utilizando recursos visuais para representar, comparar e analisar alguns algoritmos de ordenação (*Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort* com base na literatura (DIJKSTRA, 1982; ASTRACHAN, 2003; CORMEN et al., 2009; SEDGEWICK; WAYNE, 2011; KUMAR; DUTT; SAINI, 2014; BENDER; FARACH-COLTON; MOSTEIRO, 2024). A solução proposta se alinha a uma tendência crescente por metodologias de ensino mais dinâmicas, que estimulem a experimentação e o raciocínio computacional dos discentes (GOSWAMI; GUPTA; GUPTA, 2021).

1.3.2 Objetivos Específicos

Os objetivos específicos detalham as etapas e funcionalidades que foram necessárias para alcançar o objetivo geral deste trabalho, orientando o desenvolvimento, a implementação e a avaliação da ferramenta proposta. Esses objetivos contemplam tanto os aspectos técnicos relacionados à construção da aplicação quanto os aspectos educacionais voltados à visualização, análise de desempenho e apoio ao processo de ensino e aprendizagem de algoritmos de ordenação. Dessa forma, definem-se os seguintes objetivos específicos:

- Relatórios automáticos com informações técnicas e análise de complexidade (CORMEN et al., 2009; RIZVI; RAI; JAISWAL, 2024; BHAGAT et al., 2025), contribuindo para a análise objetiva do comportamento dos algoritmos e permitindo que estudantes e professores comparem seu desempenho com base em métricas formais;
- Visualização passo a passo dos algoritmos de ordenação, acompanhada de explicações textuais em português, que contribuem para a compreensão progressiva da lógica de execução e facilitam o aprendizado, mesmo para usuários com pouca familiaridade com programação;
- Execução da ordenação em modo ascendente, contribuindo para a padronização dos experimentos e permitindo a comparação consistente entre diferentes algoritmos e cenários de entrada;
- Representações gráficas animadas do funcionamento interno dos algoritmos, contribuindo para a compreensão visual de conceitos abstratos, como comparações, trocas e movimentação de elementos na estrutura de dados;
- Métricas comparativas de desempenho, incluindo tempo de execução, número de comparações, trocas e uso de memória, contribuindo para a análise empírica da eficiência dos algoritmos e para a validação prática dos conceitos teóricos estudados;
- Interface didática, intuitiva e multiplataforma (Linux e Windows), contribuindo para a acessibilidade da ferramenta e possibilitando sua utilização em diferentes ambientes educacionais, sem dependência de plataforma específica;
- Fornecimento de um recurso educacional interativo projetado para auxiliar estudantes na compreensão de algoritmos de ordenação, por meio da integração entre visualização gráfica, métricas quantitativas e explicações textuais, contribuindo potencialmente para o aprendizado desses conceitos em contextos educacionais e servindo como ferramenta de apoio ao ensino de estruturas de dados.

1.4 Atividades Metodológicas

Esta seção apresenta as atividades metodológicas realizadas para fundamentar, desenvolver e avaliar a ferramenta proposta neste trabalho. As etapas foram:

- Realizar revisão bibliográfica e benchmarking de ferramentas existentes;
- Implementar, da forma clássica, sem nenhum tipo de otimizador ou melhoria, os algoritmos *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort* com base na literatura (DIJKSTRA, 1982; ASTRACHAN, 2003; CORMEN et al., 2009; SEDGEWICK; WAYNE, 2011; KUMAR; DUTT; SAINI, 2014; BENDER; FARACH-COLTON; MOSTEIRO, 2024);
- Representar graficamente, por meio de animações, o funcionamento interno de cada algoritmo;
- Permitir simulações com diferentes tipos de entrada: ordenadas, inversamente ordenadas e aleatórias;
- Mensurar e apresentar métricas como tempo de execução, número de comparações, trocas, uso de memória e complexidade teórica (CORMEN et al., 2009; RIZVI; RAI; JAISWAL, 2024; BHAGAT et al., 2025);
- Gerar relatórios explicativos baseados nos resultados obtidos em cada simulação;
- Validar a ferramenta em cenários simulados de uso, comparando resultados teóricos e empíricos.

1.5 Organização do Trabalho

Este trabalho está estruturado em cinco capítulos, além desta introdução. O Capítulo 2 apresenta a revisão bibliográfica, abordando os principais algoritmos de ordenação, seus funcionamentos e os critérios utilizados para avaliá-los. O Capítulo 3 descreve o desenvolvimento da ferramenta proposta, incluindo o planejamento, as tecnologias utilizadas e as etapas de implementação. O Capítulo 4 apresenta os resultados obtidos, com análise do desempenho da aplicação, comparações entre algoritmos e considerações sobre a usabilidade da ferramenta. Por fim, o Capítulo 5 apresenta as considerações finais do presente trabalho, além de propostas de continuidade da pesquisa com a realização de trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo apresenta a revisão bibliográfica, englobando a fundamentação teórica e os trabalhos relacionados que embasam o desenvolvimento deste trabalho. Inicialmente, na Seção 2.1 são discutidos os conceitos fundamentais de algoritmos de ordenação, incluindo seu funcionamento, características e critérios de avaliação de desempenho. Em seguida, na Seção 2.2, são analisados trabalhos e ferramentas existentes que utilizam abordagens de visualização e análise desses algoritmos, com o objetivo de contextualizar o problema, identificar limitações das soluções atuais e justificar a proposta da ferramenta desenvolvida.

2.1 Fundamentação Teórica

Nesta seção, são apresentados os fundamentos teóricos que sustentam o desenvolvimento e a avaliação da ferramenta proposta neste trabalho. Inicialmente, na Subseção 2.1.1, discute-se o conceito de ordenação na programação e sua importância no contexto da Ciência da Computação. Em seguida, na Subseção 2.1.2, são apresentados os critérios utilizados para a avaliação de algoritmos de ordenação, incluindo análise de complexidade, uso de memória e propriedades como estabilidade e execução *in-place*, adotando-se a notação *Big-O* para expressar o crescimento assintótico do tempo e do espaço necessários (LANDAU; SUSAN; TOU, 2006; CORMEN et al., 2009). Posteriormente, nas Subseções 2.1.3 a 2.1.9, são descritos os principais algoritmos de ordenação abordados neste estudo, com destaque para seu funcionamento, características estruturais e complexidade. Por fim, nas Subseções 2.1.10 e 2.1.11, são apresentados os fundamentos relacionados à classificação de perfis de usuários e às métricas de desempenho e usabilidade adotadas, os quais subsidiam a avaliação técnica e educacional da ferramenta desenvolvida em um cenário automatizado e simulado.

2.1.1 Ordenação na Programação: Conceito e Importância

Ordenação é o processo de organizar um conjunto de elementos — como números, caracteres ou registros — seguindo uma determinada ordem, geralmente crescente ou decrescente. No contexto da ciência da computação e da programação, essa tarefa é considerada fundamental, pois serve de base para diversas operações essenciais no tratamento e manipulação de dados.

Sua importância vai muito além da simples organização estética dos dados. A ordenação é uma etapa crítica para otimizar a eficiência de vários processos computacionais. Por exemplo, algoritmos de busca mais eficientes, como a busca binária, dependem de que os dados estejam previamente ordenados para funcionarem corretamente. Além disso, procedimentos como eliminação de duplicatas, fusão de listas, compressão de dados e até mesmo algoritmos em aprendizado

de máquina e análise de grafos utilizam rotinas de ordenação como parte de seu funcionamento.

Outro aspecto relevante é o papel da ordenação no ensino de programação e algoritmos. Ela é frequentemente o primeiro tema utilizado para introduzir estudantes a conceitos essenciais como análise de complexidade (CORMEN et al., 2009), estruturas de controle, técnicas de otimização e boas práticas de codificação. Por essas razões, compreender os algoritmos de ordenação e suas características é indispensável para qualquer profissional ou estudante da área de computação. Segundo Skiena e Revilla (2003), a seguir são apresentadas algumas das principais situações em que a ordenação desempenha um papel central:

- Verificação de Unicidade: Para verificar se todos os elementos de uma coleção S são distintos, uma abordagem eficiente consiste em ordenar os elementos e, em seguida, percorrê-los sequencialmente, comparando cada elemento com seu sucessor imediato. Caso sejam encontrados dois elementos consecutivos iguais, a unicidade é violada. Essa técnica é particularmente útil por evitar comparações entre todos os pares possíveis, o que resultaria em maior complexidade;

- Remoção de Duplicatas: De modo similar à verificação de unicidade, a remoção de duplicatas pode ser feita ordenando-se o conjunto e, em seguida, percorrendo-o com dois ponteiros: um para o último elemento distinto identificado e outro para o elemento atualmente sendo examinado. Essa abordagem permite reescrever o vetor com os elementos únicos, de forma eficiente;

- Priorização de Tarefas: Em situações onde há um conjunto de tarefas com prazos ou prioridades associados, a ordenação permite reordenar essas tarefas com base nesses critérios. A ordenação, portanto, garante que as tarefas mais urgentes sejam processadas primeiro. Embora estruturas como filas de prioridade sejam mais adequadas para casos dinâmicos, a ordenação é suficiente para conjuntos fixos de tarefas;

- Busca de Mediana e Seleção de Ordem: A ordenação total de um conjunto possibilita a extração de estatísticas posicionais, como a mediana, o menor ou o maior valor, e até mesmo o k-ésimo menor ou maior elemento. Embora existam algoritmos especializados para esses problemas, a ordenação serve como solução genérica e conceitualmente simples;

- Contagem de Frequência: A identificação do elemento mais frequente de um conjunto, ou moda, também pode ser facilitada pela ordenação. Com os elementos ordenados, basta uma única varredura para contar as repetições consecutivas, identificando aquele que ocorre com maior frequência;

- Restauração da Ordem Original: Para aplicações que exigem a restauração da ordem original de dados após uma ordenação temporária, pode-se adicionar um campo auxiliar que registre a posição original de cada elemento. Esse campo pode ser utilizado posteriormente para reordenar os dados conforme a disposição original;

- Operações em Conjuntos: A ordenação facilita operações como interseção e união de conjuntos. Quando dois conjuntos estão ordenados, essas operações podem ser realizadas de forma eficiente por meio de algoritmos de mesclagem, comparando os elementos de ambas as listas em paralelo;
- Busca de Pares com Soma Alvo: Um problema clássico é determinar se existem dois elementos $x, y \in S$ tais que $x + y = z$, para um dado valor z. Após ordenar S, pode-se utilizar dois ponteiros (um no início e outro no fim do vetor) para buscar esse par de forma eficiente, reduzindo significativamente a complexidade em relação à abordagem ingênua;
- Busca Binária: Finalmente, a aplicação mais direta da ordenação é a viabilização da *busca binária*, um algoritmo fundamental para a localização eficiente de elementos em grandes conjuntos. Sem a ordenação prévia, tal método não seria aplicável.

2.1.2 Critérios para Avaliação de Algoritmos de Ordenação

A escolha de qual algoritmo de ordenação utilizar depende de uma análise criteriosa de vários fatores que influenciam diretamente sua eficiência e adequação a diferentes contextos. Para Rizvi, Rai e Jaiswal (2024), o primeiro e mais discutido critério é a complexidade de tempo, que determina a quantidade de operações realizadas conforme o tamanho da entrada aumenta. Ela pode variar de acordo com o melhor, médio ou pior caso, dependendo da organização prévia dos dados. Por exemplo, alguns algoritmos se beneficiam de entradas parcialmente ordenadas, reduzindo significativamente o número de operações necessárias, enquanto outros mantêm desempenho constante, independentemente do arranjo inicial.

Outro aspecto fundamental é a complexidade de espaço, ou seja, a quantidade de memória adicional que o algoritmo necessita além dos dados de entrada. Algoritmos que realizam a ordenação diretamente sobre o conjunto de dados, sem necessidade de estruturas auxiliares, são chamados de *in-place* e são preferíveis em cenários onde o uso eficiente da memória é crítico.

A estabilidade também é um fator importante: um algoritmo estável preserva a ordem relativa entre elementos que possuem chaves iguais. Isso é especialmente relevante quando os dados possuem múltiplos atributos e a ordenação não deve comprometer uma ordenação anterior.

Por fim, deve-se considerar ainda a facilidade de implementação e o desempenho prático. Alguns algoritmos, embora teoricamente menos eficientes, são muito mais simples de implementar e compreender, sendo adequados para fins didáticos ou para aplicações onde o volume de dados não justifica soluções mais complexas. Em contraste, algoritmos mais sofisticados podem demandar maior esforço de implementação, mas oferecem ganhos significativos de desempenho em cenários com grandes volumes de dados.

Esses critérios formam a base para a análise comparativa dos diversos algoritmos de ordenação e justificam a necessidade de ferramentas que permitam sua visualização e experimentação prática, como a que é proposta neste trabalho.

2.1.3 *Bubble Sort*

O *Bubble Sort* é um dos algoritmos de ordenação mais simples e intuitivos, sendo amplamente utilizado como introdução ao estudo de algoritmos devido à sua clareza conceitual e facilidade de compreensão. De acordo com Astrachan (2003), seu funcionamento baseia-se em múltiplas passagens sequenciais pelo conjunto de dados, nas quais pares de elementos adjacentes são comparados e trocados de posição sempre que se encontram fora da ordem desejada. Esse processo faz com que, a cada passagem completa, o maior elemento remanescente seja deslocado progressivamente para sua posição final no vetor.

Apesar de sua simplicidade, o *Bubble Sort* é considerado ineficiente para grandes conjuntos de dados, apresentando complexidade de tempo $O(n^2)$ (LANDAU; SUSAN; TOU, 2006) no pior caso, no caso médio e também no melhor caso, considerando a versão clássica do algoritmo sem mecanismos de otimização. Isso ocorre porque o algoritmo executa todas as iterações previstas pelos laços aninhados independentemente da ordem inicial dos dados, realizando o mesmo número de comparações mesmo quando o vetor já se encontra ordenado. Embora existam variações otimizadas que permitem interrupção antecipada ao detectar ausência de trocas, reduzindo o melhor caso para $O(n)$, tais otimizações não são consideradas nesta implementação, a fim de preservar a forma canônica do algoritmo conforme descrita na literatura.

Do ponto de vista estrutural, o *Bubble Sort* é classificado como um algoritmo estável e *in-place*. Sua estabilidade garante que elementos com valores iguais mantenham sua ordem relativa original após a ordenação, característica importante em aplicações onde múltiplos critérios de ordenação são utilizados. Além disso, por operar diretamente sobre o vetor original sem necessidade de estruturas auxiliares significativas, o algoritmo apresenta baixo consumo de memória adicional. Essas características, aliadas à sua simplicidade, tornam o *Bubble Sort* especialmente relevante em contextos educacionais, pois facilita a compreensão de conceitos fundamentais como comparação de elementos, trocas e controle de fluxo por meio de estruturas de repetição aninhadas.

O funcionamento detalhado do algoritmo pode ser descrito formalmente pelo pseudocódigo apresentado no Pseudocódigo 1.

Pseudocódigo 1: *Bubble Sort*

Entrada: Vetor A com n elementos
Saída: A ordenado em ordem crescente

```
1 para i ← 0 até n − 2 faça
2     para j ← 0 até n − 2 − i faça
3         se A[j] > A[j + 1] então
4             Troque A[j] com A[j + 1]
```

Fonte: Próprio autor.

No Pseudocódigo apresentado, o algoritmo utiliza duas estruturas de repetição aninhadas. O laço externo controla o número total de passagens realizadas sobre o vetor, garantindo que todos os elementos sejam posicionados corretamente ao longo das iterações. O laço interno é responsável por comparar pares de elementos adjacentes e realizar trocas sempre que necessário. Ao final de cada passagem completa do laço externo, o maior elemento ainda não ordenado é movido para sua posição definitiva no final do vetor. Esse comportamento progressivo reduz gradualmente a região do vetor que precisa ser analisada. Como não há mecanismo de interrupção antecipada nesta versão, todas as passagens são executadas integralmente, assegurando a ordenação completa do conjunto de dados conforme a definição clássica do algoritmo.

2.1.4 *Insertion Sort*

Segundo Bender, Farach-Colton e Mosteiro (2024), o *Insertion Sort* é um algoritmo de ordenação que constrói a sequência ordenada progressivamente, inserindo um elemento de cada vez na posição correta. O processo inicia-se assumindo que o primeiro elemento está ordenado e, a partir do segundo, cada elemento é comparado com os anteriores, sendo deslocado até encontrar sua posição adequada.

Este algoritmo é particularmente eficiente para conjuntos de dados pequenos ou quase ordenados, apresentando complexidade de tempo $O(n)$ (LANDAU; SUSAN; TOU, 2006) no melhor caso. No entanto, seu desempenho decai para $O(n^2)$ no caso médio e no pior caso.

O *Insertion Sort* é um algoritmo estável e *in-place*, sendo frequentemente utilizado em aplicações práticas e em algoritmos híbridos.

O funcionamento detalhado do algoritmo pode ser descrito formalmente pelo pseudocódigo apresentado no Pseudocódigo 2.

Pseudocódigo 2: *Insertion Sort*

Entrada: Vetor A com n elementos
Saída: A ordenado em ordem crescente

```
1 para i ← 1 até n − 1 faça
2     chave ← A[i];
3     j ← i − 1;
4     enquanto j ≥ 0 e A[j] > chave faça
5         A[j + 1] ← A[j];
6         j ← j − 1;
7     A[j + 1] ← chave;
```

Fonte: Próprio autor.

No Pseudocódigo apresentado, o algoritmo percorre o vetor a partir do segundo elemento, considerando o primeiro como já ordenado. A variável *chave* armazena o elemento atual, enquanto

o índice *j* percorre a parte ordenada do vetor, deslocando os elementos maiores uma posição à direita. Quando a posição correta é encontrada, a chave é inserida, expandindo a região ordenada. Esse processo é repetido até que todos os elementos estejam posicionados corretamente.

2.1.5 *Selection Sort*

De acordo com Sedgewick e Wayne (2011), o *Selection Sort* é um algoritmo que ordena os elementos selecionando repetidamente o menor elemento da parte não ordenada e o posicionando corretamente.

Sua complexidade é $O(n^2)$ em todos os casos (LANDAU; SUSAN; TOU, 2006), embora realize no máximo $O(n)$ trocas. O algoritmo é *in-place* e possui uma implementação simples, sendo útil principalmente para fins educacionais.

O funcionamento detalhado do algoritmo pode ser descrito formalmente pelo pseudocódigo apresentado no Pseudocódigo 3.

Pseudocódigo 3: *Selection Sort*

Entrada: Vetor A com n elementos
Saída: A ordenado em ordem crescente

```
1 para i ← 0 até n − 2 faça
2     min ← i;
3     para j ← i + 1 até n − 1 faça
4         se A[j] < A[min] então
5             min ← j;
6     Troque A[i] com A[min];
```

Fonte: Próprio autor.

No Pseudocódigo apresentado, o algoritmo divide o vetor em duas partes: uma ordenada e outra não ordenada. A cada iteração do laço externo, o menor elemento da parte não ordenada é encontrado e trocado com o primeiro elemento dessa região. Esse procedimento garante que a parte ordenada cresça progressivamente até que todo o vetor esteja ordenado.

2.1.6 *Merge Sort*

O *Merge Sort* é um algoritmo de ordenação baseado na estratégia de dividir para conquistar (*divide and conquer*), originalmente proposta por John von Neumann, conforme descrito por Kumar, Dutt e Saini (2014). Seu funcionamento consiste em dividir recursivamente o vetor em duas metades aproximadamente iguais até que cada subvetor contenha apenas um elemento, condição em que é considerado trivialmente ordenado. Em seguida, os subvetores são combinados por meio de um processo denominado intercalação (*merge*), no qual os elementos são

reorganizados em ordem crescente, resultando em subconjuntos progressivamente maiores e ordenados.

O Pseudocódigo 4 apresenta a estrutura geral do algoritmo, enquanto o Pseudocódigo 5 descreve o procedimento responsável pela intercalação dos subvetores ordenados.

Pseudocódigo 4: *Merge Sort*

Entrada: Vetor A, índices esq e dir
Saída: A ordenado em ordem crescente

```
1  se esq < dir então
2      meio ← ⌊(esq+dir)/2⌋;
3      MergeSort(A, esq, meio);
4      MergeSort(A, meio + 1, dir);
5      Merge(A, esq, meio, dir);
```

Fonte: Próprio autor.

Pseudocódigo 5: *Merge*

Entrada: Vetor A, índices esq, $meio$, dir
Saída: Subvetores mesclados e ordenados

```
1  L ← A[esq ... meio];
2  R ← A[meio + 1 ... dir];
3  i ← 0, j ← 0, k ← esq;
4  enquanto i < |L| e j < |R| faça
5      se L[i] ≤ R[j] então
6          A[k] ← L[i];
7          i ← i + 1;
8      senão
9          A[k] ← R[j];
10         j ← j + 1;
11     k ← k + 1;
12 enquanto i < |L| faça
13     A[k] ← L[i];
14     i ← i + 1, k ← k + 1;
15 enquanto j < |R| faça
16     A[k] ← R[j];
17     j ← j + 1, k ← k + 1;
```

Fonte: Próprio autor.

Conforme observado, o algoritmo inicia dividindo o vetor em duas partes e aplicando recursivamente o mesmo procedimento a cada uma delas. Após a ordenação das duas metades, o

procedimento de intercalação compara os elementos dos subvetores e os insere sequencialmente no vetor original em ordem crescente. Esse processo garante que, a cada nível da recursão, os subconjuntos se tornem progressivamente maiores e permaneçam ordenados até que todo o vetor esteja completamente organizado.

O *Merge Sort* apresenta complexidade de tempo $O(n \log n)$ em todos os casos — melhor, médio e pior (LANDAU; SUSAN; TOU, 2006) — pois o vetor é dividido em aproximadamente $\log n$ níveis e, em cada nível, todos os n elementos são processados durante a intercalação. Além disso, trata-se de um algoritmo estável, preservando a ordem relativa entre elementos iguais.

Como limitação, o algoritmo requer memória auxiliar adicional proporcional ao tamanho do vetor, resultando em complexidade espacial $O(n)$. Ainda assim, sua previsibilidade, estabilidade e eficiência tornam o *Merge Sort* uma escolha adequada para aplicações que exigem desempenho consistente e ordenação confiável.

2.1.7 *Quick Sort*

O *Quick Sort* é um algoritmo de ordenação baseado na estratégia de dividir para conquistar, desenvolvido por Tony Hoare conforme descrito por Cormen et al. (2009). Seu funcionamento baseia-se na escolha de um elemento denominado pivô, que é utilizado como referência para reorganizar o vetor. A partir desse pivô, os demais elementos são particionados em dois subconjuntos: aqueles com valores menores ou iguais ao pivô e aqueles com valores maiores.

O Pseudocódigo 6 apresenta o procedimento principal do algoritmo, enquanto o Pseudocódigo 7 descreve o processo de particionamento, responsável por posicionar o pivô em sua posição correta no vetor ordenado.

Pseudocódigo 6: *Quick Sort*

Entrada: Vetor A, índices esq e dir
Saída: A ordenado em ordem crescente

```
1 se esq < dir então
2     pivo ← Particiona(A, esq, dir);
3     QuickSort(A, esq, pivo − 1);
4     QuickSort(A, pivo + 1, dir);
```

Fonte: Próprio autor.

Pseudocódigo 7: Particiona

Entrada: Vetor A, índices esq e dir
Saída: Índice da posição final do pivô

```
1 pivo ← A[dir];
2 i ← esq − 1;
3 para j ← esq até dir − 1 faça
4     se A[j] ≤ pivo então
5         i ← i + 1;
6         Troque A[i] com A[j];
7 Troque A[i + 1] com A[dir];
8 retorne i + 1;
```

Fonte: Próprio autor.

Conforme apresentado nos Pseudocódigos, o algoritmo inicia realizando o particionamento do vetor, reorganizando os elementos em torno do pivô. Após essa etapa, o pivô encontra-se em sua posição definitiva, e o mesmo procedimento é aplicado recursivamente aos subconjuntos à esquerda e à direita. Esse processo continua até que todos os subconjuntos contenham apenas um elemento ou estejam vazios, resultando no vetor completamente ordenado.

O *Quick Sort* apresenta complexidade média de tempo $O(n \log n)$, sendo altamente eficiente na prática. No entanto, no pior caso, quando ocorrem particionamentos extremamente desbalanceados, sua complexidade pode atingir $O(n^2)$ (LANDAU; SUSAN; TOU, 2006). Apesar disso, estratégias adequadas de seleção do pivô reduzem significativamente a probabilidade desse cenário.

Uma de suas principais vantagens é o fato de ser um algoritmo *in-place*, necessitando apenas de uma pequena quantidade de memória auxiliar. Por outro lado, o algoritmo não é estável, podendo alterar a ordem relativa de elementos iguais.

2.1.8 *Bucket Sort*

O *Bucket Sort* é um algoritmo de ordenação baseado na distribuição dos elementos em múltiplos subconjuntos denominados baldes (*buckets*) (CORMEN et al., 2009). Cada balde representa um intervalo específico de valores, e os elementos são distribuídos nesses baldes de acordo com sua magnitude. Após a distribuição, cada balde é ordenado individualmente utilizando outro algoritmo de ordenação, e os resultados são concatenados para formar o vetor final ordenado.

O Pseudocódigo 8 apresenta as etapas principais do algoritmo, incluindo a criação dos baldes, a distribuição dos elementos e a concatenação final.

Pseudocódigo 8: *Bucket Sort*

Entrada: Vetor A com n elementos e número de baldes k

Saída: A ordenado em ordem crescente

```
para i ← 0 até k − 1 faça
    criar lista vazia B[i]
para j ← 0 até n − 1 faça
    índice ← ⌊k × A[j]⌋
    inserir A[j] em B[índice]
para i ← 0 até k − 1 faça
    ordenar B[i] usando Insertion Sort
concatenar todas as listas B[0], B[1], ..., B[k − 1] em A
```

Fonte: Próprio autor.

Conforme apresentado no Pseudocódigo, o algoritmo inicia criando uma estrutura de baldes vazios. Em seguida, cada elemento do vetor é inserido no balde correspondente com base em sua faixa de valores. Após a distribuição, cada balde é ordenado individualmente e os baldes são concatenados sequencialmente, produzindo o vetor ordenado.

A eficiência do *Bucket Sort* depende diretamente da distribuição dos dados. No caso médio, quando os elementos estão uniformemente distribuídos, o algoritmo apresenta complexidade $O(n + k)$, onde k é o número de baldes (LANDAU; SUSAN; TOU, 2006). No pior caso, quando muitos elementos são concentrados em poucos baldes, a complexidade pode degradar para $O(n^2)$.

O algoritmo requer memória adicional para armazenar os baldes, mas pode apresentar desempenho próximo ao linear quando suas condições ideais são atendidas, tornando-o eficiente para aplicações com distribuição de dados conhecida.

2.1.9 *Smooth Sort*

O *Smooth Sort* é um algoritmo de ordenação proposto por Dijkstra (1982), baseado em uma variação do *Heap Sort*. Seu diferencial está na utilização de uma estrutura denominada heap de Leonardo, que permite explorar a existência de ordenação prévia no vetor, tornando o algoritmo adaptativo.

O Pseudocódigo 9 apresenta a estrutura geral do algoritmo, incluindo a construção dos heaps e o processo de extração dos elementos em ordem.

Pseudocódigo 9: *Smooth Sort*

Entrada: Vetor A com n elementos

Saída: A ordenado em ordem crescente

construir sequência de heaps de Leonardo sobre A;
para $i \leftarrow n-1$ **até** 1 **faça**
    trocar $A[0]$ com $A[i]$;
    reduzir o tamanho da sequência de heaps;
    restaurar a propriedade de heap de Leonardo em $A[0..i-1]$;

Fonte: Próprio autor.

Conforme apresentado no Pseudocódigo, o algoritmo inicia construindo uma sequência de heaps de Leonardo a partir dos elementos do vetor. Essa estrutura mantém propriedades que permitem identificar e preservar subsequências já ordenadas. Em seguida, os elementos são removidos progressivamente do heap e posicionados corretamente no vetor, restaurando a propriedade da estrutura após cada remoção.

O *Smooth Sort* apresenta complexidade de tempo $O(n \log n)$ no pior caso e pode atingir $O(n)$ no melhor caso, quando os dados já estão ordenados (LANDAU; SUSAN; TOU, 2006). Essa característica o torna um algoritmo adaptativo, capaz de aproveitar a ordenação pré-existente.

Além disso, trata-se de um algoritmo *in-place*, exigindo apenas memória auxiliar constante. Entretanto, sua implementação é mais complexa em comparação com outros algoritmos clássicos, o que limita seu uso prático, sendo mais relevante em contextos acadêmicos e aplicações específicas.

2.1.10 Classificação dos Perfis de Usuário

Para a realização de testes em cenários simulados, foi utilizada a categorização em três perfis distintos, baseada em critérios estabelecidos na literatura de Interação Humano-Computador e usabilidade de software educacional (NIELSEN, 1993; SHNEIDERMAN et al., 2016). A definição de cada perfil considera tanto a experiência prévia com ferramentas de programação quanto a familiaridade com conceitos de algoritmos, seguindo a abordagem de (REEVES et al., 2002) para avaliação de software educacional:

- Usuário Iniciante: Caracteriza estudantes em fase inicial de aprendizado de programação, com pouca ou nenhuma experiência anterior em algoritmos de ordenação (PREECE; ROGERS; SHARP, 1994). Espera-se que apresentem:
  - Tempos de reação mais longos (0,8-1,8 segundos por ação)
  - Maior taxa de erros durante a interação (30% por ação)
  - Número reduzido de eventos, indicando hesitação na exploração da interface

  - Dificuldade em prever o comportamento do sistema

- Usuário Intermediário: Representa estudantes com conhecimento básico consolidado, possivelmente em disciplinas introdutórias de estruturas de dados (CONSTANTINE, 1994). Suas características incluem:
  - Tempos de reação moderados (0,4-1,0 segundos)
  - Ocorrência ocasional de erros (10% por ação)
  - Maior número de eventos, demonstrando confiança na navegação
  - Compreensão básica dos conceitos envolvidos
- Usuário Avançado: Corresponde a estudantes avançados ou profissionais com experiência significativa em algoritmos e interfaces similares (KELLOGG, 1987). Apresentam:
  - Tempos de reação rápidos (0,1-0,4 segundos)
  - Quase nenhum erro durante a interação (5% por ação)
  - Número elevado de eventos em curto período
  - Domínio dos conceitos e familiaridade com padrões de interface

Esta classificação tripartite é amplamente utilizada em avaliações de usabilidade por permitir capturar variações significativas no comportamento dos usuários (SAURO, 2011), além de facilitar a identificação de pontos de melhoria na interface para diferentes níveis de expertise (KRUG, 2014).

2.1.11 Métricas de Avaliação de Desempenho e Usabilidade

As métricas coletadas durante os testes foram cuidadosamente definidas para capturar aspectos essenciais da experiência do usuário:

- Sucesso (*Success*): Indica binariamente (1/0) se a operação foi concluída sem falhas críticas. Valores iguais a 1 demonstram estabilidade da aplicação.
- Tempo de Execução (*Time*): Medido em segundos, representa a responsividade da interface. Tempos menores indicam melhor desempenho e fluidez na interação.
- Eventos Enviados (*Events*): Quantidade de ações de interação do usuário simuladas durante o uso da ferramenta, correspondendo a comandos explícitos enviados à interface gráfica. Esses eventos incluem, por exemplo, cliques em botões, seleções de algoritmos, ajustes de velocidade, bem como movimentos de mouse associados a interações funcionais. Cada evento representa uma tentativa do usuário de controlar ou modificar o estado da aplicação. Perfis mais experientes tendem a realizar um maior número de eventos em intervalos

de tempo menores, refletindo maior familiaridade com a interface e maior eficiência na execução das tarefas propostas.

- Erros (*Errors*): Número de falhas não críticas ocorridas durante a interação do usuário com a interface da ferramenta. Essas falhas correspondem a ações que não comprometem o funcionamento da aplicação, mas indicam dificuldades de uso, como seleções inadequadas de opções, tentativas de executar comandos em estados inválidos ou interações que exigem correção por parte do usuário. Usuários iniciantes tendem a apresentar uma maior incidência desse tipo de erro em função da curva de aprendizado e da menor familiaridade com os elementos e fluxos da interface.
- Execuções (*Runs*): Número de repetições por cenário, garantindo significância estatística nos resultados.

2.2 Trabalhos Relacionados

Nesta seção, são apresentados e discutidos trabalhos da literatura relacionados aos algoritmos de ordenação, organizados em duas perspectivas complementares: os métodos de ordenação e as ferramentas computacionais que os implementam. Inicialmente, na Seção 2.2.1, são analisados estudos que abordam os algoritmos de ordenação de forma individual, com foco em suas características teóricas, limitações práticas e contextos de aplicação reportados em pesquisas acadêmicas. Em seguida, na Seção 2.2.2, são apresentadas ferramentas educacionais e ambientes interativos que utilizam esses algoritmos com fins didáticos, explorando recursos como visualização, análise de desempenho e interação com o usuário. Essa organização permite estabelecer uma relação entre os fundamentos teóricos dos métodos e suas aplicações em ferramentas computacionais, contribuindo para contextualizar a proposta desenvolvida neste trabalho e evidenciar suas contribuições em relação às soluções existentes.

2.2.1 Métodos

Dijkstra (1982) propôs o *Smooth Sort* como uma alternativa adaptativa e eficiente para ordenação *in-situ*, projetada para oferecer desempenho próximo a $O(n \log n)$ no pior caso, mas que se beneficia de entradas parcialmente ordenadas, podendo atingir complexidade linear. Baseado em uma estrutura de heap variante, conhecida como *heaps de Leonardo*, o algoritmo visa minimizar o número de comparações e trocas necessárias quando os dados já apresentam certa ordenação prévia. O autor argumenta que, embora sua lógica seja mais complexa que a de algoritmos como o *Heap Sort* clássico, o *Smooth Sort* é particularmente vantajoso em aplicações onde a ordenação é frequentemente aplicada sobre dados já quase ordenados, reduzindo o custo computacional de forma adaptativa. No presente trabalho, isso motivou a inclusão do Smooth Sort na ferramenta, permitindo analisar e visualizar seu comportamento adaptativo em diferentes

cenários de entrada, especialmente em conjuntos parcialmente ordenados, bem como medir o número de comparações e trocas em comparação com outros algoritmos.

Stasko et al. (1998) destacam que a visualização de algoritmos constitui uma abordagem eficaz para apoiar o ensino e a compreensão de conceitos computacionais abstratos. Segundo os autores, representações gráficas permitem observar dinamicamente o comportamento dos algoritmos, evidenciando etapas como comparações, trocas e reorganizações de dados ao longo da execução. Esse tipo de abordagem favorece a construção de modelos mentais mais precisos, auxiliando os estudantes a compreenderem não apenas o funcionamento estrutural dos algoritmos, mas também suas características de desempenho. Além disso, a utilização de recursos visuais interativos contribui para aumentar o engajamento e facilitar o aprendizado, especialmente em tópicos relacionados a algoritmos de ordenação, cuja execução envolve múltiplas etapas sequenciais e transformações progressivas dos dados. No presente trabalho, isso motivou o desenvolvimento de uma visualização animada interativa, permitindo acompanhar em tempo real as comparações, trocas e reorganizações realizadas pelos algoritmos implementados.

Astrachan (2003), em sua análise arqueológica sobre o *Bubble Sort*, revisitou a origem histórica do algoritmo e investigou criticamente sua persistente popularidade, apesar de sua ineficiência computacional. O autor identificou que, embora o *Bubble Sort* apresente uma complexidade de tempo $O(n^2)$, o que o torna inadequado para aplicações que envolvem grandes volumes de dados, ele permanece presente em materiais didáticos e cursos introdutórios de programação devido à sua simplicidade conceitual. O autor enfatizou que, sob uma perspectiva pedagógica, o *Bubble Sort* é frequentemente utilizado como exemplo de algoritmo ineficiente, servindo como ferramenta para introduzir conceitos fundamentais de algoritmos de ordenação. Contudo, o autor reforça a necessidade de os educadores priorizarem exemplos que reflitam boas práticas de desenvolvimento, evitando perpetuar o uso inadequado de algoritmos reconhecidamente ineficientes. No presente trabalho, isso motivou a inclusão do Bubble Sort como um algoritmo de referência, permitindo sua comparação com algoritmos mais eficientes e a visualização clara de suas limitações de desempenho.

Cormen et al. (2009) descreve o *QuickSort* como um dos algoritmos de ordenação mais eficientes na prática, baseado no paradigma de divisão e conquista. O algoritmo opera selecionando um elemento como pivô e particionando o conjunto de dados em dois subconjuntos, de modo que os elementos menores que o pivô sejam posicionados à esquerda e os maiores à direita, aplicando recursivamente o mesmo procedimento. Os autores destacam que a eficiência do *QuickSort* depende diretamente da escolha do pivô, uma vez que partições equilibradas conduzem à complexidade média de $O(n \log n)$, enquanto partições altamente desbalanceadas podem resultar no pior caso $O(n^2)$. Para reduzir a probabilidade desse cenário, são frequentemente empregadas estratégias como a escolha aleatória do pivô ou técnicas que buscam aproximar o pivô da mediana. Devido ao seu excelente desempenho médio, baixo consumo de memória e eficiência prática, o *QuickSort* é amplamente utilizado em aplicações reais que exigem ordenação

eficiente em memória. No presente trabalho, isso motivou a implementação do QuickSort com foco na visualização do pivô, das partições e das trocas realizadas, além da análise experimental de seu desempenho em diferentes distribuições de entrada.

Segundo Cormen et al. (2009), o *Bucket Sort* é um algoritmo de ordenação por distribuição que opera dividindo um vetor em um número finito de *buckets* (baldes), cada um dos quais é ordenado individualmente, utilizando tipicamente outro algoritmo, como o *Insertion Sort*, para, então, concatenar os resultados. A eficiência do algoritmo é altamente dependente da distribuição uniforme dos dados de entrada entre os baldes, podendo atingir um desempenho de tempo linear $O(n)$ no melhor caso. No entanto, os autores ressaltam que seu desempenho degrada para $O(n^2)$ no pior cenário, quando todos os elementos são alocados em um único balde, tornando-o sensível à distribuição dos dados e limitando sua aplicação generalizada. No presente trabalho, isso motivou a análise da influência da distribuição dos dados de entrada no desempenho dos algoritmos, destacando a importância desse fator nos experimentos realizados.

Sedgewick e Wayne (2011) apresentam o *Selection Sort* como um algoritmo de ordenação simples, baseado na seleção repetida do menor elemento da lista e sua inserção na posição correta. Os autores destacam que, independentemente da configuração inicial dos dados, o algoritmo realiza aproximadamente n^2 comparações, resultando em uma complexidade de tempo $O(n^2)$. Embora apresente baixo custo de memória por ser um algoritmo *in-place*, sua eficiência é limitada quando comparada a algoritmos mais avançados, como *QuickSort* e *Merge Sort*, especialmente em conjuntos de dados maiores. Em razão de sua simplicidade e facilidade de compreensão, o *Selection Sort* é frequentemente utilizado em contextos educacionais, embora sua aplicação prática seja restrita em cenários que exigem alto desempenho computacional. No presente trabalho, isso motivou sua inclusão como algoritmo de base para comparação experimental e visualização, permitindo analisar seu comportamento previsível em termos de comparações e trocas.

Kumar, Dutt e Saini (2014) apresentaram uma análise teórico-empírica do algoritmo *Merge Sort*, destacando sua eficiência e aplicabilidade para grandes conjuntos de dados. Os autores explicaram que o *Merge Sort* utiliza a estratégia de divisão e conquista, dividindo recursivamente a lista em sublistas menores até que cada uma contenha um único elemento, para então realizar a mesclagem ordenada. Essa abordagem confere ao algoritmo uma complexidade de tempo $O(n \log n)$, superior à dos algoritmos quadráticos como *Bubble, Selection e Insertion Sort*. A análise empírica conduzida demonstrou que o *Merge Sort* é de 24 a 241 vezes mais rápido que o *Insertion Sort*, ao lidar com vetores contendo de 10.000 a 60.000 elementos. Entretanto, os autores também destacaram como limitação a necessidade de memória adicional para armazenar temporariamente as sublistas durante o processo de mesclagem, sugerindo possíveis otimizações para reduzir esse custo. No presente trabalho, isso motivou a implementação do Merge Sort com visualização das etapas de divisão e mesclagem, bem como sua análise comparativa em relação a algoritmos quadráticos e adaptativos.

Goswami, Gupta e Gupta (2021) desenvolveram uma ferramenta interativa baseada na

web chamada *Sorting Algorithm Visualizer* com o objetivo de tornar o ensino de algoritmos de ordenação mais acessível e intuitivo por meio de animações visuais. O sistema permite que o usuário selecione entre os algoritmos *Bubble Sort, Selection Sort, Insertion Sort e Merge Sort*, visualizando sua execução em tempo real por meio de barras verticais cujo tamanho representa os valores a serem ordenados. A aplicação foi construída com *HTML5, CSS e JavaScript*, utilizando uma arquitetura baseada em Modelo-Visão-Controlador (MVC), e conta com recursos interativos como controle de velocidade, acompanhamento passo a passo e sons sincronizados com a movimentação dos elementos. Os autores destacam a importância da visualização para aprendizes visuais e defendem que compreender o funcionamento interno dos algoritmos por meio da observação gráfica é mais eficaz do que a leitura direta do código. O estudo também sugere que a adoção de tais ferramentas pode ampliar o engajamento dos estudantes e facilitar a compreensão de algoritmos mais complexos. Este estudo está diretamente relacionado com a ferramenta proposta no presente trabalho, isso motivou o desenvolvimento de uma interface interativa que permite controlar a execução dos algoritmos, ajustar parâmetros de visualização e acompanhar sua execução passo a passo.

Lim et al. (2022) propuseram uma abordagem inovadora para o ensino de algoritmos de ordenação, utilizando jogos educacionais como ferramenta pedagógica no contexto do ensino de Estruturas de Dados. O estudo desenvolveu um jogo baseado na plataforma Unity3D, permitindo aos estudantes aprender e praticar quatro algoritmos clássicos de ordenação: *Bubble Sort, Selection Sort, Insertion Sort e Quick Sort*. A metodologia incluiu dois modos: o tutorial, para a apresentação teórica e visualização animada dos algoritmos, e o modo jogo, onde os estudantes aplicam os conhecimentos adquiridos em desafios práticos, utilizando um avatar robótico para interagir com elementos virtuais. A pesquisa envolveu um experimento com dois grupos de estudantes universitários: um grupo controle que estudou por meio de métodos tradicionais e um grupo experimental que utilizou o jogo desenvolvido. Os resultados indicaram que o grupo que utilizou a aprendizagem baseada em jogos apresentou melhor desempenho e maior engajamento, completando as atividades com maior eficiência e precisão. Além disso, os autores destacam o potencial de expansão da ferramenta com a utilização de tecnologias imersivas, como a realidade virtual, para proporcionar uma experiência de aprendizagem ainda mais envolvente, o que justifica a motivação deste presente trabalho, proporcionando a adoção de elementos visuais interativos e intuitivos, visando tornar a ferramenta não apenas uma plataforma de análise experimental, mas também um recurso educacional.

Mukasheva, Kalkabayeva e Pussyrmanov (2023) investigaram o uso de ambientes de realidade virtual (VR) para o ensino de algoritmos de ordenação, propondo a aplicação *VR Sorting*, desenvolvida na plataforma *Unity3D* e compatível com o dispositivo *Oculus Quest 2*. A ferramenta permite a visualização e manipulação manual dos algoritmos *Bubble Sort e Selection Sort* em um ambiente tridimensional interativo, no qual os estudantes podem interagir diretamente com os elementos a serem ordenados. O estudo foi conduzido com 150 estudantes universitários, divididos entre grupos experimental (com uso de VR) e controle (com ensino tradicional). Os

resultados mostraram que 76,9% dos estudantes do grupo experimental alcançaram desempenho superior nas tarefas práticas de ordenação, com melhor compreensão das condições de troca e dos passos de execução dos algoritmos. A pesquisa se fundamenta na teoria da visualização construtiva, destacando a importância da imersão e da interação direta com os dados para o aprendizado significativo. Os autores concluem que a VR oferece uma abordagem inovadora e eficaz para o ensino de conceitos abstratos de programação, promovendo o engajamento, a retenção do conhecimento e o desenvolvimento de habilidades práticas, enfatizando a visualização interativa, motivando a implementação de visualizações que evidenciam claramente as interações e transformações dos dados, buscando facilitar a compreensão do funcionamento interno dos algoritmos.

Bender, Farach-Colton e Mosteiro (2024) propuseram uma significativa inovação no tradicional algoritmo de *Insertion Sort*, introduzindo o *Gapped Insertion Sort*, também conhecido como *Library Sort*. Através da inserção de lacunas estratégicas entre os elementos, os autores demonstraram que é possível reduzir a complexidade de tempo de $O(n^2)$ para $O(n \log n)$, com alta probabilidade. Essa melhoria decorre da diminuição do número de deslocamentos necessários para a inserção de novos elementos, graças à presença de espaços previamente distribuídos. A pesquisa, fundamentada em análises probabilísticas rigorosas, estabelece que tal otimização não apenas melhora o desempenho do algoritmo, mas também amplia sua aplicabilidade em sistemas que utilizam estruturas de dados para acesso rápido, como algoritmos *cache-oblivious* e para memória externa. No presente trabalho, isso motivou a análise do comportamento do Insertion Sort em diferentes cenários de entrada, especialmente em conjuntos parcialmente ordenados, permitindo avaliar sua adaptatividade e desempenho relativo.

Singh et al. (2024) propôs o desenvolvimento de uma plataforma web interativa para visualização de algoritmos de ordenação e de busca de caminhos, visando auxiliar estudantes e profissionais na compreensão e análise de tais algoritmos em tempo real. A ferramenta implementa algoritmos de ordenação como *Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort*, além de algoritmos de busca de caminho como Dijkstra, *Breadth-First Search* (BFS) e *Depth-First Search* (DFS). O visualizador permite que o usuário interaja com os parâmetros de entrada, explore a execução dos algoritmos em diferentes cenários e visualize as complexidades temporais e espaciais envolvidas. O artigo ainda realiza uma análise comparativa entre algoritmos, mostrando que o *Selection Sort* apresentou desempenho superior em tempo médio no cenário testado. Além disso, o autor apresenta uma revisão da literatura sobre ferramentas similares, como VisuAlgo e AlgoViz, destacando a importância de visualizações interativas no ensino de algoritmos. A pesquisa reforça o valor didático dessas tecnologias, apontando benefícios significativos para o aprendizado, tanto em ambientes educacionais quanto profissionais. No presente trabalho, isso motivou a inclusão de funcionalidades que permitem testar diferentes algoritmos sob variados cenários de entrada e visualizar suas características de desempenho.

Rizvi, Rai e Jaiswal (2024) realizaram uma análise crítica e comparativa do desempe-

nho de algoritmos de ordenação, considerando fatores como tempo de execução, estabilidade, adaptabilidade e uso de memória em diferentes tamanhos de entrada. O estudo aborda os algoritmos *Bubble Sort, Insertion Sort, Selection Sort, Merge Sort e Quick Sort*, classificando-os entre métodos baseados em comparação e métodos não baseados em comparação. Os autores enfatizam a importância dos algoritmos de ordenação na organização e recuperação eficiente de dados, destacando suas aplicações em áreas como comércio eletrônico, bancos de dados, motores de busca e bioinformática. A pesquisa apresenta resultados empíricos obtidos a partir de 10 execuções médias com diferentes tamanhos de dados, mostrando que algoritmos como *Quick Sort e Merge Sort* têm desempenho significativamente superior em grandes volumes de dados. O trabalho também discute como as características do conjunto de dados (como distribuição e ordem) influenciam o desempenho dos algoritmos. A conclusão destaca a importância da escolha adequada do algoritmo com base no contexto e na natureza dos dados a serem processados. No presente trabalho, isso motivou a realização de experimentos comparativos utilizando diferentes tamanhos e distribuições de dados, permitindo avaliar o desempenho relativo dos algoritmos implementados.

Bhagat et al. (2025) realizaram um extenso estudo comparativo entre sete algoritmos de ordenação — *Insertion Sort, Binary Insertion Sort, Merge Sort, Smooth Sort, Quick Sort, Tim Sort e Radix Sort* — implementados em três linguagens de programação: *Java, C++ e Python*. O estudo considerou diferentes cenários de entrada, com conjuntos de dados parcialmente ordenados e completamente desordenados, variando de 10 mil a 500 mil elementos. Para facilitar a compreensão das diferenças operacionais entre os algoritmos, os autores desenvolveram um visualizador customizado que apresenta dinamicamente o comportamento e a eficiência de cada método. Os resultados apontaram que o *Quick Sort* se destacou como o mais eficiente na maioria dos cenários, especialmente com grandes volumes de dados desordenados, enquanto o *Radix Sort* apresentou desempenho notável em casos específicos. Observou-se também que as implementações em Java superaram as demais em termos de desempenho geral, enquanto o Python apresentou limitações significativas em eficiência. A pesquisa destacou ainda que a escolha do algoritmo e da linguagem impacta diretamente na complexidade temporal e espacial das aplicações, recomendando a adoção de técnicas híbridas ou adaptativas, como o *Tim Sort*, para otimização em cenários específicos. No presente trabalho, isso motivou a implementação e avaliação experimental dos algoritmos em ambiente *Python*, permitindo analisar suas características de desempenho no contexto da ferramenta desenvolvida.

2.2.2 Ferramentas

Nesta subseção, são apresentadas ferramentas computacionais desenvolvidas com o objetivo de apoiar o ensino e a compreensão de algoritmos de ordenação por meio de recursos visuais e interativos. Essas ferramentas foram selecionadas por sua relevância acadêmica, popularidade e aplicação em contextos educacionais, permitindo analisar diferentes abordagens de visualização,

níveis de interatividade e suporte à análise de desempenho. Inicialmente, são discutidas plataformas educacionais consolidadas e amplamente utilizadas, seguidas por aplicações interativas e projetos de código aberto disponíveis publicamente. A análise dessas ferramentas permite identificar suas principais funcionalidades, limitações e contribuições, fornecendo uma base comparativa para contextualizar a proposta desenvolvida neste trabalho.

2.2.2.1 *VisuAlgo*

O *VisuAlgo* é uma plataforma educacional criada por Looi (2014) na *National University of Singapore*, em *JavaScript, HTML e CSS* (FLANAGAN, 2011; DUCKETT, 2011), com o objetivo de facilitar a aprendizagem de algoritmos e estruturas de dados por meio de animações visuais. A ferramenta cobre uma vasta gama de algoritmos, incluindo ordenação, busca em grafos, estruturas de árvore e tabelas de hash. Para seu desenvolvimento, foram usadas bibliotecas *JavaScript* como D3.js (Bostock, Mike, 2011) para animações e interatividade.

Em relação aos algoritmos de ordenação, o *VisuAlgo* permite acompanhar passo a passo a execução de métodos como *Bubble Sort, Insertion Sort, Merge Sort e Quick Sort*. No entanto, a ferramenta não exibe métricas quantitativas como tempo de execução, número de comparações ou uso de memória. Além disso, não permite a personalização de dados de entrada, limitando-se a simulações predefinidas. Seu código-fonte não é aberto, o que restringe customizações e integrações com outras plataformas.

A Figura 2.1 representa a tela inicial da ferramenta *VisuAlgo*.

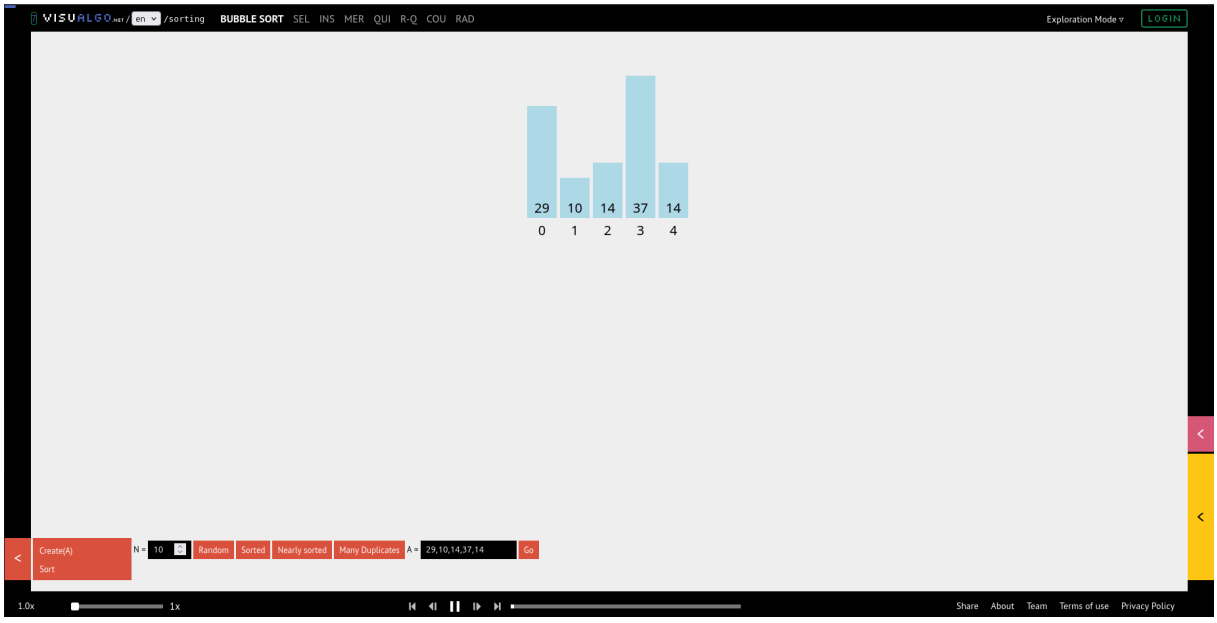

Figura 2.1 – *VisuAlgo*.
Fonte: Looi (2014).

2.2.2.2 *Algorithm Visualizer*

O *Algorithm Visualizer* é um projeto de código aberto iniciado por (PARK; KIM, 2016), em *JavaScript, HTML e CSS* (FLANAGAN, 2011; DUCKETT, 2011), com foco em permitir que os usuários visualizem o funcionamento interno de algoritmos em tempo real, ao lado da execução de seu código-fonte . Ele suporta múltiplas linguagens de programação, como *JavaScript, Python e C++* (STROUSTRUP, 1997), e abrange algoritmos de ordenação, busca, estruturas de dados, entre outros. Para seu desenvolvimento, foi usado o *React* (Facebook Inc., 2013) para a construção da interface do usuário.

Diferentemente do *VisuAlgo*, esta ferramenta permite a customização de dados de entrada e fornece uma experiência mais interativa. Os usuários podem editar o código e observar imediatamente o efeito de suas modificações na visualização. No entanto, ainda carece de indicadores numéricos detalhados (como contadores de trocas ou memória usada), e sua interface, embora poderosa, pode ser menos intuitiva para usuários iniciantes.

A Figura 2.2 representa a tela inicial da ferramenta *Algorithm Visualizer*.

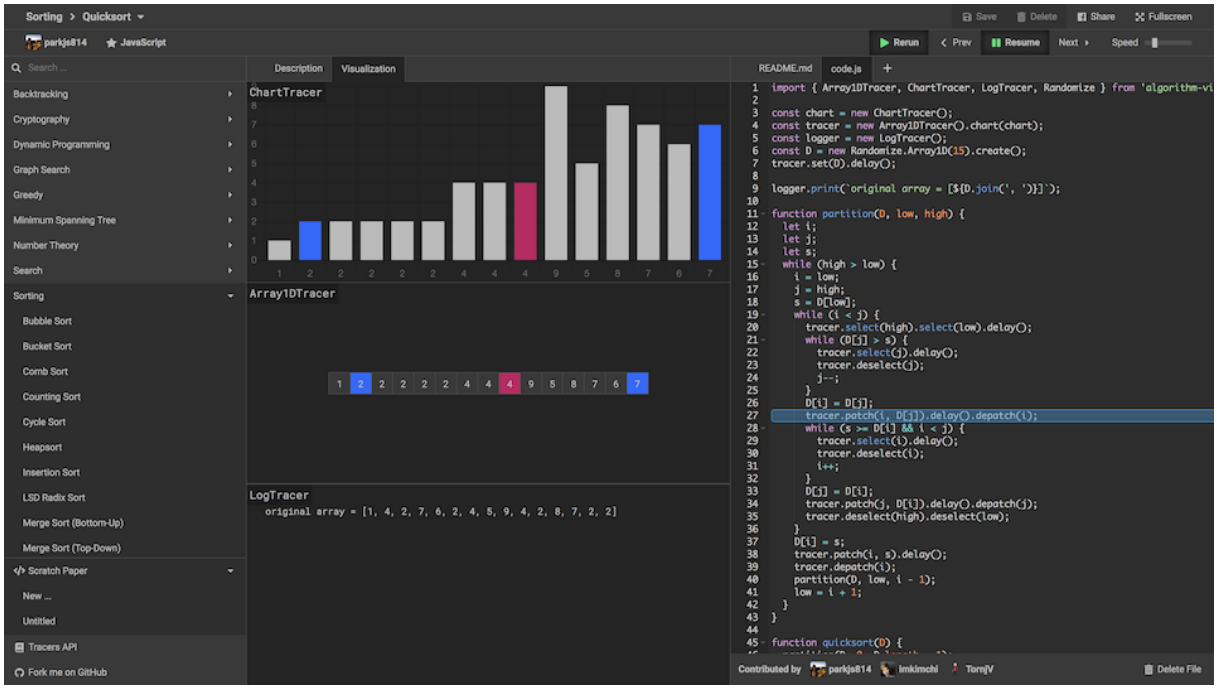

Figura 2.2 – *Algorithm Visualizer*.
Fonte: (PARK; KIM, 2016).

2.2.2.3 *Toptal Sorting Visualizer*

A *Toptal Sorting Visualizer* é uma aplicação web, feita em JavaScript, HTML e CSS (FLANAGAN, 2011; DUCKETT, 2011), focada exclusivamente em algoritmos de ordenação. Ela apresenta animações lado a lado de diferentes algoritmos, como *Bubble Sort*, *Merge Sort* e

Quick Sort, permitindo observar visualmente as diferenças de comportamento (Toptal Developers, 2019).

Sua interface é simples e responsiva, ideal para demonstrações rápidas. No entanto, não oferece personalização de entradas, nem permite acompanhar métricas de desempenho. Também não apresenta explicações didáticas sobre a lógica interna de cada algoritmo, o que a torna menos adequada para contextos educacionais mais aprofundados.

A Figura 2.3 representa a tela inicial da ferramenta *Toptal Sorting Visualizer*.

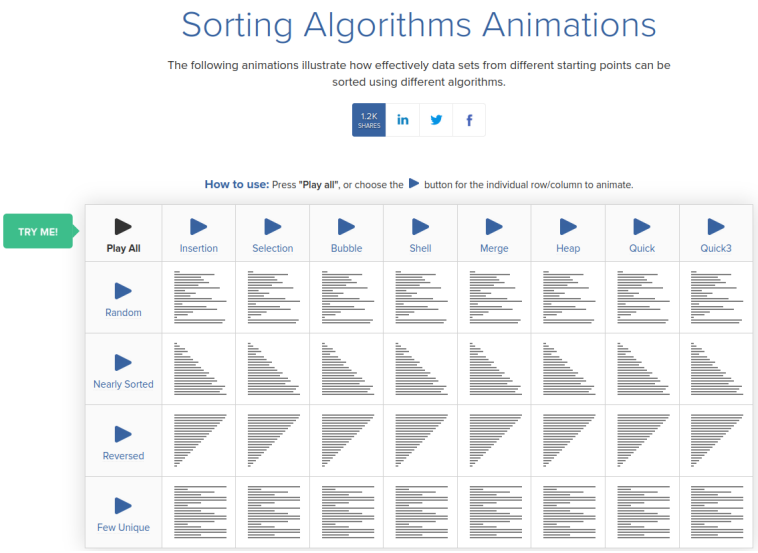

Figura 2.3 – *Toptal Sorting Visualizer*.
Fonte: (Toptal Developers, 2019).

2.2.2.4 *Sorting Visualizer* no *GitHub*

Diversos projetos no *GitHub* implementam visualizadores de algoritmos de ordenação. Um dos mais populares é o *Sorting Visualizer* , desenvolvido por (MIHAILESCU, 2019), feito em JavaScript, HTML e CSS (FLANAGAN, 2011; DUCKETT, 2011), que utiliza *React* (Facebook Inc., 2013) para criar uma interface fluida e moderna. A aplicação permite ao usuário escolher o algoritmo, controlar a velocidade da animação e visualizar ordenações com dados aleatórios.

Embora mais próxima de uma ferramenta didática completa, esse tipo de solução geralmente não possui explicações integradas, métricas quantitativas ou relatórios analíticos. Ainda assim, por ser de código aberto, oferece liberdade para modificações e serve como base para projetos acadêmicos e demonstrações técnicas.

A Figura 2.4 representa a tela inicial da ferramenta *Toptal Sorting Visualizer*.

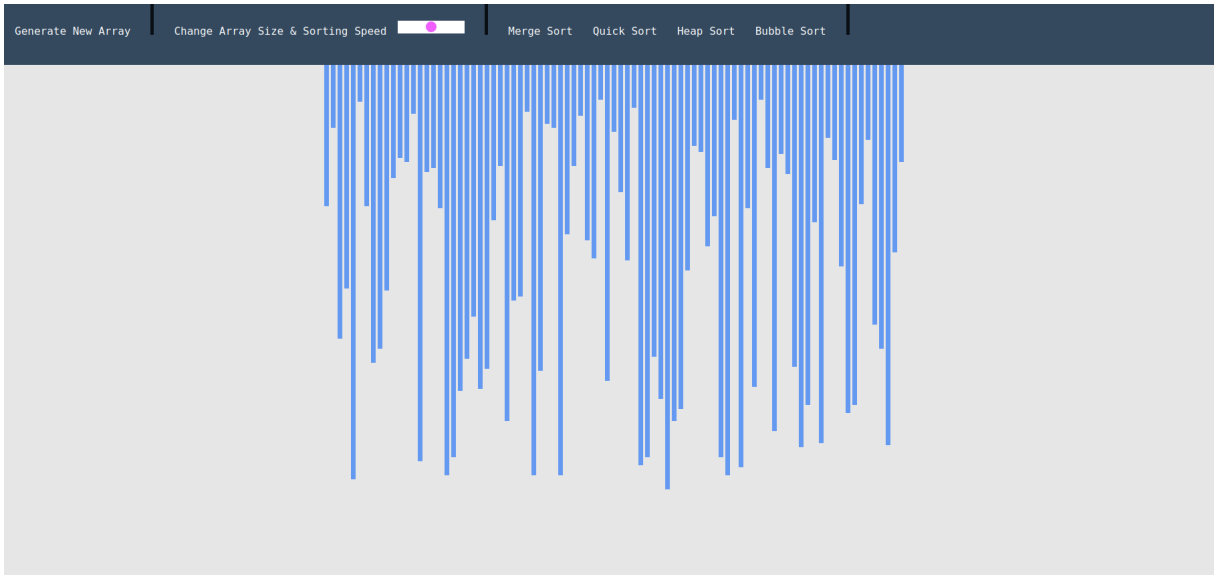

Figura 2.4 – *Sorting Visualizer* no *GitHub*.
Fonte: (MIHAILESCU, 2019).

2.3 Análise das Ferramentas Discutidas

De acordo com a Tabela 2.1, observa-se que todas as ferramentas analisadas oferecem suporte à visualização gráfica da execução de algoritmos de ordenação e incluem a implementação de métodos clássicos, o que evidencia uma convergência quanto ao uso de recursos visuais como estratégia para facilitar a compreensão desses algoritmos. Algumas ferramentas, como o *Algorithm Visualizer* e o *Sorting Visualizer* disponível no GitHub, também permitem a personalização dos dados de entrada e possuem código aberto, possibilitando maior flexibilidade e adaptação.

Entretanto, a ferramenta *Estou com Sort* apresenta diferenciais relevantes em relação às demais. Destaca-se, principalmente, o suporte à coleta e exibição de métricas quantitativas detalhadas, incluindo tempo de execução, número de comparações e trocas, uma funcionalidade ausente nas demais ferramentas analisadas. Além disso, a ferramenta oferece a geração automática de relatórios analíticos, permitindo não apenas a visualização do processo, mas também a análise técnica do desempenho dos algoritmos. Outro diferencial importante é a integração entre visualização animada, explicações textuais em português e recursos de análise, proporcionando uma abordagem mais completa e adequada ao contexto educacional.

Dessa forma, enquanto as ferramentas existentes concentram-se predominantemente na visualização do funcionamento dos algoritmos, a ferramenta proposta amplia esse escopo ao incorporar recursos de análise quantitativa e documentação automática, contribuindo de forma mais abrangente para o ensino, a experimentação e a compreensão do comportamento dos algoritmos de ordenação.

Funcionalidade	**VisuAlgo (LOOI, 2014)**	**Algorithm Visualizer (PARK; KIM, 2016)**	**Toptal Sorting Visualizer (Toptal Developers, 2019)**	**Sorting Visualizer (*Github*) (MIHAILESCU, 2019)**	**Estou com Sort (Ferramenta Proposta)**
Visualização dos dados sendo ordenados	✓	✓	✓	✓	✓
Algoritmos de ordenação clássicos	✓	✓	✓	✓	✓
Personalização de dados de entrada	✗	✓	✗	✓	✓
Código aberto	✗	✓	✗	✓	✓
Métricas quantitativas (tempo, trocas, comparações)	✗	✗	✗	✗	✓
Explicações em tempo real	✓	✓	✗	✗	✓
Relatórios analíticos	✗	✗	✗	✗	✓

Tabela 2.1 – Comparação de funcionalidades
Fonte: Próprio autor.

3 Desenvolvimento

Neste capítulo, será abordado o processo de desenvolvimento da ferramenta proposta. A Seção 3.1 contempla a metodologia seguida no trabalho, bem como as tecnologias utilizadas, a descrição experimental, as métricas de desempenho, os parâmetros de entrada e o impacto na execução. Já a seção 3.2 contempla a arquitetura do sistema, com a implementação dos algoritmos, a interface gráfica e o funcionamento geral da ferramenta.

3.1 Metodologia

Esta seção descreve a metodologia adotada no desenvolvimento deste trabalho, incluindo a caracterização do tipo de pesquisa e os procedimentos metodológicos utilizados na concepção, implementação e validação da ferramenta proposta. Inicialmente, apresenta-se o enquadramento científico da pesquisa, destacando sua natureza, abordagem e finalidade. Em seguida, são detalhadas as etapas metodológicas realizadas, que abrangem desde a revisão teórica e definição dos requisitos até o desenvolvimento, implementação e avaliação da aplicação, de modo a garantir sua consistência técnica e adequação ao contexto educacional.

3.1.1 Tipo de Pesquisa

Esta pesquisa é de natureza aplicada, pois visa ao desenvolvimento de um sistema com aplicação prática no ensino de algoritmos. A abordagem é predominantemente qualitativa, com elementos quantitativos nas análises comparativas de desempenho. Caracteriza-se também como uma pesquisa exploratória, ao propor uma representação visual interativa de algoritmos de ordenação clássicos e investigar seu comportamento sob diferentes cenários de entrada.

3.1.2 Procedimentos Metodológicos

A metodologia foi organizada nas seguintes etapas:

1. Revisão teórica e *benchmarking*: foi realizado um levantamento bibliográfico sobre os principais algoritmos de ordenação, bem como uma análise comparativa de ferramentas educacionais existentes, tais como o *VisuAlgo* (LOOI, 2014), o *Algorithm Visualizer* (PARK; KIM, 2016), o *Toptal Sorting Visualizer* (Toptal Developers, 2019) e o *Sorting Visualizer* do *GitHub* (MIHAILESCU, 2019). Essa etapa permitiu identificar os recursos mais relevantes, lacunas pedagógicas e oportunidades de inovação que fundamentaram as decisões de projeto adotadas na ferramenta proposta.

2. Definição dos requisitos: com base na revisão de literatura e no *benchmarking*, foram definidos os requisitos funcionais e não funcionais da ferramenta. Entre os principais objetivos estavam a visualização animada dos algoritmos, a coleta de métricas computacionais (tempo, trocas, comparações), a personalização de entradas e a simplicidade da interface para fins didáticos.

3. Desenvolvimento incremental da aplicação: a implementação foi realizada de forma modular. Inicialmente, foram desenvolvidos e testados os algoritmos de ordenação (*Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Bucket Sort e Smooth Sort*). Em seguida, foram implementados mecanismos para coleta automática de métricas. Por fim, a interface gráfica foi construída e integrada aos algoritmos. Neste trabalho, adota-se a ordenação em ordem crescente, seguindo a convenção amplamente utilizada no *benchmarking* das ferramentas existentes e em implementações de referência.

4. Testes e validação: a ferramenta foi testada com conjuntos de dados variados (ordenados, reversos e aleatórios), avaliando-se o desempenho dos algoritmos implementados. As métricas coletadas (tempo, comparações e trocas) são comparadas entre si e com a teoria, permitindo validar o funcionamento da aplicação tanto do ponto de vista funcional quanto pedagógico.

3.2 Arquitetura de Implementação do Sistema

A ferramenta "Estou com Sort"foi desenvolvida com o objetivo de fornecer um ambiente educacional interativo para visualização e análise de algoritmos de ordenação. A ferramenta tem como objetivo executar as ordenações para obter como resultado final sempre um vetor ordenado de forma crescente.

A estrutura do código está organizada de forma a separar responsabilidades entre arquivos distintos, favorecendo a legibilidade, a manutenção e a evolução do sistema. A Figura 3.1 ilustra o fluxo geral de funcionamento da aplicação.

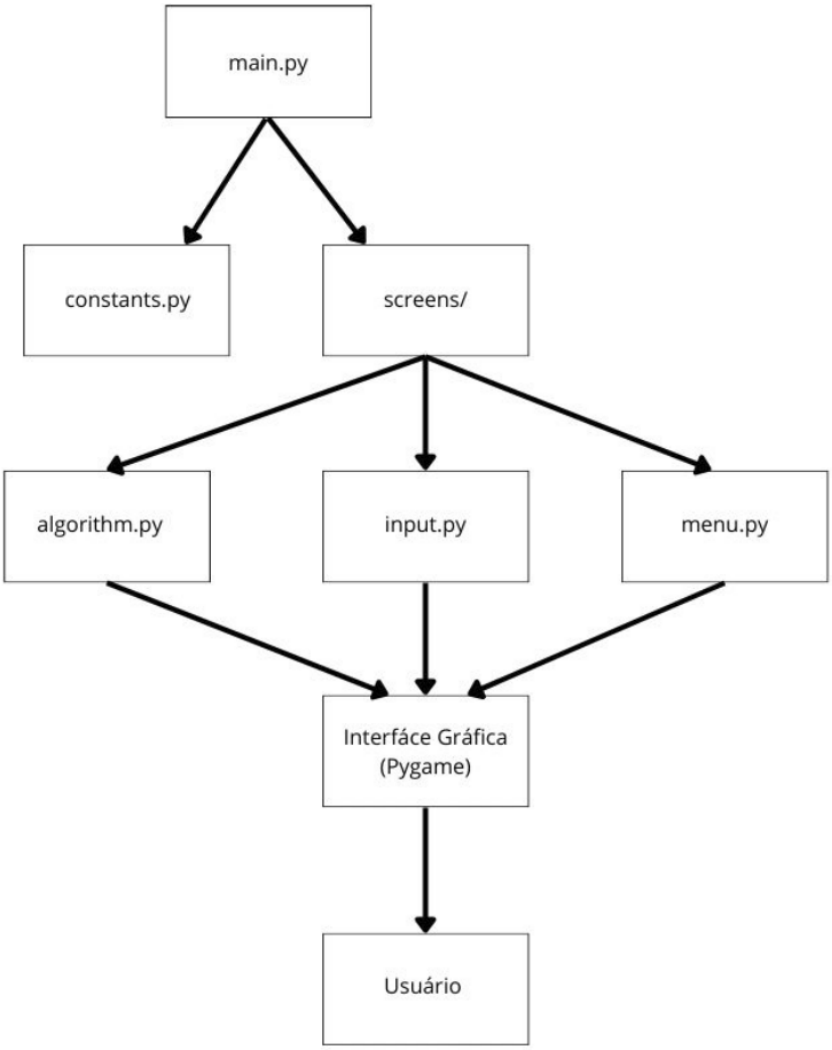

Figura 3.1 – Fluxo geral da aplicação.
Fonte: Próprio autor.

O arquivo principal, `main.py`, atua como ponto de entrada da aplicação, responsável por inicializar a janela gráfica e controlar a navegação entre diferentes telas. O diretório `screens/` concentra os módulos referentes a cada tela da interface, como `menu.py`, `algorithm.py`, `input.py`, entre outros. Há também o arquivo `constants.py`, que centraliza definições de cores, tamanhos e textos, facilitando a personalização visual do sistema.

3.2.1 Fluxo do Usuário

A Figura 3.2 apresenta o fluxo completo de interação do usuário com a ferramenta, desde o acesso inicial até a obtenção do relatório final. O processo inicia-se na tela de menu principal, onde o usuário pode optar por inserir manualmente os valores do vetor ou gerar automaticamente um conjunto de dados aleatórios. Caso escolha a inserção manual, o usuário é direcionado para uma tela específica de entrada, na qual informa individualmente os valores que comporão o vetor.

Alternativamente, ao selecionar a geração automática, o sistema solicita o tamanho desejado do vetor e realiza sua criação de forma pseudoaleatória.

Após a definição dos dados de entrada, o usuário é encaminhado à tela de seleção do algoritmo, onde pode escolher entre os métodos de ordenação disponíveis na ferramenta. Em seguida, é possível ajustar a velocidade de execução da animação, permitindo maior controle sobre a visualização do processo, especialmente em contextos educacionais, nos quais a observação detalhada das etapas é importante.

Com essas configurações definidas, a ferramenta inicia a execução do algoritmo selecionado, apresentando graficamente cada etapa do processo de ordenação, incluindo comparações e trocas entre os elementos. Durante essa fase, o usuário pode pausar e retomar a execução a qualquer momento, possibilitando uma análise mais detalhada dos estados intermediários do vetor.

Ao término da execução, quando o vetor se encontra completamente ordenado, a ferramenta gera automaticamente uma tela de relatório, contendo informações relevantes sobre o processo realizado, como o vetor ordenado, o tempo total de execução, o número de comparações e o número de trocas efetuadas. Esse fluxo estruturado permite ao usuário acompanhar todas as etapas do processo de forma clara e interativa, desde a configuração inicial até a análise final dos resultados obtidos.

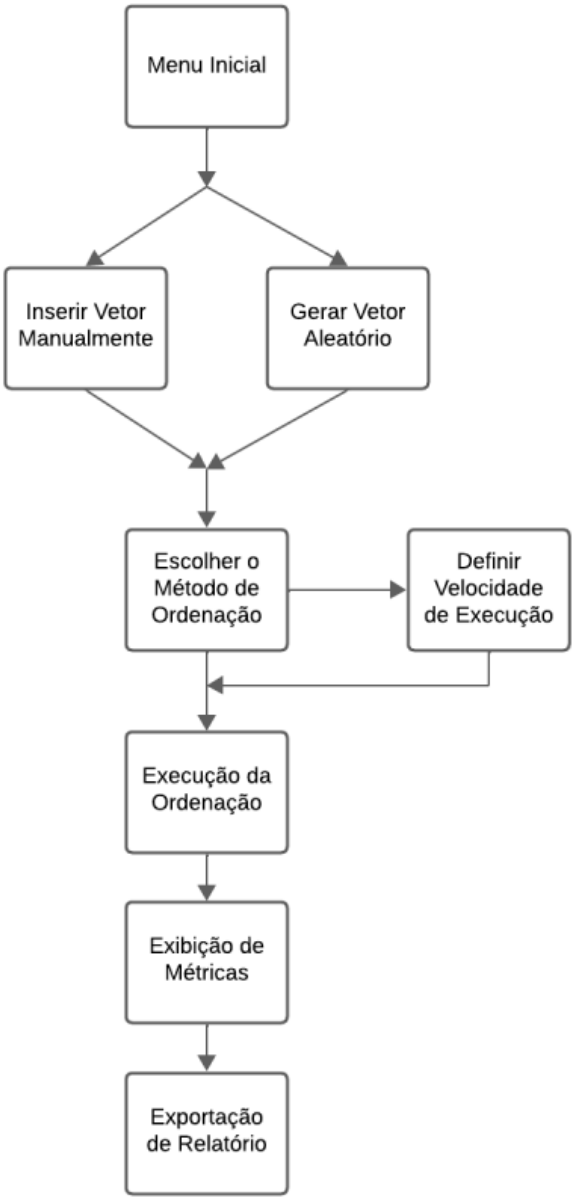

Figura 3.2 – Diagrama de Fluxo do Usuário.
Fonte: Próprio autor.

3.2.2 Tecnologias Utilizadas

As principais tecnologias utilizadas foram:

- *Python*: linguagem de programação de alto nível, amplamente adotada para fins didáticos e científicos (Python Software Foundation, 2024a).
- *Pygame*: biblioteca para criação de jogos e animações em `Python`, utilizada aqui para construir a interface gráfica e animar os algoritmos (SHINNERS, 2024).
- *GitHub*: plataforma de hospedagem de código com controle de versão distribuído, baseada no sistema Git (GitHub Inc., 2024). O projeto pode ser encontrado no repositório: (Estou

com Sort, 2025).

- *Visual Studio Code (VS Code)*: editor de código multiplataforma com suporte a extensões, utilizado como ambiente de desenvolvimento (Microsoft Corporation, 2024).
- *Graphviz*: software de código aberto para geração de diagramas e fluxogramas a partir de descrições textuais (ELLSON et al., 2024). Foi utilizado para criar representações gráficas da arquitetura do sistema e do fluxo de execução dos algoritmos, facilitando a documentação visual do projeto.
- *Matplotlib*: biblioteca de código aberto para `Python` utilizada na geração de visualizações gráficas bidimensionais (HUNTER, 2007). Foi empregada neste trabalho para criar gráficos comparativos entre os algoritmos de ordenação, permitindo analisar de forma visual métricas como tempo de execução, número de comparações e trocas.
- *Pandas*: biblioteca de código aberto para `Python` que fornece estruturas de dados de alto desempenho e ferramentas para análise de dados, utilizada para processar e organizar os resultados dos experimentos (MCKINNEY, 2010).

Além disso, é fundamental salientar que a ferramenta foi desenvolvida para desktop e apresenta caráter multiplataforma, podendo ser executada tanto em sistemas Linux quanto em Windows. Durante o processo de desenvolvimento e testes, foi utilizado um notebook *Acer Nitro V 15* com as seguintes especificações: sistema operacional Ubuntu 24.04.3 (kernel 6.14.0-27-generic), processador Intel Core i5-13450HX (13ª geração), GPU Nvidia GeForce RTX 4050, 16 GB de memória RAM e 512 GB de armazenamento em SSD.

3.2.3 Descrição Experimental

Para validar a eficácia da ferramenta, os algoritmos são executados com entradas de diferentes naturezas: ordenadas, parcialmente ordenadas, aleatórias e reversas. As execuções são cronometradas e monitoradas para coletar métricas como tempo de execução, número de comparações e número de trocas. Os dados obtidos são utilizados para comparar o desempenho dos algoritmos entre si e verificar sua aderência ao comportamento esperado na teoria.

3.2.4 Métricas de Desempenho: Tempo, Comparações e Trocas

A análise de desempenho dos algoritmos de ordenação desenvolvidos neste trabalho considera três métricas principais: tempo de execução, quantidade de comparações e quantidade de trocas.

O tempo de execução representa a duração total necessária para que o algoritmo processe o conjunto de dados, do início ao fim. Essa métrica é fundamental para avaliar a eficiência em termos práticos, especialmente quando se trabalha com entradas de tamanhos variados. No

presente sistema, a medição é realizada com o auxílio da biblioteca `time` (Python Software Foundation, 2024d), que permite cronometrar e registrar intervalos de forma precisa.

A quantidade de comparações indica quantas vezes os elementos do vetor foram comparados entre si ao longo do processo de ordenação. Essa métrica está diretamente relacionada à complexidade computacional do algoritmo e serve como parâmetro para verificar a aderência entre a implementação prática e a análise teórica.

Já a quantidade de trocas contabiliza quantas vezes dois elementos tiveram suas posições alteradas no vetor. Esse número é relevante para compreender o custo adicional de movimentação de dados, que pode impactar o desempenho, especialmente em algoritmos menos eficientes.

No contexto do presente trabalho, essas três métricas permitem não apenas validar a correção das implementações, mas também comparar objetivamente o comportamento dos diferentes métodos de ordenação em múltiplos cenários de entrada.

3.2.5 Bibliotecas para Controle e Execução

O funcionamento da aplicação depende de um conjunto de bibliotecas que desempenham papéis distintos, mas complementares, garantindo tanto a exatidão das medições quanto a qualidade da visualização.

3.2.5.1 Controle de Tempo e Animação

A biblioteca *time* (Python Software Foundation, 2024d) é utilizada para medir intervalos de tempo e introduzir pequenas pausas entre as etapas de execução, permitindo que a animação seja perceptível pelo usuário. Essa funcionalidade é essencial para fins didáticos, pois possibilita a observação detalhada do comportamento de cada algoritmo. Como exemplo da utilização dessa biblioteca, usou-se o *pygame.time.Clock()* para controlar a taxa de atualização dos quadros (*frames per second*), garantindo a execução sincronizada das animações. Esse mecanismo permite regular a velocidade de execução dos algoritmos, tornando a visualização compreensível para o usuário. Além disso, assegura consistência temporal na renderização, independentemente do desempenho do hardware utilizado.

A biblioteca *pygame* (SHINNERS, 2024) é responsável por toda a interface gráfica e animações, desenhando os elementos visuais que representam os valores do vetor e atualizando a tela a cada operação relevante. Essa abordagem transforma um conceito abstrato em uma experiência visual interativa.

A biblioteca *random* (Python Software Foundation, 2024b) é utilizada para gerar vetores iniciais em diferentes configurações — ordenados, aleatórios ou reversos — possibilitando a análise de desempenho em múltiplos cenários e tornando a aplicação mais versátil para fins experimentais.

3.2.5.2 Geração de Relatórios

A biblioteca *csv* (FOUNDATION, 2023a) oferece o módulo padrão do *Python* para leitura e escrita de arquivos no formato CSV (*Comma-Separated Values*), utilizado para armazenar e recuperar dados de experimentos.

A Biblioteca *datetime* (FOUNDATION, 2023b) possui o módulo padrão do *Python* para manipulação de datas e horas, usado no registro de timestamps dos relatórios e controle temporal das execuções.

3.2.5.3 Integração com Ferramentas Externas

A biblioteca *sys* (Python Software Foundation, 2024c) oferece funções e variáveis que interagem diretamente com o interpretador *Python*, permitindo, por exemplo, encerrar a aplicação de forma controlada ou manipular parâmetros globais de execução.

A biblioteca *subprocess* (FOUNDATION, 2023c) oferece o módulo padrão do *Python* para a criação de processos adicionais, empregado na execução de comandos externos e na integração com outras ferramentas.

3.2.6 Parâmetros de Entrada e Impacto na Execução

A aplicação foi projetada para oferecer ao usuário opções de personalização que afetam diretamente a execução e o comportamento visual dos algoritmos. Entre os parâmetros disponíveis estão:

- Tamanho do vetor: define a quantidade de elementos a serem ordenados. Vetores maiores tendem a aumentar o tempo de execução e evidenciam com mais clareza as diferenças de eficiência entre algoritmos. Em contrapartida, vetores menores permitem uma análise mais rápida e direta.
- Velocidade da animação: controla o intervalo entre cada atualização visual, ajustando o ritmo com que as operações são exibidas. Velocidades mais lentas favorecem o estudo detalhado do processo, enquanto velocidades mais rápidas aproximam a execução de um cenário real de produção.
- Tipo de ordenação inicial: o usuário pode escolher entre vetor *ordenado*, *aleatório* ou *reverso*. Essa configuração influencia diretamente o número de comparações e trocas, permitindo observar o comportamento dos algoritmos em seu melhor, pior e caso médio.

Essas opções de entrada tornam a ferramenta não apenas um recurso de visualização, mas também um ambiente experimental no qual é possível simular diferentes condições e analisar o impacto das escolhas iniciais no desempenho final do algoritmo.

3.2.7 Implementação dos Algoritmos

A implementação dos algoritmos de ordenação foi realizada com foco tanto na eficiência lógica quanto na visualização didática. Foram implementados os algoritmos clássicos: *Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort*, um algoritmo não-comparativo: *Bucket Sort* , e um algoritmo mais eficiente para grandes volumes de dados: *Smooth Sort*. A implementação de cada algoritmo e seus respectivos fluxogramas encontram-se no apêndice A.

Cada algoritmo foi adaptado para funcionar em tempo real com a biblioteca *Pygame*, de modo que, a cada iteração significativa (como uma troca ou comparação), a tela fosse atualizada, permitindo a animação da execução. As métricas computadas durante a execução, como número de comparações, trocas e tempo total, são registradas automaticamente e podem ser exportadas para análise posterior.

3.2.8 Interface Gráfica

A interface gráfica foi construída com a biblioteca *Pygame* (SHINNERS, 2024), prezando pela simplicidade e intuição para o usuário. O sistema é composto por diversas telas interativas, como:

- Tela inicial (menu principal) - Figura 3.3;
- Tela de entrada manual (Figura 3.4) ou geração automática de dados (Figura 3.5);
- Tela de escolha do algoritmo (Figura 3.6);
- Tela de escolha da velocidade de execução da ordenação (Figura 3.7);
- Tela de visualização da execução (Figuras 3.8, 3.10 e 3.11);
- Tela de relatório (Figura 3.12).

Cada tela é implementada como um módulo independente dentro do diretório `screens/`. A definição de cores contrastantes e a organização visual da interface foram orientadas por princípios básicos de usabilidade, como a visibilidade do estado do sistema e a consistência visual, conforme as heurísticas propostas por Nielsen (1993).

Figura 3.3 – Menu principal da ferramenta.
Fonte: Próprio autor.

O fluxo de interação inicia-se no menu principal, onde o usuário pode:

1. Inserir manualmente os valores do vetor;
2. Gerar automaticamente um vetor aleatório;
3. Acessar a tela "Sobre", que apresenta informações sobre a ferramenta.

Ao selecionar a opção ‘inserção manual”, o usuário é direcionado para uma tela de entrada de dados (Figura 3.4), onde informa cada elemento do vetor. O limite é fixado em no mínimo 2 números, para que haja o que ordenar, e no máximo 10 números, para que, durante o processo de ordenação, graficamente fique visível na tela a movimentação dos números. Concluída a entrada, avança para a tela de escolha do algoritmo.

Figura 3.4 – Menu de inserção manual.
Fonte: Próprio autor.

No caso da opção “gerar aleatório”, o sistema solicita o tamanho do vetor desejado (Figura 3.5). Com base nesse valor, um conjunto de números é gerado pseudoaleatoriamente utilizando a biblioteca `random` (Python Software Foundation, 2024b). Em seguida, o usuário é levado para a mesma tela de escolha do algoritmo (Figura 3.6).

Figura 3.5 – Menu de inserção aleatória.
Fonte: Próprio autor.

A tela de visualização inicia a execução animada do algoritmo escolhido. Cada comparação e troca são refletidas graficamente em tempo real, e as métricas de desempenho são registradas com o auxílio da biblioteca `time` (Python Software Foundation, 2024d). A renderização e atualização dos elementos visuais são processadas pelo `pygame` (SHINNERS, 2024), enquanto o gerenciamento de encerramento e parâmetros de execução é realizado pela biblioteca `sys` (Python Software Foundation, 2024c).

Na Figura 3.6 vê-se que a ferramenta gerou aleatoriamente 5 valores; esse tamanho de 5 foi escolhido apenas como exemplo para a demonstração da ordenação na ferramenta.

Ao selecionar a engrenagem no canto inferior direito da tela apresentada na Figura 3.6, uma tela de seleção de velocidade de execução é exibida, conforme ilustrado na Figura 3.7, permitindo que o usuário ajuste o tempo entre as atualizações visuais realizadas durante a ordenação. A ferramenta disponibiliza cinco configurações predefinidas de velocidade — muito rápido (100 ms), rápido (300 ms), normal (1100 ms), lento (1500 ms) e muito lento (2000 ms) — possibilitando uma experiência personalizada na qual o usuário pode acompanhar a execução do algoritmo de forma mais detalhada ou acelerada, de acordo com sua necessidade.

Figura 3.6 – Menu de escolha dos algoritmos.
Fonte: Próprio autor.

Figura 3.7 – Menu de escolha da velocidade de execução.
Fonte: Próprio autor.

A ferramenta conta com a demonstração e explicação passo a passo do processo de ordenação. Na Figura 3.8, é possível ver o início do processo de ordenação do *Insertion Sort*, com colunas desenhadas representando os valores e uma explicação no topo da tela sobre o que está acontecendo.

Ao clicar no botão localizado no canto inferior direito da tela, denominado *Pausar*, o usuário pode interromper a execução da ordenação a qualquer momento, congelando imediatamente a animação e possibilitando a observação detalhada do estado atual do vetor, conforme ilustrado na Figura 3.9. Ao acionar novamente o mesmo botão, agora identificado como "Continuar", a execução é retomada exatamente do ponto em que foi interrompida, garantindo maior controle sobre a análise dos estados intermediários do algoritmo. Essa funcionalidade é especialmente relevante para fins didáticos, pois permite ao estudante examinar com calma cada etapa crítica do processo de ordenação, discutir o comportamento do algoritmo e acompanhar as explicações do passo a passo sem perder o contexto da execução.

Na Figura 3.10 a ordenação está em curso, onde comparações e trocas são realizadas. Apenas um print da tela foi tirado desse processo de ordenação para que não fique repetitivo, pois o que difere as telas são as explicações de cada passo e o estado do vetor naquele momento.

Por fim, na Figura 3.11 a ordenação se encerra, pois o vetor foi ordenado de forma crescente, com as barras em ordem crescente, bem como os valores. Este foi apenas um exemplo ilustrativo com capturas de tela da ferramenta em momentos aleatórios do processo de ordenação.

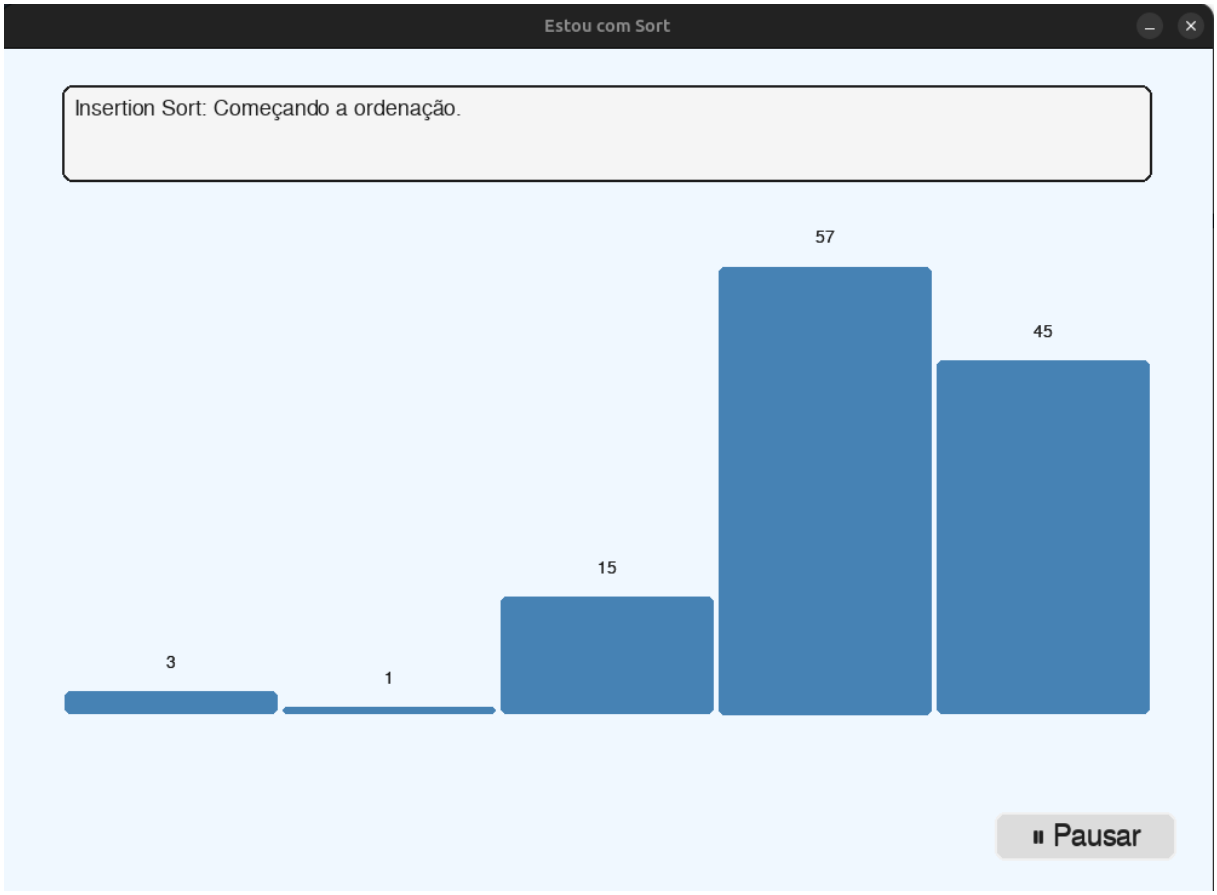

Figura 3.8 – Início da ordenação.
Fonte: Próprio autor.

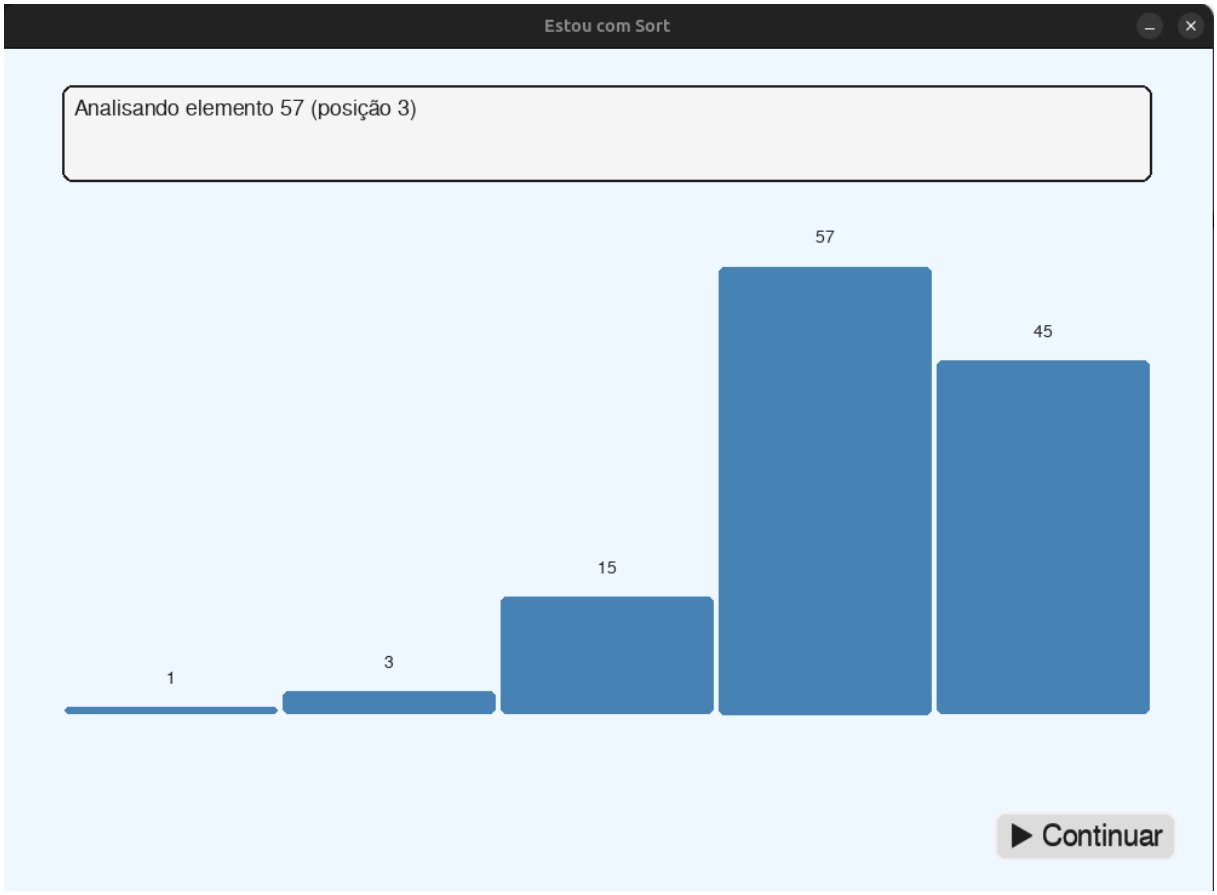

Figura 3.9 – Ordenação em estado de pausa.
Fonte: Próprio autor.

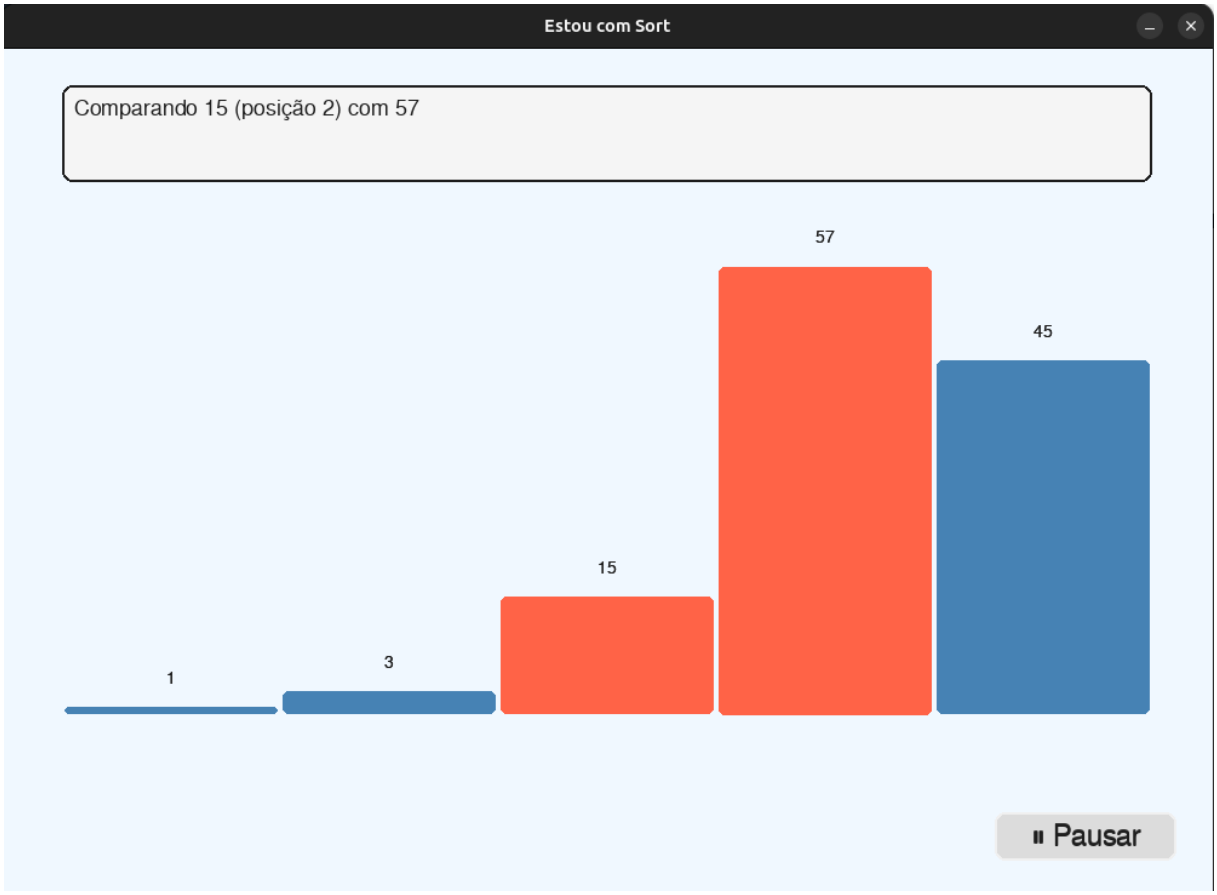

Figura 3.10 – Processo de ordenação.
Fonte: Próprio autor.

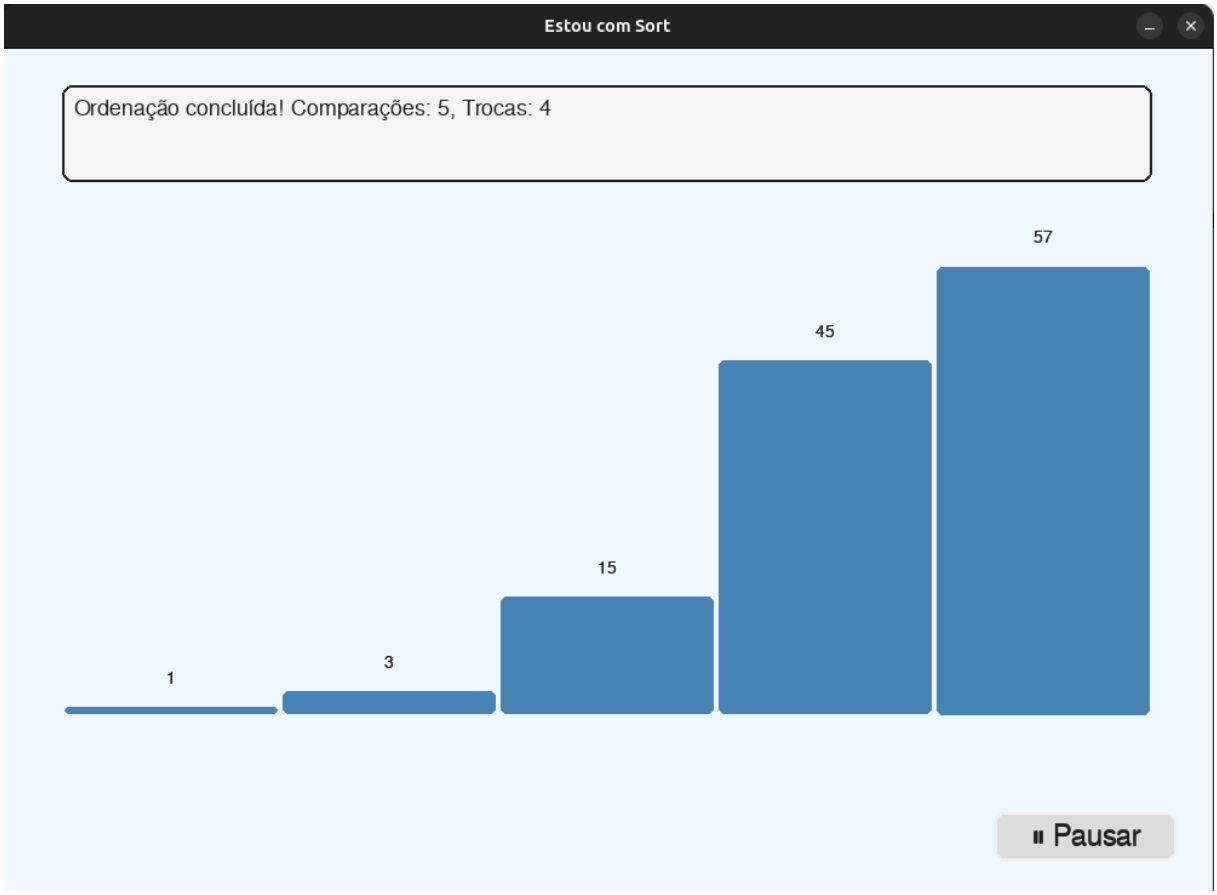

Figura 3.11 – Fim da ordenação.
Fonte: Próprio autor.

Ao término, a aplicação apresenta a tela de relatório (Figura 3.12), com um resumo das métricas obtidas: tempo total de execução, quantidade de comparações e trocas, além do vetor final ordenado.

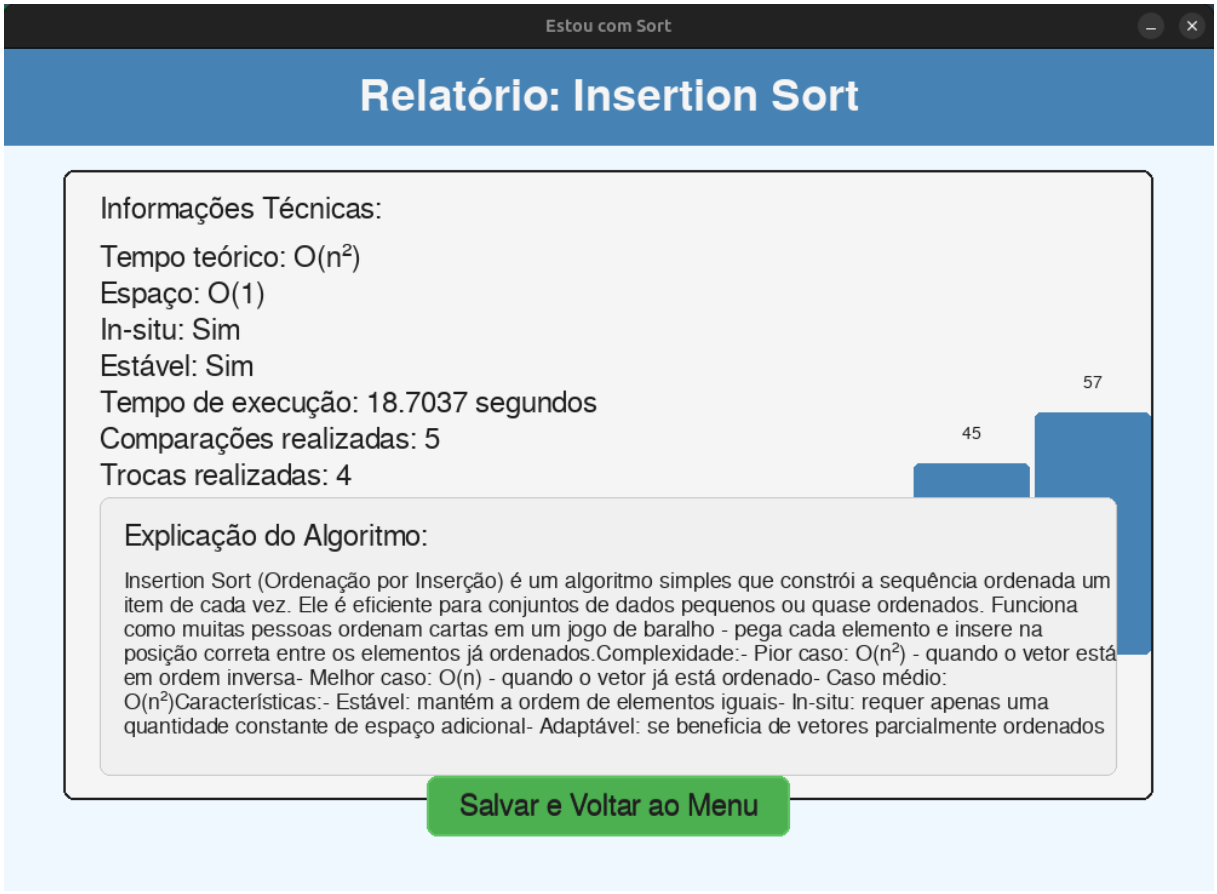

Figura 3.12 – Tela de relatório com resultados da execução.
Fonte: Próprio autor.

3.2.9 Funcionalidades da Ferramenta

As funcionalidades disponibilizadas pela ferramenta são resultado direto da arquitetura de implementação descrita nas subseções anteriores, que integra os módulos de interface gráfica, execução dos algoritmos, coleta de métricas e geração de relatórios. Cada funcionalidade corresponde à atuação coordenada desses componentes, permitindo ao usuário interagir com o sistema, configurar parâmetros de execução, acompanhar visualmente o processo de ordenação e analisar os resultados obtidos. Dessa forma, as funcionalidades representam a manifestação operacional da arquitetura da ferramenta, evidenciando como seus componentes estruturais trabalham em conjunto para atender aos objetivos educacionais e analíticos propostos neste trabalho.

- Escolha de algoritmo entre sete opções;
- Entrada de dados manual ou geração aleatória;
- Escolha da velocidade da animação;
- Possibilidade de pausar a animação;

- Exibição de métricas: tempo de execução, comparações e trocas;
- Relatório automático em formato texto (`relatorio_ordenacao.txt`);
- Navegação simples entre telas e retorno ao menu inicial.

4 Resultados

Este capítulo apresenta e discute os resultados obtidos com a execução dos algoritmos de ordenação implementados: *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort*. Foram considerados diferentes tamanhos de vetores e três cenários de ordenação (ordenado, aleatório e inverso), com o objetivo de comparar o desempenho dos métodos em termos de tempo de execução, número de comparações e número de trocas realizadas.

Além disso, o capítulo também apresenta os resultados dos testes de usabilidade e desempenho da interface gráfica da ferramenta "Estou com Sort". Esses testes tiveram como objetivo avaliar a estabilidade da aplicação, a responsividade da interface e o comportamento do sistema diante de diferentes perfis de usuário, simulando cenários realistas de interação. Para isso, foram conduzidos experimentos automatizados baseados em métricas consolidadas na literatura de Interação Humano-Computador (NIELSEN, 1993; SHNEIDERMAN et al., 2016), permitindo verificar não apenas a correção funcional da ferramenta, mas também sua adequação como recurso educacional interativo. Dessa forma, os resultados discutidos abrangem tanto aspectos da experiência de uso quanto aspectos computacionais, com resultados quantitativos e razões teóricas e práticas que explicam o comportamento observado de cada algoritmo, com base na literatura clássica e em estudos experimentais recentes (CORMEN et al., 2009; MUKASHEVA; KALKABAYEVA; PUSSYRMANOV, 2023; RIZVI; RAI; JAISWAL, 2024; BHAGAT et al., 2025), reforçando a validade da solução proposta.

4.1 Configuração Experimental dos Testes de Execução dos Algoritmos

Para os testes, a interface gráfica da ferramenta foi abstraída; dessa forma, foi possível realizar os diferentes experimentos considerando apenas a implementação dos algoritmos. Essas implementações são exatamente as mesmas que existem internamente na ferramenta. Diante disso, os testes realizados utilizaram vetores de três tamanhos distintos: $n = 1000$, $n = 1000$ e $n = 100000$. Para cada tamanho, executaram-se os algoritmos em três condições distintas de ordenação inicial:

- **Ordenado**: vetor já em ordem crescente;
- **Aleatório**: vetor com elementos dispostos de forma aleatória;
- **Inverso**: vetor em ordem decrescente.

As métricas consideradas foram:

- **Tempo de execução (s)**: medido por meio da biblioteca `time` (Python Software Foundation, 2024d);
- **Comparações**: número total de verificações entre pares de elementos;
- **Trocas**: quantidade de permutas realizadas.

Os algoritmos implementados na ferramenta e as execuções ocorreram sob as mesmas condições de hardware e software. Nos métodos de complexidade quadrática — *Bubble Sort*, *Insertion Sort* e *Selection Sort* — os testes com $n = 100000$ elementos foram desconsiderados, uma vez que o tempo de execução se mostrou inviável, conforme apontam estudos clássicos de desempenho (ASTRACHAN, 2003; SEDGEWICK; WAYNE, 2011).

4.2 Análise dos Resultados de Desempenho dos Algoritmos

Em uma etapa inicial, foram utilizados vetores com apenas cinco elementos, com o intuito de validar o correto funcionamento dos algoritmos na ferramenta e observar, de forma qualitativa, o comportamento das animações desenvolvidas. Os resultados mostraram que, para tamanhos tão reduzidos, as diferenças de desempenho são praticamente imperceptíveis, já que o custo computacional é mínimo.

Por se tratar de uma ferramenta com fins didáticos, exemplos com vetores pequenos são suficientes para demonstrar o funcionamento interno dos algoritmos, evidenciando de forma clara o *passo a passo* de cada método, as comparações realizadas e as trocas efetuadas. Essa abordagem permite visualizar o processo de ordenação antes de analisar casos mais complexos, reforçando o caráter educacional do sistema. Estratégias semelhantes são frequentemente empregadas em plataformas de ensino de algoritmos, como o *VisuAlgo* e o *Algorithm Visualizer* (LOOI, 2014; PARK; KIM, 2016), nas quais a ênfase inicial recai sobre a compreensão conceitual e não sobre o desempenho computacional.

A Tabela 4.1 apresenta os resultados obtidos para os experimentos com vetores de tamanho $n = 5$ nos três cenários considerados (ordenado, aleatório e inverso). Observa-se que os tempos de execução são praticamente desprezíveis em todos os casos, o que evidencia que, para tamanhos pequenos, as diferenças de desempenho entre os algoritmos não são significativas. Esse comportamento é esperado, pois a complexidade assintótica $O(n^2)$ dos métodos quadráticos não se torna relevante em amostras tão reduzidas. Assim, esta etapa teve caráter essencialmente ilustrativo, adequada ao propósito didático da ferramenta, servindo para demonstrar o funcionamento passo a passo dos algoritmos antes da análise com vetores maiores (LANDAU; SUSAN; TOU, 2006; CORMEN et al., 2009).

Tabela 4.1 – Resultados detalhados para vetores de tamanho $n = 5$.
Fonte: Próprio autor.

Algoritmo	**Cenário**	**Tempo (s)**	**Comparações**	**Trocas**
Bubble Sort	Ordenado	0,010	10	0
Bubble Sort	Aleatório	0,013	10	6
Bubble Sort	Inverso	0,016	10	10
Bucket Sort	Ordenado	0,001	1	0
Bucket Sort	Aleatório	0,001	4	2
Bucket Sort	Inverso	0,001	2	1
Insertion Sort	Ordenado	0,002	4	0
Insertion Sort	Aleatório	0,006	9	6
Insertion Sort	Inverso	0,010	10	10
Selection Sort	Ordenado	0,003	10	0
Selection Sort	Aleatório	0,005	10	4
Selection Sort	Inverso	0,007	10	4
Merge Sort	Ordenado	0,014	5	0
Merge Sort	Aleatório	0,021	8	6
Merge Sort	Inverso	0,025	8	6
Quick Sort	Ordenado	0,008	10	14
Quick Sort	Aleatório	0,010	6	6
Quick Sort	Inverso	0,012	6	6
Smooth Sort	Inverso	0,001	2	1
Smooth Sort	Inverso	0,001	2	0
Smooth Sort	Inverso	0,001	2	0

4.2.1 Resultados para $n = 1000$

Ampliando o conjunto de dados de entrada nos testes, os resultados para vetores de 1.000 elementos estão apresentados na Tabela 4.2. Esse tamanho intermediário já permite observar diferenças de comportamento entre os algoritmos.

Tabela 4.2 – Resultados experimentais para vetores de tamanho $n = 1000$.
Fonte: Próprio autor.

Algoritmo	**Cenário**	**Tempo (s)**	**Comparações**	**Trocas**
Bubble Sort	Ordenado	0,065	499500	0
Bubble Sort	Aleatório	0,105	499500	248336
Bubble Sort	Inverso	0,146	499500	499500
Bucket Sort	Ordenado	0,001	0	1000
Bucket Sort	Aleatório	0,002	8019	9019
Bucket Sort	Inverso	0,004	15632	16632
Insertion Sort	Ordenado	0,000	999	0
Insertion Sort	Aleatório	0,061	249326	248336
Insertion Sort	Inverso	0,139	499500	499500
Merge Sort	Ordenado	0,002	4932	4932
Merge Sort	Aleatório	0,004	8707	8707
Merge Sort	Inverso	0,002	5044	5044
Quick Sort	Ordenado	0,003	10956	6282
Quick Sort	Aleatório	0,003	10697	5894
Quick Sort	Inverso	0,003	11276	6473
Selection Sort	Ordenado	0,058	499500	0
Selection Sort	Aleatório	0,058	499500	992
Selection Sort	Inverso	0,058	499500	500
Smooth Sort	Ordenado	0,001	0	1000
Smooth Sort	Aleatório	0,001	0	1000
Smooth Sort	Inverso	0,001	0	1000

O *Bubble Sort* e o *Insertion Sort* apresentaram tempos consideravelmente maiores nos cenários aleatório e inverso. Isso ocorre porque ambos realizam comparações sucessivas em pares adjacentes e demandam múltiplas passagens sobre o vetor até que todos os elementos estejam em ordem. O comportamento quadrático ($O(n^2)$) se torna perceptível à medida que a desordem inicial aumenta (ASTRACHAN, 2003; BENDER; FARACH-COLTON; MOSTEIRO, 2024).

O *Insertion Sort*, no entanto, destacou-se no cenário ordenado, com um tempo significativamente inferior. Tal comportamento é explicado por sua natureza adaptativa: quando os elementos já estão quase em ordem, o algoritmo realiza pouquíssimas trocas, atingindo uma complexidade próxima de $O(n)$ (CORMEN et al., 2009).

O *Selection Sort*, por outro lado, manteve um desempenho praticamente constante entre os três cenários, o que se explica pela sua estrutura — sempre realiza o mesmo número de comparações, independentemente da ordem inicial, sendo pouco influenciado pela disposição dos dados (SEDGEWICK; WAYNE, 2011).

Já os algoritmos de complexidade $O(n \log n)$, *Merge Sort* e *Quick Sort*, apresentaram tempos extremamente reduzidos. O *Merge Sort* manteve a estabilidade em todos os cenários, refletindo sua natureza determinística e a ausência de pivôs. O *Quick Sort*, por sua vez, teve leve

degradação no vetor inverso, efeito conhecido e decorrente da escolha de pivô em uma posição desfavorável (CORMEN et al., 2009). Mesmo assim, ambos se mostraram substancialmente mais eficientes que os algoritmos quadráticos.

O *Bucket Sort* apresentou um desempenho notavelmente eficiente para o vetor ordenado, com tempo praticamente constante e um baixo número de comparações, resultado direto de sua natureza linear quando os dados estão uniformemente distribuídos (CORMEN et al., 2009). No cenário aleatório, manteve um tempo reduzido, pois a distribuição dos elementos entre os baldes foi equilibrada, permitindo que cada subvetor interno fosse rapidamente ordenado. Já no vetor inverso, observou-se um aumento moderado nas operações devido ao acúmulo desigual de elementos em alguns baldes, o que demanda mais ordenações internas. Ainda assim, seu desempenho permaneceu superior aos algoritmos quadráticos, confirmando a boa escalabilidade desse método.

O *Smooth Sort*, por sua vez, demonstrou tempos extremamente baixos e estáveis nos três cenários, comportamento atribuído à sua estrutura adaptativa baseada em heaps de Leonardo, que permite aproveitar sequências parcialmente ordenadas (DIJKSTRA, 1982). Mesmo em entradas inversas, o algoritmo ajusta dinamicamente as árvores durante a ordenação, mantendo complexidade próxima de $O(n \log n)$ e reduzindo o número de comparações desnecessárias. Esse comportamento reforça sua eficiência e estabilidade em diferentes tipos de entrada.

4.2.2 Resultados para $n = 10000$

Ao aumentar o tamanho do vetor para 10000 elementos (Tabela 4.3), a diferença entre as abordagens tornou-se mais acentuada. Os algoritmos quadráticos apresentaram crescimento abrupto no tempo de execução, confirmando o comportamento previsto pela análise de complexidade.

Tabela 4.3 – Resultados experimentais para vetores de tamanho $n = 10000$.
Fonte: Próprio autor.

Algoritmo	**Cenário**	**Tempo (s)**	**Comparações**	**Trocas**
Bubble Sort	Ordenado	8,030	49995000	0
Bubble Sort	Aleatório	12,129	49995000	24737382
Bubble Sort	Inverso	16,440	49995000	49995000
Bucket Sort	Ordenado	0,003	0	10000
Bucket Sort	Aleatório	0,062	249264	259264
Bucket Sort	Inverso	0,118	495000	505000
Insertion Sort	Ordenado	0,002	9999	0
Insertion Sort	Aleatório	7,706	24747375	24737382
Insertion Sort	Inverso	14,408	49995000	49995000
Merge Sort	Ordenado	0,030	64608	64608
Merge Sort	Aleatório	0,044	120487	120487
Merge Sort	Inverso	0,031	69008	69008
Quick Sort	Ordenado	0,054	156890	88395
Quick Sort	Aleatório	0,037	150685	85163
Quick Sort	Inverso	0,038	149442	81281
Selection Sort	Ordenado	7,106	49995000	0
Selection Sort	Aleatório	5,907	49995000	9994
Selection Sort	Inverso	7,093	49995000	5000
Smooth Sort	Ordenado	0,007	0	10000
Smooth Sort	Aleatório	0,007	0	10000
Smooth Sort	Inverso	0,007	0	10000

O *Bubble Sort* tornou-se claramente ineficiente, levando quase 20 segundos em vetor inverso. Isso se deve à necessidade de múltiplas passagens completas, mesmo após a maior parte dos elementos já estar posicionada (ASTRACHAN, 2003). O *Insertion Sort* manteve excelente desempenho em vetor ordenado, confirmando sua eficiência adaptativa, mas apresentou crescimento quadrático nos demais casos.

O *Selection Sort* manteve um comportamento uniforme, pois sempre percorre o vetor completo a cada iteração, independentemente da disposição inicial dos dados. Isso explica o tempo constante, embora seja relativamente alto.

Os métodos mais eficientes — *Merge Sort* e *Quick Sort* — apresentaram ganhos expressivos. O tempo de execução manteve-se inferior a 0,03 segundos, o que está em consonância com estudos empíricos que destacam a escalabilidade dessas abordagens (BHAGAT et al., 2025; RIZVI; RAI; JAISWAL, 2024). O *Quick Sort*, entretanto, voltou a mostrar leve queda de desempenho no cenário inverso, confirmando a sensibilidade à escolha do pivô, especialmente quando este é o primeiro elemento (CORMEN et al., 2009).

O *Bucket Sort* manteve um comportamento eficiente em vetores ordenados, com desempenho praticamente linear, pois cada elemento foi rapidamente alocado em seu balde correspondente

(CORMEN et al., 2009). Entretanto, no vetor inverso, a distribuição tornou-se desigual, concentrando muitos elementos nos mesmos baldes, o que elevou o número de comparações e trocas internas. Apesar disso, o tempo total de execução permaneceu baixo, evidenciando a escalabilidade do método para volumes maiores de dados.

Já o *Smooth Sort* apresentou desempenho constante e estável independentemente do cenário inicial. Essa característica se deve à sua capacidade de reestruturar o heap de Leonardo para se adaptar à organização parcial dos elementos (DIJKSTRA, 1982). Assim, mesmo para grandes vetores, o algoritmo evita degradações significativas de desempenho, combinando eficiência assintótica e adaptabilidade prática.

4.2.3 Resultados para $n = 100000$

Devido à inviabilidade computacional, apenas *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort* foram testados com vetores de 100000 elementos. Os resultados estão dispostos na Tabela 4.4.

Tabela 4.4 – Resultados experimentais para vetores de tamanho $n = 100000$.
Fonte: Próprio autor.

Algoritmo	**Cenário**	**Tempo (s)**	**Comparações**	**Trocas**
Bucket Sort	Ordenado	0,039	0	100000
Bucket Sort	Aleatório	1,920	7861320	7961320
Bucket Sort	Inverso	4,104	15772824	15872824
Merge Sort	Ordenado	0,394	815024	815024
Merge Sort	Aleatório	0,598	1536427	1536427
Merge Sort	Inverso	0,391	853904	853904
Quick Sort	Ordenado	0,504	2175527	1214824
Quick Sort	Aleatório	0,546	2122549	1175248
Quick Sort	Inverso	0,470	2001072	1099670
Smooth Sort	Ordenado	0,066	0	100000
Smooth Sort	Aleatório	0,066	0	100000
Smooth Sort	Inverso	0,071	0	100000

Ambos os algoritmos mantiveram um desempenho altamente eficiente, com tempos inferiores a meio segundo. O *Quick Sort* superou o *Merge Sort* nos casos ordenado e aleatório, confirmando seu potencial quando a escolha do pivô é adequada. Em contrapartida, no vetor inverso, o tempo aumentou significativamente, um comportamento amplamente documentado na literatura (CORMEN et al., 2009).

O *Merge Sort*, embora ligeiramente mais lento, manteve a estabilidade entre os três cenários, graças à sua estrutura recursiva determinística, que divide o problema de forma equilibrada, independentemente da entrada. Contudo, seu consumo de memória é maior, uma vez que o

algoritmo necessita de vetores auxiliares para a etapa de intercalação (KUMAR; DUTT; SAINI, 2014).

O *Bucket Sort* apresentou um desempenho muito competitivo para o vetor ordenado, reforçando sua eficiência quase linear em dados uniformemente distribuídos (CORMEN et al., 2009). Nos cenários aleatório e inverso, entretanto, a distribuição desigual dos elementos entre os baldes aumentou a carga de processamento interno, elevando o número de comparações e trocas. Esse comportamento está alinhado à teoria, que aponta a perda de eficiência quando os dados não seguem uma distribuição uniforme. Mesmo assim, o método manteve um desempenho globalmente favorável em comparação com os algoritmos tradicionais de mesma ordem de complexidade.

O *Smooth Sort* novamente demonstrou uma estabilidade notável, apresentando tempos praticamente constantes entre os três cenários (DIJKSTRA, 1982). Essa consistência se deve à estrutura do algoritmo, que permite reorganizar eficientemente os heaps sem depender da disposição inicial dos dados. Além disso, seu consumo de memória é reduzido em relação ao *Merge Sort*, o que o torna uma alternativa interessante para ordenações de grande porte com limitação de recursos.

4.2.4 Síntese Final

Os resultados obtidos permitem chegar a algumas conclusões importantes:

- Os algoritmos de complexidade $O(n^2)$ — *Bubble Sort*, *Insertion Sort* e *Selection Sort* — são adequados apenas para pequenos volumes de dados. O *Insertion Sort*, contudo, mantém relevância prática em cenários onde os dados estão parcialmente ordenados.
- O *Merge Sort* mostrou-se o mais estável entre todos os algoritmos testados, apresentando tempos consistentes e comportamento previsível em qualquer tipo de entrada.
- O *Quick Sort* demonstrou ser o mais rápido em geral, especialmente em vetores aleatórios, embora apresente degradação em casos desfavoráveis de pivô.

De forma complementar, o *Bucket Sort* e o *Smooth Sort* confirmaram desempenhos condizentes com suas naturezas distintas. O primeiro mostrou-se altamente eficiente em dados uniformemente distribuídos, mas sensível a distribuições assimétricas, enquanto o segundo manteve desempenho estável mesmo em grandes volumes e entradas desordenadas, aproveitando sua estrutura adaptativa de *heaps* (DIJKSTRA, 1982; CORMEN et al., 2009).

Essas conclusões estão alinhadas com a teoria clássica de análise de algoritmos (LANDAU; SUSAN; TOU, 2006; CORMEN et al., 2009) e com estudos contemporâneos sobre a avaliação empírica de desempenho (RIZVI; RAI; JAISWAL, 2024; SINGH et al., 2024; BHAGAT et al., 2025). Os experimentos realizados validam, portanto, não apenas a implementação

desenvolvida, mas também os princípios fundamentais de eficiência e comportamento assintótico dos algoritmos de ordenação.

4.3 Configuração Experimental dos Testes de Desempenho e Usabilidade

Para validar a usabilidade e a estabilidade da ferramenta “Estou com Sort”, foram realizados testes automatizados que simularam diferentes cenários de interação do usuário com a interface gráfica. Estes testes visaram verificar o correto funcionamento dos componentes visuais, a responsividade da aplicação e a coleta adequada de métricas de desempenho.

É importante destacar que os testes realizados neste estudo foram baseados em simulações automatizadas e não envolveram diretamente usuários humanos reais. Embora os parâmetros adotados tenham sido fundamentados em modelos consolidados de usabilidade (NIELSEN, 1993; SAURO, 2011; SHNEIDERMAN et al., 2016), a simulação não captura integralmente fatores humanos complexos, como percepção subjetiva, carga cognitiva e estratégias individuais de interação. Dessa forma, a realização de estudos empíricos com usuários reais constitui uma etapa futura importante para a validação completa da usabilidade da ferramenta, permitindo avaliar aspectos qualitativos da experiência do usuário, como facilidade de uso, satisfação e eficácia pedagógica, conforme recomendado na literatura de avaliação de sistemas interativos (REEVES et al., 2002; KRUG, 2014).

4.3.1 Testes de Desempenho com Diferentes Perfis

Foram simulados três perfis de usuário com diferentes padrões de interação através do script `user_simulation.py`. Cada perfil executou cinco testes consecutivos para garantir robustez estatística. Os parâmetros comportamentais foram calibrados com base em estudos de usabilidade para representar realisticamente cada categoria de usuário.

Os testes foram realizados por meio de simulação automatizada de interações com a interface gráfica, utilizando scripts desenvolvidos especificamente para este fim. O script `user_simulation.py` gera eventos sintéticos equivalentes às ações de um usuário real, incluindo cliques, seleções de algoritmos e interações com controles da interface. Esses eventos são enviados diretamente ao sistema de eventos da biblioteca *Pygame*, permitindo reproduzir o comportamento típico de interação humano-computador de forma controlada e reprodutível.

Os parâmetros comportamentais adotados na simulação, como tempo de reação entre ações, taxa de erro e número de eventos, foram definidos com base em modelos amplamente utilizados na literatura de usabilidade, como os princípios de Nielsen (NIELSEN, 1993) e as diretrizes de interação humano-computador descritas por Shneiderman (SHNEIDERMAN et al., 2016). Esses modelos estabelecem que usuários iniciantes apresentam maior tempo de reação

e maior incidência de erros, enquanto usuários experientes realizam interações mais rápidas e eficientes, justificando os intervalos temporais e taxas de erro adotados na simulação.

Durante a execução dos testes, a própria aplicação foi responsável por coletar automaticamente as métricas de desempenho, incluindo tempo de execução, número de eventos processados, número de erros detectados e taxa de sucesso das operações. Essas métricas foram registradas diretamente pelo sistema durante a execução da interface gráfica, garantindo precisão e consistência na coleta dos dados experimentais, sem intervenção manual ou viés observacional.

4.3.2 Testes de Funcionalidade Básica

Os resultados apresentados na Tabela 4.5 demonstram o comportamento da aplicação em cenários fundamentais de inicialização e interação básica. A coluna Sucesso indica se o cenário foi executado corretamente, sendo representada pelo valor 1 em todos os casos, o que confirma que a aplicação respondeu conforme o esperado em todas as situações testadas. A coluna Tempo Médio (s) representa o tempo necessário para a execução completa de cada cenário, incluindo a inicialização da interface e o processamento dos eventos simulados, observando-se valores próximos de 2 segundos, o que indica um tempo de resposta consistente e adequado para uma aplicação educacional interativa.

A coluna Eventos refere-se à quantidade de eventos processados durante a execução de cada cenário, como a inicialização da janela, a renderização da interface e a interação simulada do usuário. Já a coluna Execuções indica o número de vezes que cada cenário foi executado durante os testes automatizados, garantindo a validação do comportamento da aplicação em condições controladas.

De modo geral, os resultados confirmam que a ferramenta apresenta comportamento estável durante sua inicialização, execução e encerramento, sem falhas ou interrupções inesperadas. Esse resultado evidencia a confiabilidade da estrutura básica da aplicação, demonstrando que os mecanismos de inicialização da interface gráfica, processamento de eventos e encerramento do sistema estão funcionando corretamente. Essa validação é fundamental para garantir que as funcionalidades mais avançadas da ferramenta possam ser executadas sobre uma base estável e consistente.

Tabela 4.5 – Resultados dos testes de funcionalidade básica

Cenário de Teste	**Sucesso**	**Tempo Médio (s)**	**Eventos**	**Execuções**
Abrir app e clicar no menu inicial	1	2,24	2	1
Abrir app e aguardar 3 segundos	1	2,001	2	1
Abrir app e fechar rapidamente	1	2,001	2	1

Fonte: Próprio autor

4.3.3 Análise dos Resultados

Os resultados demonstram que a ferramenta apresenta estabilidade consistente em todos os cenários testados, com taxa de sucesso de 100% nas operações básicas. A análise da Tabela 4.6 revela padrões claros de desempenho entre os diferentes perfis de usuário, validando a adequação da classificação adotada:

- Usuários avançados: apresentaram os melhores tempos de resposta (média de 1,202 s), com execuções mais eficientes, menor variação temporal e nenhum erro registrado, demonstrando domínio da interface e compreensão sólida dos conceitos;
- Usuários intermediários: mantiveram desempenho consistente (média de 3,236 s) com baixa ocorrência de erros (0,4 em média), indicando boa adaptação à ferramenta e familiaridade com os padrões de interação;
- Usuários iniciantes: apresentaram maior variabilidade nos tempos de execução (média de 4,527 s) e maior incidência de erros (1,4 em média), refletindo a curva de aprendizado esperada para usuários não familiarizados com ferramentas de visualização de algoritmos.

A Tabela 4.6 apresenta um resumo consolidado dos resultados por perfil, enquanto a Tabela 4.7 mostra os dados completos coletados durante as execuções.

Tabela 4.6 – Resumo dos resultados por perfil de usuário

Perfil	**Sucesso**	**Tempo Médio (s)**	**Eventos Médios**	**Erros Médios**	**Execuções**
Iniciante	1	4,527	2,8	1,4	5
Intermediário	1	3,236	4,4	0,4	5
Avançado	1	1,202	4,8	0,0	5

Fonte: Próprio autor

Tabela 4.7 – Resultados detalhados dos testes com diferentes perfis de usuário

Perfil	Sucesso	Tempo (s)	Eventos	Erros	Execução
iniciante	1	3,639	2	2	1
iniciante	1	3,439	2	1	2
iniciante	1	2,139	2	0	3
iniciante	1	6,644	4	2	4
iniciante	1	6,776	4	2	5
intermediario	1	2,454	4	0	1
intermediario	1	3,011	4	1	2
intermediario	1	4,615	5	1	3
intermediario	1	3,617	5	0	4
intermediario	1	2,484	4	0	5
avancado	1	1,218	5	0	1
avancado	1	1,345	6	0	2
avancado	1	0,675	4	0	3
avancado	1	1,425	5	0	4
avancado	1	1,348	4	0	5

Fonte: Próprio autor

A progressão observada nos eventos enviados (de 2,8 para 4,8 em média) corrobora a premissa de que usuários mais experientes realizam mais ações em menos tempo, demonstrando maior engajamento e confiança na exploração da interface. A ferramenta demonstrou robustez mesmo em cenários com múltiplos eventos e interações sequenciais, mantendo a responsividade da interface para todos os perfis.

Os tempos de carregamento inicial permaneceram consistentes em aproximadamente 2,0 segundos, indicando uma inicialização estável da aplicação. A ausência de falhas críticas em todos os testes valida a qualidade da implementação da interface gráfica e sua adequação para uso educacional por usuários com diferentes níveis de experiência em programação e algoritmos, atendendo assim ao objetivo de ser uma ferramenta inclusiva e acessível.

5 Considerações Finais

Após o desenvolvimento da ferramenta e a análise dos resultados obtidos, torna-se possível reunir as principais percepções e aprendizados alcançados ao longo do estudo. Este capítulo apresenta a conclusão do trabalho, sintetizando as observações mais relevantes extraídas do processo de implementação e avaliação da ferramenta proposta.

5.1 Conclusão

O desenvolvimento da ferramenta possibilitou a visualização animada e a comparação de diferentes algoritmos de ordenação, buscando aproximar a teoria da prática por meio de uma abordagem visual, intuitiva e interativa. Para isso, foram implementados os algoritmos *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, *Bucket Sort* e *Smooth Sort*, integrando recursos gráficos e métricas quantitativas que permitiram analisar, em tempo real, seu comportamento diante de distintos cenários de entrada, conforme defendem Stasko et al. (1998) e Goswami, Gupta e Gupta (2021) quanto à importância da visualização no ensino de algoritmos.

A análise dos resultados confirmou que a aplicação cumpre seu objetivo pedagógico ao possibilitar a compreensão detalhada de cada etapa do processo de ordenação, bem como a observação de métricas como tempo de execução, número de comparações e trocas realizadas. Essa abordagem facilitou a identificação, de forma prática, das diferenças de desempenho entre algoritmos de complexidade quadrática e algoritmos mais eficientes, de ordem $O(n \log n)$, validando conceitos discutidos por Rizvi, Rai e Jaiswal (2024) e Bhagat et al. (2025) sobre desempenho comparativo.

Ao longo do desenvolvimento, verificou-se que a inclusão de elementos interativos, como a seleção do tipo de entrada (ordenada, aleatória ou inversa) e a geração automática de relatórios, agregou valor à ferramenta, tornando-a versátil e de fácil utilização. Esses recursos ampliam o potencial de aplicação tanto em contextos educacionais quanto em ambientes de estudo individual, promovendo um aprendizado mais ativo e engajador, como sugerido por Lim et al. (2022) e Mukasheva, Kalkabayeva e Pussyrmanov (2023).

O estudo também demonstrou que a integração entre teoria e prática, mediada por recursos visuais, favorece a fixação de conceitos e estimula o raciocínio lógico dos estudantes, além de facilitar a comparação direta entre algoritmos. Tal constatação reforça a relevância de soluções que aliam clareza didática, usabilidade e rigor técnico no ensino de Ciência da Computação (SKIENA; REVILLA, 2003).

Por fim, ressalta-se que uma versão inicial da ferramenta interativa para o ensino de algoritmos de ordenação foi desenvolvida e testada.

5.2 Trabalhos Futuros

Os resultados obtidos neste trabalho demonstram o potencial da ferramenta desenvolvida como recurso educacional para o ensino e análise de algoritmos de ordenação. Entretanto, diversas possibilidades de expansão podem ser exploradas com o objetivo de ampliar sua aplicabilidade, robustez e impacto pedagógico.

5.2.1 Evolução para Arquitetura Web Multiplataforma

Um dos principais eixos de evolução consiste na adaptação da ferramenta para uma arquitetura web multiplataforma. Atualmente, a aplicação opera como um software desktop desenvolvido em *Python* com a biblioteca *Pygame*, o que limita sua acessibilidade a ambientes onde o software está instalado. A migração para uma arquitetura baseada em tecnologias web permitirá o acesso por meio de navegadores, mantendo sua independência do sistema operacional, ampliando significativamente o alcance da ferramenta e facilitando sua adoção em ambientes educacionais formais e remotos.

5.2.2 Expansão do Conjunto de Algoritmos

Outro eixo importante refere-se à ampliação do conjunto de algoritmos suportados. A inclusão de algoritmos adicionais, como *TimSort, Radix Sort e Counting Sort*, permitirá análises comparativas mais abrangentes, incluindo métodos não baseados em comparação. Esses algoritmos possuem características distintas em termos de complexidade, estabilidade e aplicabilidade, contribuindo para uma compreensão mais completa dos diferentes paradigmas de ordenação.

5.2.3 Implementação de Algoritmos Híbridos e Adaptativos

Além disso, pretende-se implementar algoritmos híbridos e adaptativos, que combinam diferentes estratégias de ordenação de acordo com as características dos dados de entrada. Esses algoritmos são amplamente utilizados em sistemas reais devido à sua eficiência prática, como é o caso do próprio *TimSort*, que combina características do *Merge Sort* e do *Insertion Sort*. A inclusão desses métodos permitirá análises mais próximas de cenários reais de aplicação.

5.2.4 Aprimoramento da Análise Experimental e Estatística

Outro aspecto relevante envolve o aprimoramento dos mecanismos de análise experimental e estatística da ferramenta. A incorporação de métricas adicionais e métodos estatísticos mais robustos permitirá análises mais precisas e fundamentadas, incluindo a avaliação de variabilidade, consistência e desempenho em diferentes cenários de entrada. Isso tornará a ferramenta mais adequada não apenas para fins didáticos, mas também para experimentação acadêmica.

5.2.5 Expansão para Visualização de Algoritmos de Grafos

A expansão do escopo da ferramenta para incluir a visualização de algoritmos de grafos também representa uma direção promissora. A implementação de algoritmos como *Breadth-First Search* (BFS), *Depth-First Search* (DFS) e *Dijkstra* permitirá que a ferramenta seja utilizada no ensino de estruturas de dados mais avançadas, ampliando sua aplicabilidade para disciplinas além de algoritmos de ordenação.

5.2.6 Avaliação Empírica com Usuários

Outro passo fundamental consiste na realização de avaliações empíricas com estudantes reais, com o objetivo de validar a eficácia pedagógica da ferramenta em contextos educacionais reais. Esses estudos permitirão avaliar aspectos como facilidade de uso, compreensão dos algoritmos e impacto no processo de aprendizagem, fornecendo evidências mais concretas sobre sua utilidade como ferramenta educacional.

5.2.7 Disponibilização como Recurso Institucional

Por fim, pretende-se disponibilizar a ferramenta como recurso institucional ou projeto de extensão acadêmica, permitindo sua utilização por estudantes, professores e pesquisadores. Essa iniciativa poderá contribuir para a disseminação do ensino de algoritmos de forma mais interativa e acessível, além de possibilitar futuras contribuições e melhorias por parte da comunidade acadêmica.

Referências

ASTRACHAN, O. *Bubble Sort: An Archaeological Algorithmic Analysis*. 2003. Technical Report, Duke University. Acesso em 26 Mai. 2025. Disponível em: <https://users.cs.duke.edu/~ola/papers/bubble.pdf>.

BENDER, M. A.; FARACH-COLTON, M.; MOSTEIRO, M. A. *Insertion Sort is O(n log n)*. 2024. ArXiv preprint. Disponível em: <https://arxiv.org/abs/cs/0407003>.

BHAGAT, K.; DAS, A. K.; AGRAHARI, S. K.; SHAH, S. A.; T, D. R.; RAMASAMY, G. *Cross-Language Comparative Study and Performance Benchmarking of Sorting Algorithms*. 2025. SSRN Electronic Journal. Available online: <https://ssrn.com/abstract=5088751> [Acesso em 26 Mai. 2025. Disponível em: <https://ssrn.com/abstract=5088751>.

Bostock, Mike. *D3.js: Data-Driven Documents*. [S.l.], 2011. Acesso em: 19 ago. 2025. Disponível em: <https://d3js.org/>.

CONSTANTINE, L. L. The usage-centered software engineering. *IEEE Software*, IEEE, v. 11, n. 2, p. 34–41, 1994. Disponível em: <https://www.academia.edu/20977072/Usage_centered_software_engineering_an_agile_approach_to_integrating_users_user_interfaces_and_usability_into_software_engineering_practice>.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd. ed. Cambridge, MA: MIT Press, 2009. Obra clássica de algoritmos, com explicações detalhadas sobre análise de complexidade e notação Big-O. ISBN 9780262033848. Disponível em: <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.

DIJKSTRA, E. W. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, v. 1, n. 3, p. 223–233, 1982. Proposto por Dijkstra como versão adaptativa do Heap Sort. Disponível em: <https://www.sciencedirect.com/science/article/pii/0167642382900168>.

DUCKETT, J. *HTML and CSS: Design and Build Websites*. Indianapolis, IN: Wiley, 2011. Disponível em: <https://www.wiley.com/en-us/HTML+and+CSS%3A+Design+and+Build+Websites-p-9781118008188>.

ELLSON, J.; GANSNER, E. R.; NORTH, S. C.; KOUTSOFIOS, E.; WOODHULL, G. *Graphviz - Graph Visualization Software*. 2024. <https://graphviz.org/>. Acesso em: 27 jul. 2025.

Estou com Sort. *Repositório GitHub do projeto*. 2025. <https://github.com/LeandroZanetti30/Estou-com-Sort>.

Facebook Inc. *React – A JavaScript Library for Building User Interfaces*. [S.l.], 2013. Acesso em: 19 ago. 2025. Disponível em: <https://reactjs.org/>.

FLANAGAN, D. *JavaScript: The Definitive Guide*. 6th. ed. Sebastopol, CA: O'Reilly Media, 2011. Disponível em: <https://www.oreilly.com/library/view/javascript-the-definitive/9781449393854/>.

FOUNDATION, P. S. *csv — CSV File Reading and Writing*. 2023. Disponível em: <https://docs.python.org/3/library/csv.html>.

FOUNDATION, P. S. *datetime — Basic date and time types*. 2023. Disponível em: <https://docs.python.org/3/library/datetime.html>.

FOUNDATION, P. S. *subprocess — Subprocess management*. 2023. Disponível em: <https://docs.python.org/3/library/subprocess.html>.

GitHub Inc. *GitHub: Where the world builds software*. 2024. <https://github.com/>. Acesso em: 27 jul. 2025.

GOSWAMI, A. D. B.; GUPTA, A.; GUPTA, A. Algorithm visualizer: Enhancing learning of algorithms. *International Research Journal of Modernization in Engineering, Technology and Science*, v. 3, p. 461–465, 2021. ISSN 2582-5208. Disponível em: <https://www.studocu.com/in/document/jawaharlal-nehru-college/msc-computer-science/irjmets-653443/62320125>.

HUNTER, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, IEEE, v. 9, n. 3, p. 90–95, 2007. Disponível em: <https://ieeexplore.ieee.org/document/4160265>.

KELLOGG, W. Conceptual consistency in the user interface: effects on user performance. In: . [s.n.], 1987. p. 389–394. Disponível em: <https://www.sciencedirect.com/science/chapter/edited-volume/abs/pii/B9780444703040500686>.

KRUG, S. *Don't Make Me Think: A Common Sense Approach to Web Usability*. Indianapolis, IN: New Riders, 2014. ISBN 978-0-321-96551-6. Disponível em: <https://dn790002.ca.archive.org/0/items/SteveKrugDontMakeMeThink/Steve_Krug_Don%E2%80%99t_Make_Me_Think%2C.pdf>.

KUMAR, A.; DUTT, A.; SAINI, G. Merge sort algorithm. *International Journal of Research*, v. 1, n. 11, p. 16–21, 2014. ISSN 2348-6848. Disponível em: <https://journals.pen2print.org/index.php/ijr/article/download/1071/1015>.

LANDAU; SUSAN; TOU, E. Big-o notation. *Notices of the American Mathematical Society*, v. 53, n. 9, p. 1019–1020, 2006. Artigo introdutório que explica a notação Big-O e sua aplicação na análise de algoritmos. Disponível em: <https://old.maa.org/press/periodicals/convergence/math-origins-orders-of-growth>.

LIM, W. H.; CAI, Y.; YAO, D.; CAO, Q. Visualize and learn sorting algorithms in data structure subject in a game-based learning. In: *IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. IEEE, 2022. p. 437–442. Disponível em: <https://doi.org/10.1109/ISMAR-Adjunct57072.2022.00083>.

LOOI, K. W. *VisuAlgo: Visualising Data Structures and Algorithms through Animation*. 2014. Online. Disponível em: <https://visualgo.net/en>. Acesso em: 27 jul. 2025.

MCKINNEY, W. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, v. 445, p. 51–56, 2010. Disponível em: <https://pandas.pydata.org/about/citing.html>.

Microsoft Corporation. *Visual Studio Code*. 2024. <https://code.visualstudio.com/>. Acesso em: 27 jul. 2025.

MIHAILESCU, C. *Sorting Visualizer*. 2019. Repositório GitHub. Disponível em: <https://github.com/clementmihailescu/Sorting-Visualizer>. Acesso em: 27 jul. 2025.

MUKASHEVA, M.; KALKABAYEVA, Z.; PUSSYRMANOV, N. Visualization of sorting algorithms in the virtual reality environment. *Frontiers in Education*, v. 8, p. 1195200, 2023. Disponível em: <https://doi.org/10.3389/feduc.2023.1195200>.

NIELSEN, J. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1993. ISBN 0-12-518406-9. Disponível em: <https://www.sciencedirect.com/book/monograph/9780125184069/usability-engineering>.

PARK, W.; KIM, J. *Algorithm Visualizer*. 2016. Projeto de código aberto. Disponível em: <https://algorithm-visualizer.org/>. Acesso em: 27 jul. 2025.

Paulo Freire. *Pedagogia da Autonomia: saberes necessários à prática educativa*. 1996. <https://nepegeo.paginas.ufsc.br/files/2018/11/Pedagogia-da-Autonomia-Paulo-Freire.pdf>. Acesso em: 18 ago. 2025.

PREECE, J.; ROGERS, Y.; SHARP, H. Human-computer interaction. *Addison-Wesley*, 1994. Disponível em: <https://archive.org/details/humancomputerint0000pree>.

Python Software Foundation. *Python Programming Language*. 2024. <https://www.python.org/>. Acesso em: 27 jul. 2025.

Python Software Foundation. *random — Generate pseudo-random numbers*. 2024. <https://docs.python.org/3/library/random.html>. Acesso em: 27 jul. 2025.

Python Software Foundation. *sys — System-specific parameters and functions*. 2024. <https://docs.python.org/3/library/sys.html>. Acesso em: 27 jul. 2025.

Python Software Foundation. *time — Time access and conversions*. 2024. <https://docs.python.org/3/library/time.html>. Acesso em: 27 jul. 2025.

REEVES, T. C.; BENSON, L.; ELLIOTT, D.; GRANT, M.; HOLSCHUH, D.; KIM, B.; KIM, H.; LAUBER, E.; LOH, C. S. Usability and instructional design heuristics for e-learning evaluation. In: *Proceedings of ED-MEDIA 2002: World Conference on Educational Multimedia, Hypermedia and Telecommunications*. Association for the Advancement of Computing in Education, 2002. Disponível em: <https://files.eric.ed.gov/fulltext/ED477084.pdf>.

RIZVI, Q. M.; RAI, H.; JAISWAL, R. Sorting algorithms in focus: A critical examination of sorting algorithm performance. In: *Conference on Sorting Algorithms*. [s.n.], 2024. Available at ResearchGate. Disponível em: <https://www.researchgate.net/publication/378962637>.

SAURO, J. Iso standards and enterprise software: A case study using sumi and sus in an international sale: Design, user experience, and usability. *Springer Nature*, 2011. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-642-21675-6_21>.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Addison-Wesley Professional, 2011. Acesso em 26 Mai. 2025. ISBN 9780321573513. Disponível em: <https://algs4.cs.princeton.edu/home/>.

SHINNERS, P. *Pygame - Python Game Development*. 2024. <https://www.pygame.org/>. Acesso em: 27 jul. 2025.

SHNEIDERMAN, B.; PLAISANT, C.; COHEN, M.; JACOBS, S.; ELMQVIST, N. Designing the user interface: strategies for effective human-computer interaction. *Pearson*, 2016. Disponível em: <https://www.researchgate.net/publication/240953629_Designing_the_user_interface_strategies_for_effective_human-computerinteraction>.

SINGH, S.; SINGH, S.; SINGH, V.; HAZELA4, B. Sorting visualizer: A visual journey through sorting algorithms. *Journal of Informatics Electrical and Electronics Engineering*, v. 5, n. 1, p. 1–9, 2024. ISSN 2582-7006. Disponível em: <https://jieee.a2zjournals.com/index.php/ieee/article/view/107>.

SKIENA, S. S.; REVILLA, M. A. *Programming Challenges: The Programming Contest Training Manual*. New York: Springer, 2003. Disponível em: <https://link.springer.com/book/10.1007/b97559>.

STASKO, J. T.; DOMINGUE, J.; BROWN, M. H.; PRICE, B. A. (Ed.). *Software Visualization: Programming as a Multimedia Experience*. Cambridge, MA: MIT Press, 1998. Acesso em 26 Mai. 2025. ISBN 9780262193955. Disponível em: <https://books.google.com/books?id=13PaPtQqtHUC>.

STROUSTRUP, B. *The C++ Programming Language*. 3rd. ed. Boston, MA: Addison-Wesley, 1997. Referência geral sobre a linguagem C++. Disponível em: <https://www.informit.com/store/c-plus-plus-programming-language-9780201889543>.

Toptal Developers. *Sorting Algorithm Animations*. 2019. Ferramenta Web. Disponível em: <https://www.toptal.com/developers/sorting-algorithms>. Acesso em: 27 jul. 2025.

Apêndices

APÊNDICE A – Implementação e Fluxogramas dos Métodos de Ordenação Existentes na Ferramenta

A.1 *Bubble Sort*

A.1.1 Implementação do *Bubble Sort*

```
1 def bubble_sort(vetor):
2 for i in range(n):
3     for j in range(0, n - i - 1):
4         if vetor[j] > vetor[j + 1]:
5             vetor[j], vetor[j + 1] = vetor[j + 1], vetor[j]
6             draw()  # Atualiza a interface
7             pygame.time.wait(tempo)
```

Listing A.1 – Trecho da implementação do Bubble Sort.

Fonte: Próprio autor.

A.1.2 Fluxo do *Bubble Sort*

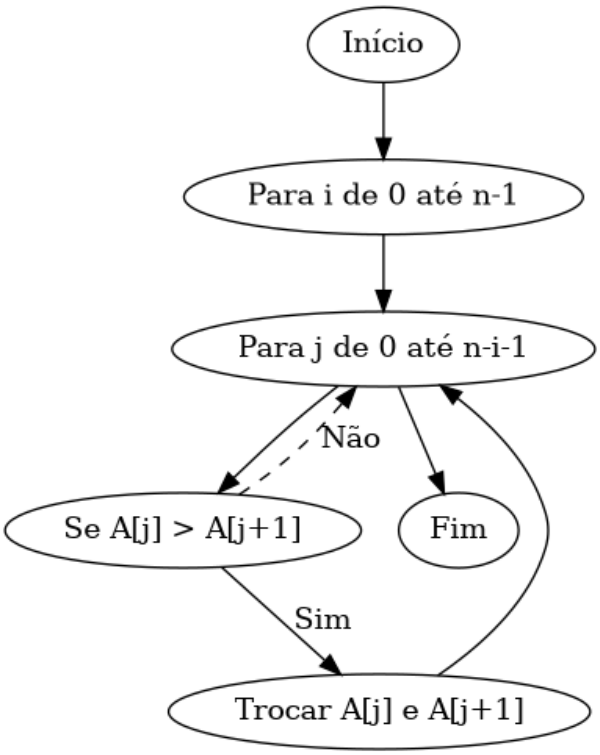

Figura A.1 – Fluxograma *Bubble Sort*.
Fonte: Próprio autor.

A.2 *Insertion Sort*

A.2.1 Implementação do *Insertion Sort*

```
1  def insertion_sort(vetor):
2  for i in range(1, len(vetor)):
3      chave = vetor[i]
4      j = i - 1
5      while j >= 0 and vetor[j] > chave:
6          vetor[j + 1] = vetor[j]
7          j -= 1
8          draw()
9          pygame.time.wait(tempo)
10     vetor[j + 1] = chave
11     draw()
12     pygame.time.wait(tempo)
```

Listing A.2 – Trecho da implementação do Insertion Sort.

Fonte: Próprio autor.

A.2.2 Fluxo do *Insertion Sort*

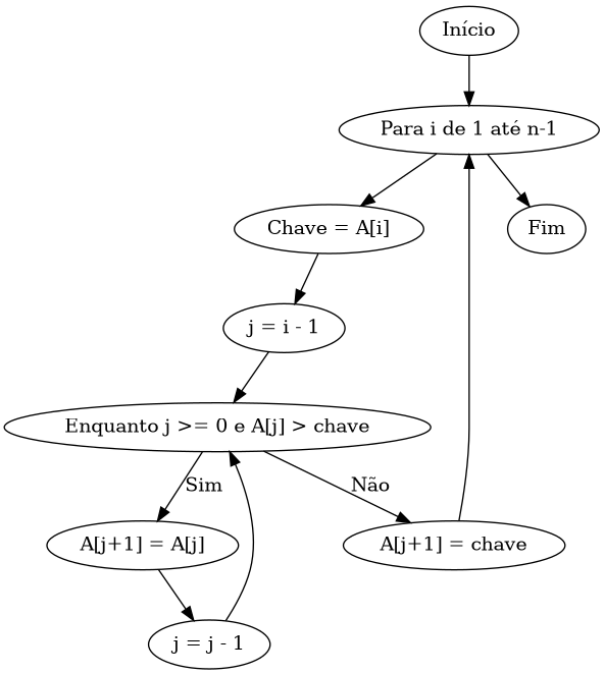

Figura A.2 – Fluxograma *Insertion Sort*.
Fonte: Próprio autor.

A.3 *Selection Sort*

A.3.1 Implementação do *Selection Sort*

```
def selection_sort(vetor):
for i in range(len(vetor)):
    indice_minimo = i
    for j in range(i + 1, len(vetor)):
        if vetor[j] < vetor[indice_minimo]:
            indice_minimo = j
        draw()
        pygame.time.wait(tempo)
    vetor[i], vetor[indice_minimo] = vetor[indice_minimo], vetor[i]
    draw()
    pygame.time.wait(tempo)
```

Listing A.3 – Trecho da implementação do Selection Sort.

Fonte: Próprio autor.

A.3.2 Fluxo do *Selection Sort*

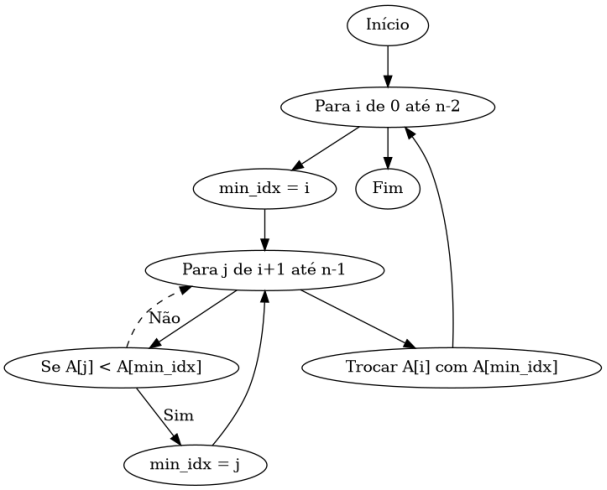

Figura A.3 – Fluxograma *Selection Sort*.
Fonte: Próprio autor.

A.4 *Merge Sort*

A.4.1 Implementação do *Merge Sort*

```
1  def merge_sort(vetor):
2      if len(vetor) > 1:
3          meio = len(vetor) // 2
4          esquerda = vetor[:meio]
5          direita = vetor[meio:]
6
7          merge_sort(esquerda)
8          merge_sort(direita)
9
10         i = j = k = 0
11         while i < len(esquerda) and j < len(direita):
12             if esquerda[i] < direita[j]:
13                 vetor[k] = esquerda[i]
14                 i += 1
15             else:
16                 vetor[k] = direita[j]
17                 j += 1
18             k += 1
19             draw()
20             pygame.time.wait(tempo)
21
```

```
22          while i < len(esquerda):
23              vetor[k] = esquerda[i]
24              i += 1
25              k += 1
26              draw()
27              pygame.time.wait(tempo)
28
29          while j < len(direita):
30              vetor[k] = direita[j]
31              j += 1
32              k += 1
33              draw()
34              pygame.time.wait(tempo)
```

Listing A.4 – Trecho da implementação do Merge Sort.

Fonte: Próprio autor.

A.4.2 Fluxo do *Merge Sort*

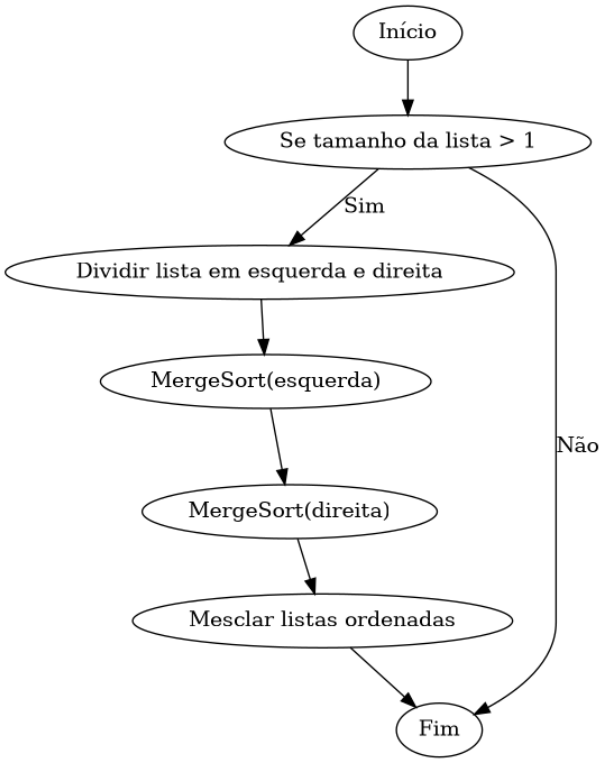

Figura A.4 – Fluxograma *Merge Sort*.
Fonte: Próprio autor.

A.5 *Quick Sort*

A.5.1 Implementação do *Quick Sort*

```
1  def quick_sort(vetor, inicio, fim):
2      if inicio < fim:
3          pivo = particiona(vetor, inicio, fim)
4          quick_sort(vetor, inicio, pivo - 1)
5          quick_sort(vetor, pivo + 1, fim)
6
7  def particiona(vetor, inicio, fim):
8      pivo = vetor[fim]
9      i = inicio - 1
10     for j in range(inicio, fim):
11         if vetor[j] <= pivo:
12             i += 1
13             vetor[i], vetor[j] = vetor[j], vetor[i]
14             draw()
15             pygame.time.wait(tempo)
16     vetor[i + 1], vetor[fim] = vetor[fim], vetor[i + 1]
17     draw()
18     pygame.time.wait(tempo)
19     return i + 1
```

Listing A.5 – Trecho da implementação do Quick Sort.

Fonte: Próprio autor.

A.5.2 Fluxo do *Quick Sort*

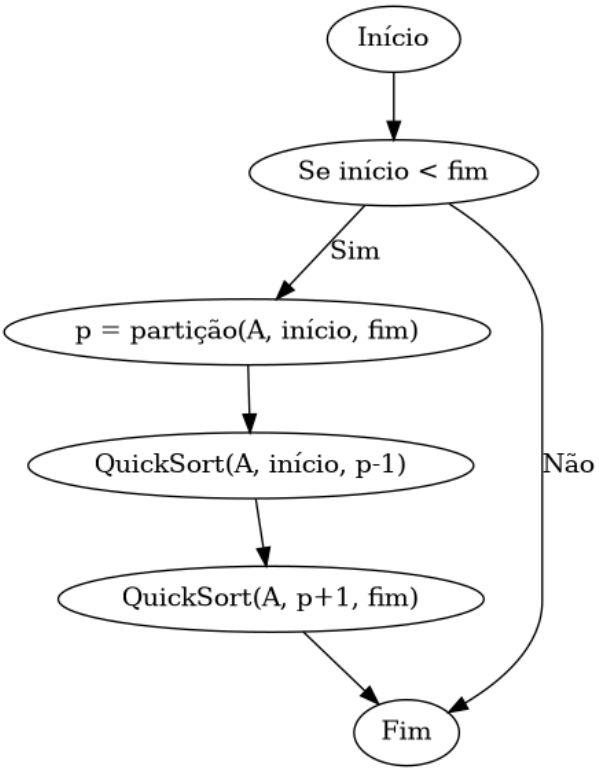

Figura A.5 – Fluxograma *Quick Sort*.
Fonte: Próprio autor.

A.6 *Bucket Sort*

A.6.1 Implementação do *Bucket Sort*

```
1  def bucket_sort(vetor):
2      n = len(vetor)
3      if n == 0:
4          return vetor
5
6
7      k = 10
8
9      buckets = [[] for _ in range(k)]
10
11
12     for num in vetor:
13         index = int(k * num)
14         if index == k:
15             index = k - 1
```

```
16          buckets[index].append(num)
17          draw()
18          pygame.time.wait(tempo)
19
20
21      for i in range(k):
22          insertion_sort_bucket(buckets[i])
23
24
25      idx = 0
26      for b in buckets:
27          for num in b:
28              vetor[idx] = num
29              idx += 1
30              draw()
31              pygame.time.wait(tempo)
32
33  def insertion_sort_bucket(bucket):
34      for i in range(1, len(bucket)):
35          chave = bucket[i]
36          j = i - 1
37          while j >= 0 and bucket[j] > chave:
38              bucket[j + 1] = bucket[j]
39              j -= 1
40          bucket[j + 1] = chave
```

Listing A.6 – Trecho da implementação do Bucket Sort.

Fonte: Próprio autor.

A.6.2 Fluxo do *Bucket Sort*

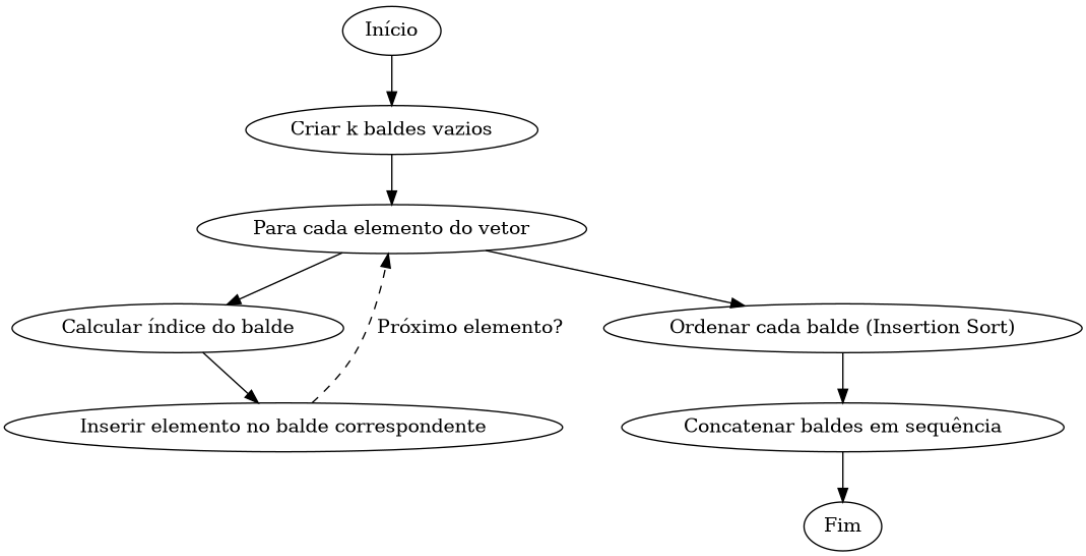

Figura A.6 – Fluxograma *Bucket Sort*.
Fonte: Próprio autor.

A.7 *Smooth Sort*

A.7.1 Implementação do *Smooth Sort*

```
1
2  def smooth_sort(vetor):
3      n = len(vetor)
4      if n < 2:
5          return
6
7      L = [1, 1]
8      while L[-1] < n:
9          L.append(L[-1] + L[-2] + 1)
10
11     sizes = []
12     for i in range(n):
13         sizes.append(1)
14         restore_heap(vetor, sizes)
15         draw()
16         pygame.time.wait(tempo)
17
18     for i in range(n - 1, -1, -1):
19         if sizes:
20             tamanho = sizes.pop()
```

```
21              sift_down(vetor, i - tamanho + 1, tamanho)
22              draw()
23              pygame.time.wait(tempo)
24
25  def restore_heap(vetor, sizes):
26      while len(sizes) >= 2 and sizes[-1] == sizes[-2] + 1:
27          sizes.pop()
28          sizes[-1] += 1
29
30  def sift_down(vetor, inicio, tamanho):
31      if tamanho < 2:
32          return
33      m = inicio + tamanho - 1
34      filho1 = inicio + tamanho - 2
35      filho2 = inicio + tamanho - 3 - (tamanho - 2) // 2
36
37      maior = inicio
38      if vetor[filho1] > vetor[maior]:
39          maior = filho1
40      if vetor[filho2] > vetor[maior]:
41          maior = filho2
42      if maior != inicio:
43          vetor[inicio], vetor[maior] = vetor[maior], vetor[inicio]
```

Listing A.7 – Trecho da implementação do Smooth Sort.

Fonte: Próprio autor.

A.7.2 Fluxo do *Smooth Sort*

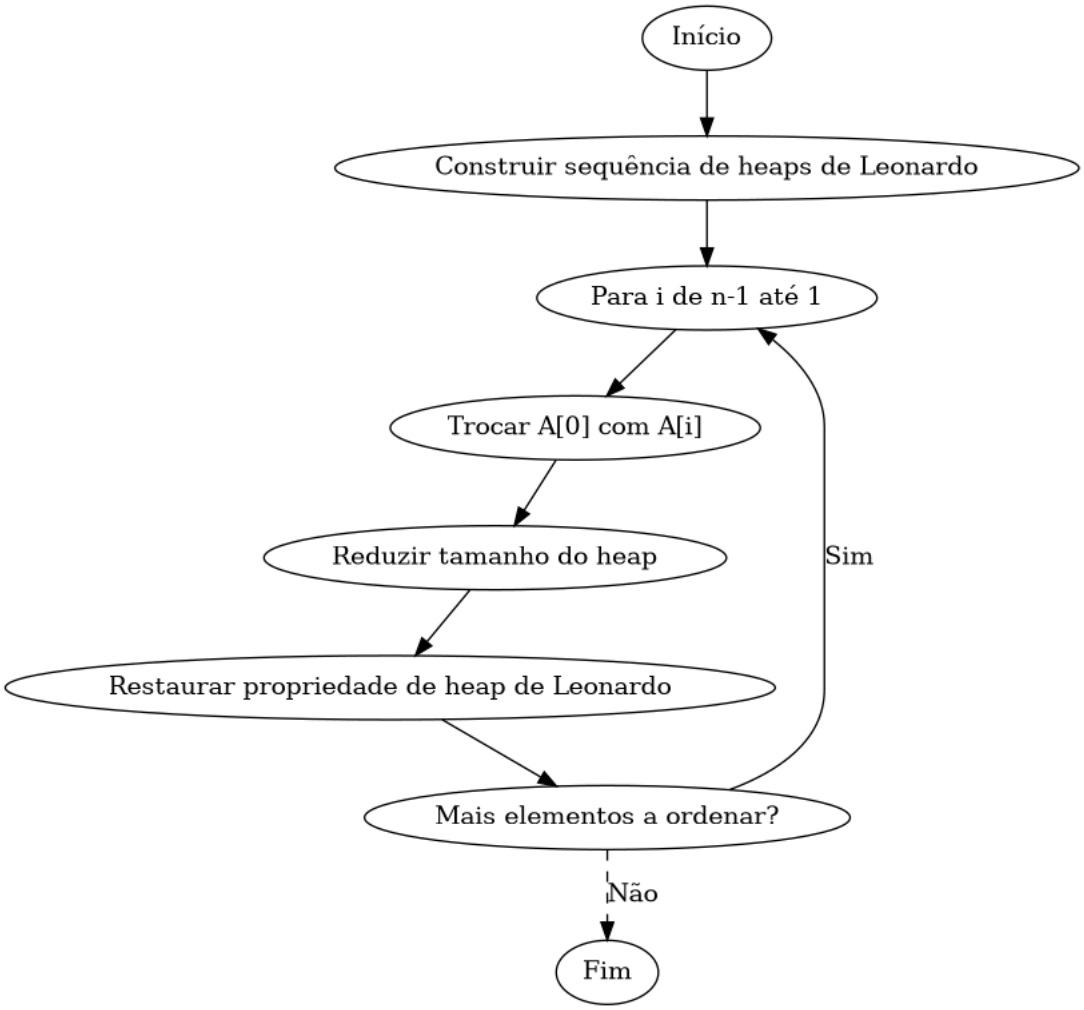

Figura A.7 – Fluxograma *Smooth Sort*.
Fonte: Próprio autor.