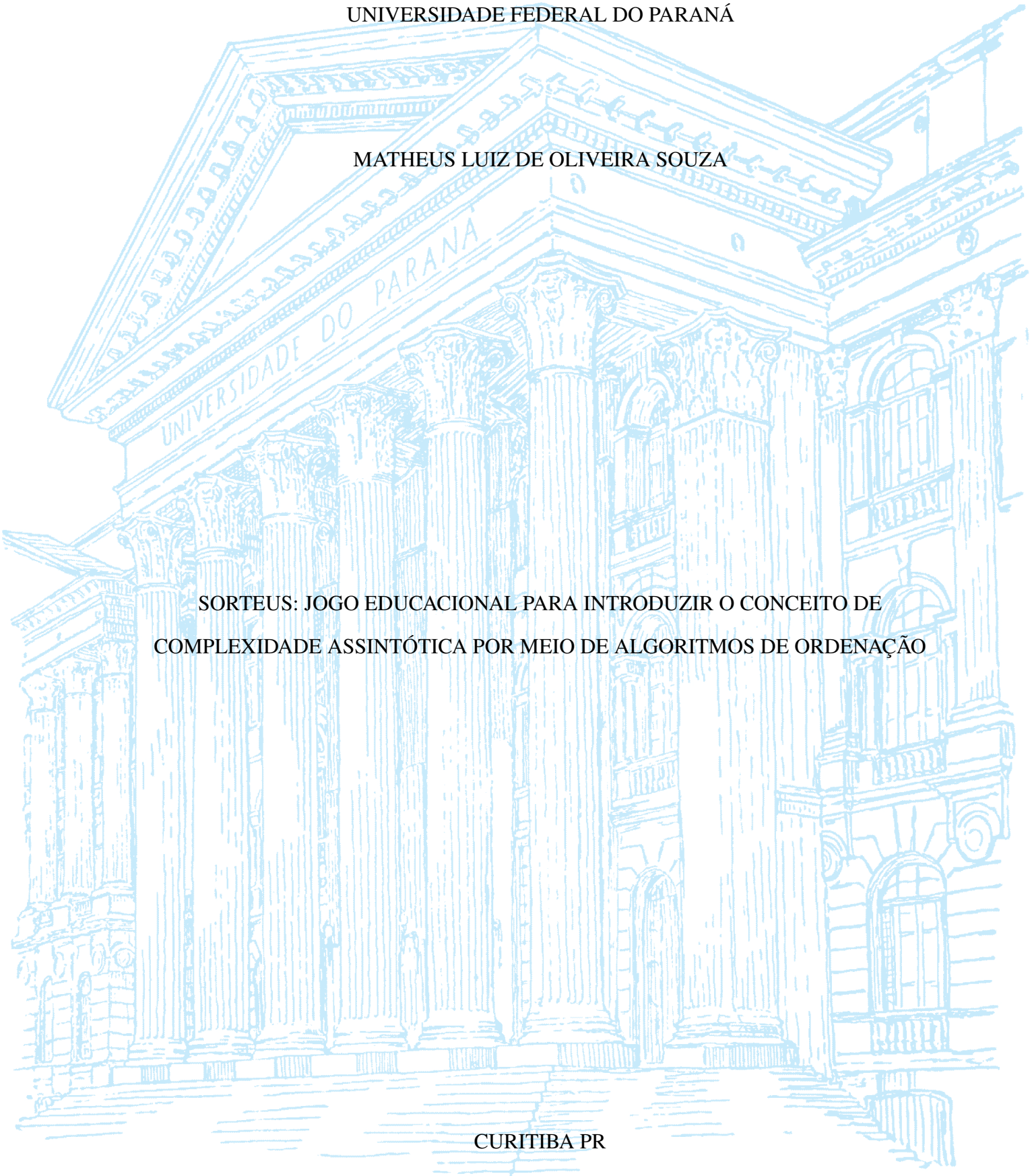
UNIVERSIDADE FEDERAL DO PARANÁ

MATHEUS LUIZ DE OLIVEIRA SOUZA



SORTEUS: JOGO EDUCACIONAL PARA INTRODUZIR O CONCEITO DE COMPLEXIDADE ASSINTÓTICA POR MEIO DE ALGORITMOS DE ORDENAÇÃO

CURITIBA PR

2025

MATHEUS LUIZ DE OLIVEIRA SOUZA

SORTEUS: JOGO EDUCACIONAL PARA INTRODUZIR O CONCEITO DE COMPLEXIDADE ASSINTÓTICA POR MEIO DE ALGORITMOS DE ORDENAÇÃO

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientadora: Rachel Carlos Duque Reis.

CURITIBA PR

2025

À minha mãe e aos meus avós, cujo apoio e incentivo foram os alicerces fundamentais para a conquista deste objetivo.

AGRADECIMENTOS

Expresso minha gratidão à minha orientadora, Rachel Carlos Duque Reis, cujo conhecimento, paciência e conselhos foram fundamentais para nortear a construção deste trabalho. À minha mãe e aos meus avós, agradeço por serem o porto seguro em minha vida; sem o apoio incondicional, o incentivo e o amor de vocês, a conclusão desta etapa não seria possível. Por fim, aos amigos que conquistei durante a graduação, agradeço pela parceria, pelas trocas de conhecimento e por tornarem o ambiente acadêmico um espaço de crescimento mútuo e memórias valiosas.

RESUMO

No contexto da Ciência da Computação, os algoritmos de ordenação desempenham um papel fundamental no processamento de dados, e analisar a eficiência desses algoritmos, por meio da complexidade assintótica, é uma competência essencial para estudantes da área. No entanto, observa-se a carência de ferramentas educacionais que, além de demonstrarem a execução passo a passo desses algoritmos, sejam capazes de mostrar visualmente o impacto do custo computacional e do volume de dados sobre o desempenho. Diante desse cenário, este trabalho apresenta o desenvolvimento de Sorteus, um jogo educacional digital projetado para auxiliar no ensino de algoritmos de ordenação e introduzir conceitos de complexidade assintótica, com foco na notação *Big O*. Desenvolvido na engine Godot e utilizando um modelo incremental, o jogo integra uma narrativa em que o jogador deve solucionar desafios de ordenação em três cenários distintos: aplicando os algoritmos *Bubble Sort*, *Merge Sort* e *Radix Sort*. A principal contribuição desta pesquisa reside na mecânica que vincula a energia do personagem ao esforço computacional de cada algoritmo (comparações ou movimentos), permitindo que o aluno visualize, de forma lúdica, as diferenças de eficiência entre os algoritmos no pior caso. O jogo inclui ainda fases de quiz e pré-quiz com gráficos de desempenho para consolidar o aprendizado teórico. Como resultado, o jogo Sorteus apresenta-se como uma ferramenta de apoio pedagógico capaz de contribuir para o processo de ensino-aprendizagem em cursos de Tecnologia da Informação.

Palavras-chave: Jogo Educacional. Algoritmos de Ordenação. Complexidade Assintótica. Notação Assintótica. Notação *Big O*.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTO E MOTIVAÇÃO

No âmbito da Ciência da Computação, um algoritmo representa qualquer procedimento computacional bem definido que recebe um valor ou conjunto de valores de entrada e produz um valor ou conjunto de valores como saída (Cormen et al., 2012). Entre as diversas categorias existentes, os algoritmos de ordenação desempenham um papel fundamental na computação. Segundo Sedgewick e Wayne (2011), organizar dados é frequentemente o primeiro passo para lidar com o grande volume de informações gerado pelos sistemas modernos, além de servir como ponto de partida para a solução de outros problemas computacionais e desempenhar um papel central no processamento de dados comerciais e científicos.

Uma forma de avaliar a eficiência desses algoritmos é por meio da análise da complexidade assintótica. Segundo Liu (2013), a complexidade assintótica descreve o comportamento limite de uma função de complexidade quando seu argumento aumenta, ignorando fatores constantes e parcelas de menor magnitude na função, ou seja, essa métrica foca em como o consumo de recursos (tempo ou espaço) escala à medida que o tamanho da entrada de dados cresce.

Para representar matematicamente o comportamento analisado pela complexidade assintótica, utiliza-se a notação assintótica. Dentre as notações existentes, destaca-se a *Big O* (O), que define o limite superior de crescimento de uma função. Neste trabalho, seguindo a abordagem metodológica de Bhargava (2017), a notação *Big O* é aplicada para caracterizar o comportamento do algoritmo em seu pior cenário[1]. Dessa forma, a notação é empregada como uma métrica de garantia, assegurando que, independentemente da entrada, o desempenho do algoritmo não excederá o limite estabelecido (Bhargava, 2017).

Para ajudar na compreensão da dinâmica desses algoritmos de ordenação e da análise de eficiência, o uso de sistemas educacionais torna-se estratégico. Nesse cenário, destacam-se os jogos educacionais digitais, os quais, segundo Savi e Ulbricht (2008), proporcionam práticas inovadoras onde o aluno tem a chance de aprender de forma mais ativa, dinâmica e motivadora, tornando-se instrumentos importantes no processo de ensino e aprendizagem. Nesse contexto, tem-se os jogos Ordena (Fritsch et al., 2016) e Isle Sort (Rolim et al., 2024) que têm como objetivo auxiliar na visualização e no entendimento da lógica de alguns algoritmos de ordenação como *Selection Sort* e *Insertion Sort*.

Apesar das importantes contribuições dos jogos educacionais no contexto de algoritmos de ordenação, observou-se que a maioria deles foca, predominantemente, na execução passo a passo de algoritmos baseados em comparação (ex.: *Bubble Sort*, *Selection Sort*, *Insertion Sort*). Embora existam trabalhos que mencionem a complexidade assintótica (Fritsch et al., 2016), nota-se uma carência de abordagens que explorem esse conceito de maneira visual e integrada à mecânica de jogo. Na literatura atual, o custo computacional raramente é traduzido em elementos de jogabilidade, dificultando que o jogador perceba, na prática, o impacto que o aumento do volume de dados exerce sobre o tempo de execução.

[1] Embora a notação *Big O* (O) defina matematicamente um limite superior assintótico que pode ser aplicado a diferentes cenários (melhor caso, caso médio ou pior caso), este trabalho adota a convenção de utilizá-la para expressar a complexidade de pior caso, conforme apresentado em Bhargava (2017), visando estabelecer um limite máximo em relação ao tempo ou espaço para os algoritmos analisados.

No intuito de contribuir com as pesquisas na área de Jogos Educacionais, este trabalho apresenta Sorteus, uma ferramenta que busca integrar o conceito de complexidade assintótica à jogabilidade, vinculando o desempenho do algoritmo de ordenação à energia do personagem ao longo do processo. A ferramenta tem como público-alvo estudantes de graduação em Computação e amplia o escopo de ensino ao incluir algoritmos de ordenação da classe de tempo linear, como o *Radix Sort* (Cormen et al., 2012; Sedgewick e Wayne, 2011), que são pouco explorados nos jogos educacionais.

1.2 OBJETIVO

Este trabalho tem como objetivo apresentar um jogo educacional digital que auxilie no ensino de algoritmos de ordenação, oferecendo uma introdução à complexidade assintótica focada na notação *Big O*.

1.3 DESAFIOS

O desenvolvimento deste trabalho apresentou desafios técnicos e de planejamento pedagógico. O primeiro deles consistiu no aprendizado da ferramenta de desenvolvimento de jogos Godot, exigindo a compreensão de seus recursos para uma implementação eficaz das mecânicas do jogo.

Outro desafio técnico foi o de transpor a lógica dos algoritmos de ordenação para uma linguagem visual intuitiva e didática. Essa dificuldade foi particularmente acentuada no algoritmo *Merge Sort* devido à necessidade de representar visualmente a estratégia de "Divisão e Conquista" (Cormen et al., 2012) de forma prática e compreensível para o jogador.

Por fim, buscou-se uma forma eficiente de construir o conhecimento a respeito dos conceitos básicos de complexidade assintótica. O desafio residiu em como apresentar a notação *Big O*, evitando o uso de definições matemáticas formais, para que o conteúdo se mantivesse acessível e adequado ao contexto do jogo.

1.4 CONTRIBUIÇÃO

A principal contribuição deste trabalho consiste no desenvolvimento e disponibilização do jogo educacional digital Sorteus, uma ferramenta de apoio ao ensino-aprendizagem em cursos de graduação na área de Tecnologia da Informação. O jogo se encontra atualmente disponível em https://sorteus.netlify.app .

Além do software em si, este trabalho buscou contribuir para a área de Jogos Educacionais ao apresentar dois diferenciais em relação aos trabalhos relacionados pesquisados:

1. Foco na complexidade assintótica: A integração do conceito de complexidade assintótica e da notação *Big O* diretamente na mecânica do jogo, permitindo que o estudante visualize o desempenho dos algoritmos por meio do esforço do personagem durante o processo de ordenação, algo pouco explorado visualmente em trabalhos da literatura.

2. Diversificação dos algoritmos: A apresentação didática do *Radix Sort*, um algoritmo de ordenação que opera em tempo linear e não comparativo, expandindo o escopo de jogos educacionais que, predominantemente, buscam ensinar apenas os algoritmos baseados em comparação.

1.5 ORGANIZAÇÃO DO DOCUMENTO

Este Trabalho de Conclusão de Curso (TCC) está estruturado em cinco capítulos, além do capítulo de introdução:

- Capítulo 2 (Fundamentação Teórica): Apresenta os conceitos essenciais para a compreensão deste trabalho, abordando complexidade assintótica, notação assintótica, com foco na notação *Big O*, e os algoritmos de ordenação, detalhando o funcionamento e a complexidade do *Bubble Sort*, *Merge Sort* e *Radix Sort*.

- Capítulo 3 (Trabalhos Relacionados): Analisa jogos educacionais similares voltados ao ensino de algoritmos de ordenação, comparando suas características, públicos-alvo e abordagens pedagógicas, a fim de identificar lacunas que justifiquem o desenvolvimento da presente pesquisa.

- Capítulo 4 (Materiais e Métodos): Descreve a metodologia adotada para o desenvolvimento do jogo, incluindo o modelo incremental, o levantamento de requisitos funcionais e não funcionais, a prototipação das interfaces e as ferramentas de implementação utilizadas.

- Capítulo 5 (Resultados): Apresenta o jogo desenvolvido, exibindo as principais telas e funcionalidades implementadas. O capítulo detalha a integração entre a narrativa e as mecânicas de jogo para os algoritmos *Bubble Sort*, *Merge Sort* e *Radix Sort*, demonstrando como os recursos visuais e interativos foram utilizados para auxiliar no processo de ensino e aprendizagem dos algoritmos em questão e de conceitos de complexidade assintótica focados na notação *Big O*.

- Capítulo 6 (Conclusão): Sintetiza as contribuições deste trabalho e apresenta direcionamentos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo a apresentação de três conceitos essenciais para o entendimento deste trabalho: complexidade assintótica, notação assintótica e algoritmos de ordenação. Inicialmente, serão introduzidos os conceitos de Complexidade Assintótica (Seção 2.1) e Notação Assintótica (Seção 2.2), com foco na notação *Big O*, usados no contexto da análise de algoritmos para classificar a complexidade e a eficiência de algoritmos na medida em que a quantidade dos dados de entrada aumenta. Na sequência, será apresentada a definição de Algoritmos de Ordenação (Seção 2.3), seguida pela análise de métodos específicos. Serão exploradas tanto as estratégias baseadas em comparação — exemplificadas pelo *Bubble Sort* (com critério de parada) e pelo *Merge Sort* — quanto os métodos que operam em tempo linear, representados pelo algoritmo *Radix Sort*.

2.1 COMPLEXIDADE ASSINTÓTICA

Para avaliar a eficiência de algoritmos sem depender de especificidades de hardware ou linguagem de programação, torna-se essencial o entendimento do conceito de complexidade. Conforme define Liu (2013), a complexidade de um algoritmo consiste na mensuração quantitativa dos recursos utilizados — tais como tempo de processamento e espaço em memória — em função das dimensões da entrada de dados.

A autora destaca ainda a importância da complexidade assintótica, que descreve o comportamento limite da função de complexidade à medida que seu argumento (o tamanho da entrada) cresce. Nessa análise, desconsideram-se fatores constantes e termos de menor relevância na função, visando focar exclusivamente na taxa de crescimento. Isso permite compreender, de forma objetiva, a escalabilidade do algoritmo frente a grandes volumes de dados.

Embora fundamentada em conceitos matemáticos, a análise apresentada neste TCC foca na classificação prática dos algoritmos. O intuito é compreender as diferenças de desempenho sem adentrar em demonstrações complexas, utilizando-se, como padrão de representação, a notação assintótica.

2.2 NOTAÇÃO ASSINTÓTICA

Um dos propósitos da notação assintótica é fornecer uma caracterização objetiva da eficiência de um algoritmo, além de permitir a comparação de desempenho entre diferentes algoritmos (Cormen et al., 2012). Dentre as notações assintóticas mais utilizadas, destacam-se a notação O (*Big O*), a notação (Ômega) e a notação (Theta) (Szwarcfiter e Markezon, 2010). Neste TCC, a análise terá como foco a notação *Big O*, seguindo a abordagem de Bhargava (2017), por ser uma ferramenta frequentemente adotada para a caracterização do pior cenário de execução.

Embora matematicamente a notação *Big O* defina um limite superior assintótico genérico — podendo ser utilizada tanto ao pior quanto ao melhor caso — sua aplicação neste trabalho concentra-se na garantia de desempenho no pior cenário. No contexto de tempo de execução, a notação é usada para descrever como o esforço computacional cresce à medida que aumenta o tamanho da entrada, focando na quantidade de operações realizadas em vez de medidas temporais absolutas. Sob essa ótica, o *Big O* expressa o comportamento no pior caso possível, assegurando que o algoritmo não excederá o limite de operações indicado (Bhargava, 2017).

Para exemplificar, a Figura 2.1 apresenta um gráfico de complexidade *Big O*, que ilustra como diferentes algoritmos se comportam em relação ao tempo de execução conforme o tamanho da entrada aumenta. No eixo horizontal, é exibida a quantidade de elementos da entrada (n), enquanto no eixo vertical está o mapeamento do número de operações. Cada curva representa uma função de complexidade, como $O(1)$, $O(logn)$, $O(n)$, $O(nlogn)$, $O(n^2)$, $O(2^n)$ e $O(n!)$. As cores indicam o nível de eficiência: verde indica algoritmos excelentes (rápidos com entradas grandes), amarelo para bons, laranja para razoáveis e vermelho para ineficientes (inviáveis para entradas muito grandes).

Vale ressaltar que, em classes específicas de algoritmos, como os algoritmos baseados em comparação, ter complexidade de pior caso $O(nlogn)$ é assintoticamente ótimo. A demonstração matemática desse resultado pode ser encontrada em Cormen et al. (2012).

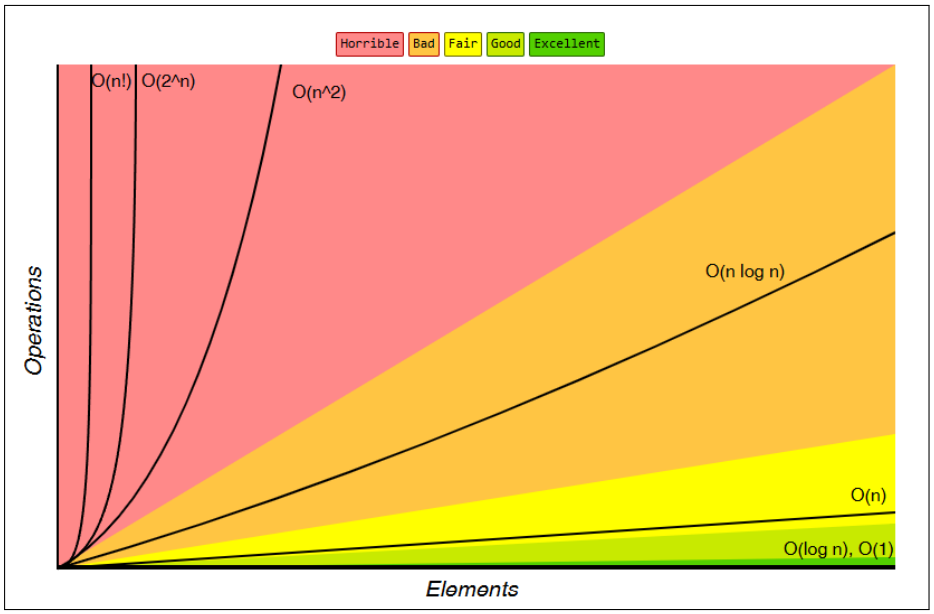
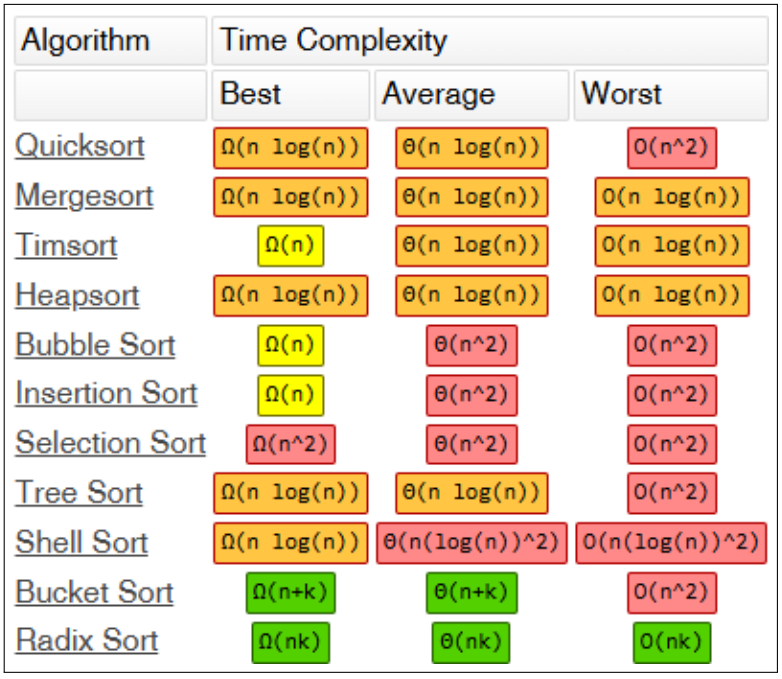


Figura 2.1: Gráfico de complexidade usando a notação *Big O*. Fonte: Big-O Cheat Sheet.

Na Figura 2.2, é possível verificar as complexidades assintóticas, em relação ao tempo de execução, de alguns algoritmos de ordenação. A notação *Big O* está vinculada com o pior caso de cada algoritmo e encontra-se na coluna mais à direita (*Worst*).



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)

Figura 2.2: Complexidade de tempo no melhor caso (*best*), caso médio (*average*) e pior caso (*worst*) para diferentes algoritmos de ordenação. Fonte: Big-O Cheat Sheet.

2.3 ALGORITMOS DE ORDENAÇÃO

2.3.1 Definição geral

Um algoritmo é qualquer procedimento computacional bem definido que recebe um valor ou conjunto de valores de entrada e produz um valor ou conjunto de valores como saída (Cormen et al., 2012). Um algoritmo de ordenação, por sua vez, é uma sequência de passos computacionais que organiza os elementos de um conjunto em uma determinada ordem, geralmente crescente ou decrescente.

De maneira mais formal, ordenar um conjunto de dados significa organizar seus elementos de forma que, para qualquer par de elementos a_i e a_j, onde i e j são os índices dos elementos no conjunto, se $i < j$, então $a_i \leq a_j$, para o caso da ordenação crescente. Por exemplo, considere o vetor $A = [8, 3, 5, 1]$. Nele temos $a_0 = 8$, $a_1 = 3$, $a_2 = 5$ e $a_3 = 1$. Embora os índices obedeçam à relação $0 < 1$, os seus valores correspondentes ($a_0 = 8$ e $a_1 = 3$) não satisfazem a condição $a_0 \leq a_1$, uma vez que $8 > 3$. Portanto, a ordem está incorreta e precisa ser corrigida para possibilitar a obtenção da saída correta. Com isso, após a ordenação, teríamos $A = [1, 3, 5, 8]$, onde para todos os pares com $i < j$, temos $a_i \leq a_j$, o que caracteriza uma ordenação crescente. A mesma lógica pode ser aplicada para a ordenação decrescente, bastando inverter a comparação dos elementos para $a_i \geq a_j$.

2.3.2 Tipos de Algoritmos de Ordenação

Segundo Cormen et al. (2012), os algoritmos de ordenação podem ser classificados em dois grandes grupos: (1) os baseados em comparação e (2) os que operam em tempo linear. Os algoritmos de ordenação por comparação determinam a ordem dos elementos a partir de comparações entre pares. Nesse contexto, tem-se como exemplos o *Bubble Sort*, *Selection Sort*, *Quick Sort* e *Merge Sort*, cada um com diferentes níveis de eficiência e aplicabilidade.

Por outro lado, os algoritmos de ordenação em tempo linear exploram características específicas dos dados para alcançar maior eficiência, sem utilizar comparações diretas. Entre eles, destacam-se o *Counting Sort*, o *Radix Sort* e o *Bucket Sort*. A seguir são detalhados dois algoritmos de ordenação baseados em comparação (Seção 2.3.2.1 e Seção 2.3.2.2) e um algoritmo que, em condições específicas, opera em tempo linear (Seção 2.3.2.3).

2.3.2.1 Bubble Sort

O *Bubble Sort* é um algoritmo de ordenação simples que possui o seguinte mecanismo: uma iteração desse algoritmo consiste em percorrer uma lista do início ao fim e, durante esse percurso, são realizadas trocas de posição entre dois elementos consecutivos sempre que estes estiverem fora de ordem (Szwarcfiter e Markezon, 2010). Se a ordenação for crescente, a intenção desse método é mover os elementos maiores em direção ao fim da lista, ou seja, após a primeira iteração, o maior elemento estará garantidamente na última posição. Na segunda, o segundo maior elemento estará na penúltima e assim por diante.

Além disso, esse algoritmo possui versões com e sem critérios de parada. Na versão sem critério de parada (não otimizada), o algoritmo executa um número fixo de iterações (*loop* aninhado indo de 1 até n, onde n é a quantidade elementos da lista), resultando em uma complexidade de tempo quadrática para todos os cenários. Na versão com critério de parada (otimizada), por sua vez, foram incorporadas duas melhorias:

1. Como a cada iteração o maior elemento vai para a sua posição final, não é necessário compará-lo novamente, ou seja, é reduzido a posição final da lista em um elemento a cada iteração.

2. Se o algoritmo completar uma iteração e não realizar nenhuma troca, então a lista já ficará ordenada e o algoritmo pode parar.

Com essas melhorias, a versão otimizada consegue obter uma complexidade linear no melhor caso (conforme apresentado na Figura 2.2 na coluna *Best*). Isso ocorre quando a lista já está ordenada, permitindo que apenas uma única passagem seja realizada sem a necessidade de trocas, o que leva à finalização do algoritmo. Porém, assim como a versão sem critério de parada, essa versão continua sendo quadrática no pior caso, ou seja, $O(n^2)$ (cenário de uma lista na ordem inversa ao objetivo da ordenação) (Szwarcfiter e Markezon, 2010).

A Figura 2.3 ilustra o funcionamento do *Bubble Sort*, com critério de parada, utilizando como exemplo a lista não ordenada [7,4,9,2] para ser colocada em ordem crescente. À esquerda das setas (em cada iteração) estão representados os estados da lista antes de uma comparação e os elementos destacados em azul representam os elementos em análise. À direita da seta (em cada iteração) estão os resultados após a comparação. Os elementos destacados em verde mostram que eles trocaram de posição. Por fim, os elementos marcados em roxo indicam os elementos que já estão em sua posição final e não serão mais utilizados para comparação.

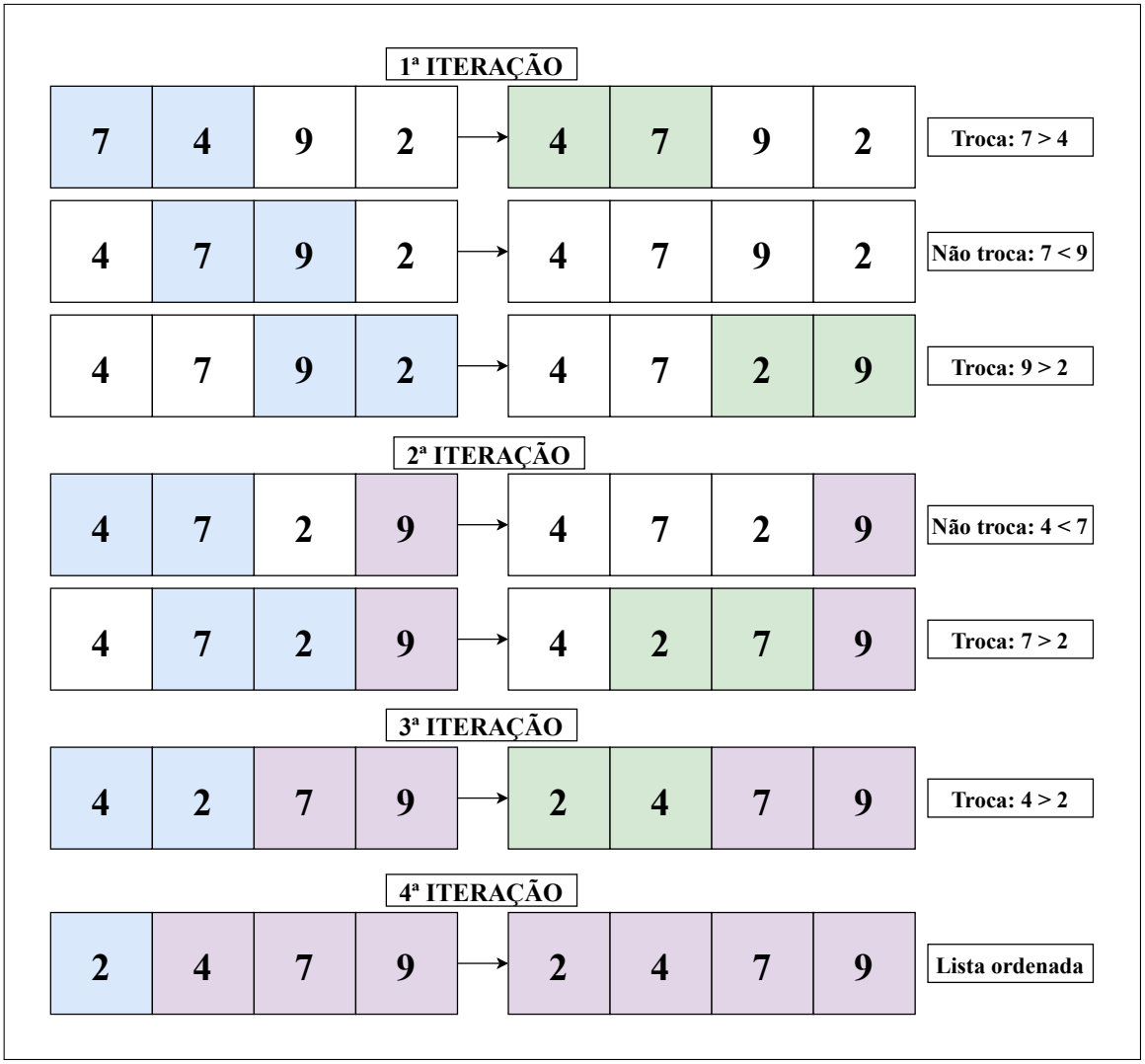


Figura 2.3: Exemplo de execução do *Bubble Sort* com critério de parada. Fonte: Autoria própria.

2.3.2.2 Merge Sort

Muitos algoritmos são construídos com base em chamadas recursivas, de modo que, para resolver um problema maior, eles resolvem versões menores do mesmo problema. Essa estratégia é chamada de divisão e conquista (Cormen et al., 2012). A ideia central é dividir o problema original em subproblemas menos complexos, resolver cada um deles e depois juntar os resultados. Essa técnica é geralmente aplicada em três etapas:

1. Divisão: O problema é separado em subproblemas menores e semelhantes ao original.
2. Resolução/Conquista: Cada subproblema é tratado individualmente e, geralmente, de forma recursiva. Quando os subproblemas são simples o suficiente, eles podem ser resolvidos diretamente sem mais divisões.
3. Combinação: As soluções das partes são reunidas para formar a resposta final do problema original.

Um exemplo de algoritmo de ordenação que utiliza a estratégia de divisão e conquista é o *Merge Sort* (Cormen et al., 2012). O funcionamento desse algoritmo segue os seguintes passos:

1. Dividir: A lista a ser ordenada é dividida ao meio, criando duas listas menores com aproximadamente o mesmo tamanho. Pode haver variação de tamanho entre as listas quando a lista a ser dividida ao meio tiver uma quantidade ímpar de elementos.

2. Conquistar: Cada sub-lista é ordenada recursivamente aplicando o próprio *Merge Sort*.

3. Mesclar: As metades já ordenadas são combinadas em uma nova lista ordenada e esse processo continua até a obtenção da solução final.

A Figura 2.4 apresenta um exemplo de execução do algoritmo *Merge Sort* sobre a lista desordenada $[7, 4, 9, 2]$. Inicialmente, a lista é dividida em duas metades: $[7, 4]$ e $[9, 2]$. Cada uma dessas partes é subdividida até que se obtenham listas com apenas um elemento: $[7]$, $[4]$, $[9]$ e $[2]$. Em seguida, o algoritmo realiza a etapa de mesclagem, unindo as sub-listas de forma ordenada. Assim, $[7]$ e $[4]$ se tornam $[4, 7]$, enquanto $[9]$ e $[2]$ formam $[2, 9]$. Por fim, essas listas intermediárias são combinadas, resultando na lista final ordenada $[2, 4, 7, 9]$. Por fim, em termos de complexidade assintótica em relação ao tempo, o *Merge Sort* é $O(nlogn)$ (Roughgarden, 2017).

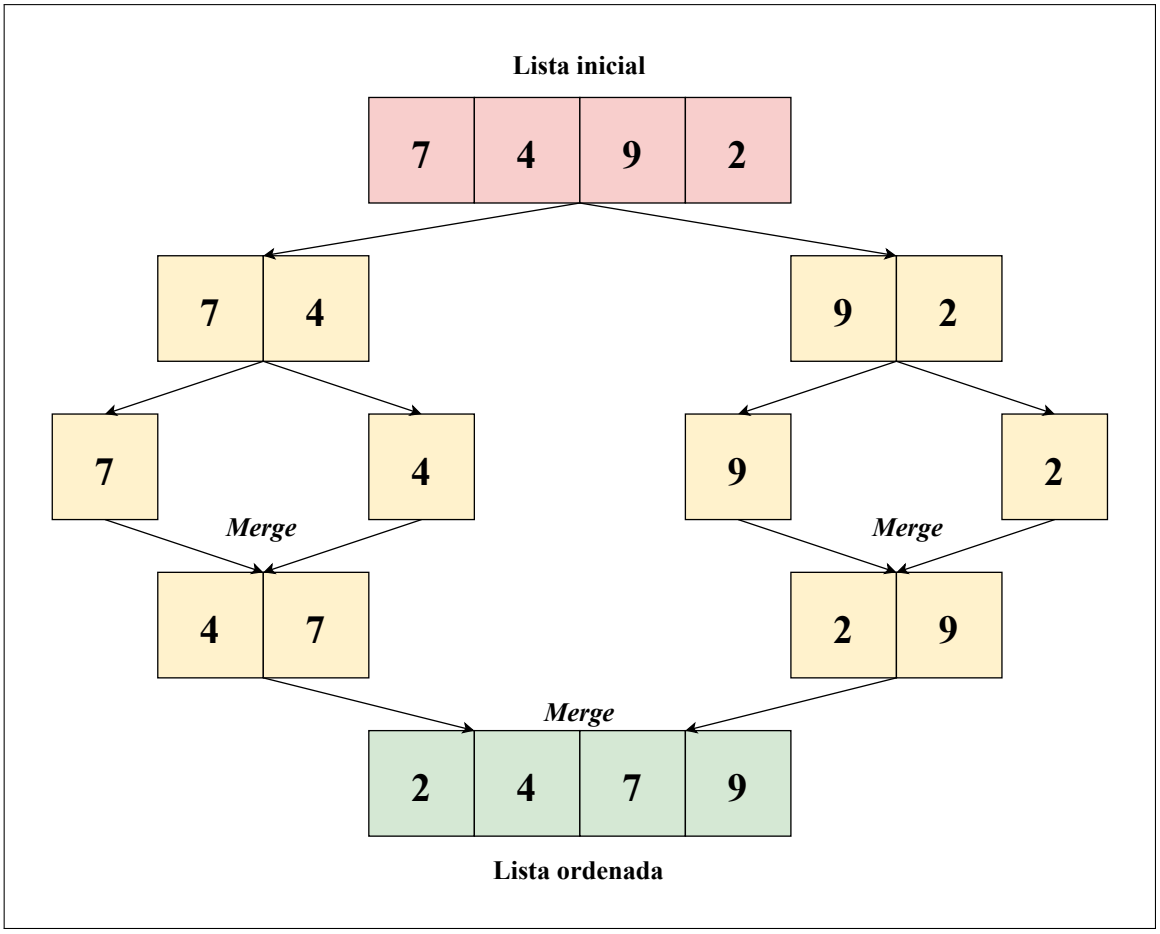


Figura 2.4: Exemplo de execução do Merge Sort. Fonte: Autoria própria.

2.3.2.3 Radix Sort LSD

O *Radix Sort LSD* (*Least-Significant-Digit first*) é um algoritmo de ordenação que opera ordenando os elementos com base nos seus dígitos ou caracteres, começando sempre pelo menos significativo (o mais à direita) e avançando para o mais significativo (o mais à esquerda)

(Sedgewick e Wayne, 2011). Esse algoritmo é útil para ordenar dados onde todas as chaves (os itens a serem ordenados) possuem a mesma quantidade de algarismos (para o caso de números) (Cormen et al., 2012) ou a mesma quantidade de caracteres (para o caso de *strings*) (Sedgewick e Wayne, 2011).

O funcionamento central do *Radix Sort LSD* depende fundamentalmente da utilização de outro algoritmo de ordenação que seja estável para cada passagem nos dados (Sedgewick e Wayne, 2011). Por exemplo, o *Counting Sort* é frequentemente utilizado como o algoritmo intermediário para o *Radix Sort*, conforme demonstrado nas provas dos Lemas 8.3 e 8.4 apresentadas em Cormen et al. (2012). Para dados com X dígitos ou X caracteres, o algoritmo é aplicado X vezes, uma única vez para cada posição, começando pelo dígito ou caractere menos significativo (Sedgewick e Wayne, 2011). A Figura 2.5 exibe um exemplo de funcionamento do algoritmo *Radix Sort LSD* para a lista não ordenada $[329, 457, 657, 839, 436, 720, 355]$.

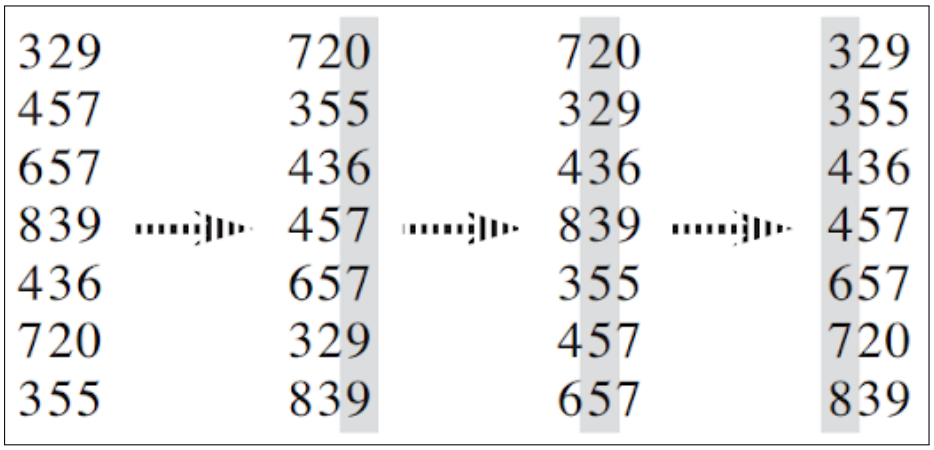


Figura 2.5: Exemplo de execução do *Radix Sort LSD*. Fonte: (Cormen et al., 2012).

Conforme ilustrado na Figura 2.5, o primeiro bloco de números à esquerda representa a entrada. Em cada um dos blocos subsequentes, a coluna destacada em cinza indica o dígito analisado naquele passo da ordenação. A ordenação começa pela casa das unidades (dígito menos significativo), segue para as dezenas e finaliza na casa das centenas (dígito mais significativo). O último bloco à direita mostra a etapa final, na qual o dígito mais significativo é considerado, completando assim a ordenação dos elementos e resultando na lista $[329, 355, 436, 457, 657, 720, 839]$.

Como já apresentado, no *Radix Sort LSD* a estabilidade da ordenação em cada passagem é fundamental para a corretude do algoritmo (Cormen et al., 2012). Uma ordenação estável garante que, se dois elementos possuem o mesmo valor na posição de dígito que está sendo ordenado no momento, a ordem relativa entre eles será preservada. Como as passagens anteriores já estabeleceram a ordem correta com base nos dígitos menos significativos, manter essa ordem para elementos que são iguais no dígito atual assegura que eles permaneçam na posição correta em relação uns aos outros (Sedgewick e Wayne, 2011).

Por fim, é definido em Goldman e Goldman (2008) que o *Radix Sort* ordena em tempo $O(d(n + b))$. Das variáveis apresentadas, n representa a quantidade de elementos, d é o número máximo de dígitos ou caracteres e b corresponde à quantidade de possíveis valores que cada dígito pode assumir. No jogo, foi utilizado o sistema decimal, o que fixa a base em $b = 10$ (representando os baldes de 0 a 9). Como a notação *Big O* foca na taxa de crescimento da função e ignora constantes aditivas, o termo b torna-se irrelevante para grandes entradas. Assim, a complexidade do algoritmo neste cenário é denotada como $O(d \cdot n)$.

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados cinco trabalhos que abordam o desenvolvimento de jogos educacionais digitais voltados ao ensino de algoritmos de ordenação. Na análise desses trabalhos, buscou-se identificar as principais características dos jogos e como eles podem contribuir para o ensino e aprendizagem desses algoritmos.

Fritsch et al. (2016) apresentam Ordena, um jogo educacional digital voltado a estudantes de cursos de graduação na área de Tecnologia da Informação, com o objetivo de facilitar a compreensão de algoritmos de ordenação, conteúdo comumente trabalhado em disciplinas de estruturas de dados. Desenvolvido na plataforma Unity 3D para o sistema operacional Android, o jogo propõe ao usuário o desafio de ordenar uma lista de bolas numeradas fora de sequência, realizando trocas entre pares conforme os passos de execução dos algoritmos *Bubble Sort*, *Selection Sort* e *Insertion Sort*. Trocas incorretas são sinalizadas, exigindo que o jogador finalize a ordenação com no máximo dois erros e dentro de um limite de tempo específico.

O jogo apresenta 72 fases com dificuldade progressiva, variando quanto ao tempo disponível e número de trocas exigidas para a ordenação completa, além de exibir a complexidade do pior caso de cada algoritmo. Como o jogo ainda não foi formalmente avaliado, os autores ressaltam a importância de investigações futuras sobre sua eficácia como ferramenta de apoio ao processo de aprendizagem.

Rolim et al. (2024) apresentam o desenvolvimento de um jogo educacional digital com o objetivo de auxiliar no ensino de algoritmos de ordenação e busca. O jogo, denominado Isle Sort, foi construído utilizando a plataforma Unity e pertence ao gênero *Puzzle*. Além disso, foi projetado para proporcionar uma compreensão prática e visual dos algoritmos *Bubble Sort*, *Insertion Sort*, *Selection Sort* e Busca Binária.

O jogo tem como público-alvo indivíduos com 15 anos ou mais que enfrentam dificuldades no aprendizado de algoritmos e estruturas de dados. Sua construção fundamenta-se em duas teorias: a Teoria da Carga Cognitiva, buscando aprimorar o processo de aprendizagem por meio da redução da sobrecarga cognitiva dos estudantes (situação em que uma pessoa recebe uma gama de informações em um curto intervalo de tempo), e a Teoria da Aprendizagem Significativa, promovendo uma compreensão mais profunda e efetiva do conteúdo. Como se trata de um protótipo, ainda em fase de desenvolvimento, testes envolvendo o público alvo ainda não foram realizados, e, portanto, o artigo não apresenta o mapeamento de seus resultados.

Battistella et al. (2016) apresentam um jogo desenvolvido para auxiliar no aprendizado do algoritmo de ordenação *Heapsort*, chamado SORTIA 2.0. Trata-se de um jogo *single player* que adota a abordagem de Aprendizado Baseado em Jogos para tornar o ensino de conteúdos da computação mais atrativo para os alunos. A escolha pelo algoritmo *Heapsort* deve-se à dificuldade que os estudantes dos cursos de graduação em Ciência da Computação, público-alvo deste jogo, frequentemente enfrentam para compreender e aplicar este algoritmo. Como o *Heapsort* possui um nível de complexidade relativamente alto, o jogo busca contribuir para a assimilação mais eficaz do conteúdo.

O jogo foi implementado utilizando as linguagens de programação Javascript, PHP e a linguagem de marcação HTML5. Para a dinâmica de interação, os docentes responsáveis pela disciplina de Estrutura de Dados da Universidade Federal de Santa Catarina, inicialmente, explicaram o algoritmo em uma aula expositiva de aproximadamente uma hora. Em seguida, 25 estudantes jogaram por cerca de uma hora e, ao final, avaliaram a aplicação com base nos quesitos de “motivação”, “experiência do usuário” e “aprendizagem”. Os resultados indicaram

uma boa aceitação do jogo, onde 80% dos estudantes relataram estarem satisfeitos com o jogo no quesito “motivação”, 88% avaliaram a “experiência do usuário” como positiva, e 92% afirmaram que a ferramenta contribuiu significativamente para o “aprendizado” na disciplina.

Raia et al. (2023) descrevem o desenvolvimento do FruitSort, um jogo educacional digital voltado para crianças de 7 a 11 anos com deficiência auditiva, com o objetivo de apoiar o ensino do pensamento computacional por meio de um ambiente lúdico e inclusivo. A proposta do jogo é explorar o algoritmo de ordenação *Merge Sort*, substituindo números por frutas de diferentes tamanhos para facilitar a compreensão visual e manter o engajamento do público-alvo.

O jogo foi desenvolvido utilizando a linguagem de programação JavaScript, a linguagem de marcação HTML e a linguagem de estilos CSS. Embora o FruitSort ainda esteja em fase de protótipo, seu *design* contempla elementos de interatividade, usabilidade e acessibilidade, visando a construção de um ambiente educacional inclusivo e estimulador, apesar de ainda não contar com validações práticas junto ao público-alvo.

Fürlinger (2024) apresenta o *design* e o desenvolvimento de um jogo educacional digital de cartas voltado ao ensino de algoritmos de ordenação para crianças. A mecânica consiste nos jogadores virarem duas cartas por vez e decidirem, com base na lógica do algoritmo em uso (*Bubble Sort*, *Insertion Sort* ou *Selection Sort*), se devem trocá-las de posição ou prosseguir.

A aplicação oferece dois modos de interação: o “Modo Livre” e o “Modo Guiado”. O “Modo Livre” permite aos usuários explorarem livremente as ações de comparação e troca sem diretrizes fixas, correções automáticas ou dicas, incentivando-os a ordenar as cartas conforme sua própria estratégia. O “Modo Guiado”, por sua vez, conduz os jogadores passo a passo pelos algoritmos *Bubble Sort*, *Insertion Sort* e *Selection Sort*, fornecendo dicas visuais e mensagens corretivas sempre que há desvios em relação aos procedimentos de execução do algoritmo em questão.

O jogo foi desenvolvido com Vue.js, HTML, CSS e JavaScript. Como ainda não foi submetido a uma avaliação formal, o autor destaca a necessidade de coletar *feedback* do público-alvo para orientar melhorias futuras.

A Tabela 3.1 sintetiza as principais características dos jogos educacionais apresentados neste capítulo, incluindo o público-alvo, os algoritmos de ordenação explorados, as tecnologias empregadas e a presença, ou não, de análises sobre a complexidade dos algoritmos. Essas informações oferecem uma visão estruturada das abordagens adotadas no desenvolvimento dos jogos educacionais, evidenciando como essas ferramentas podem ser utilizadas para o aprendizado de conceitos relacionados à ordenação.

Tabela 3.1: Resumo dos jogos educacionais voltados ao ensino de algoritmos de ordenação.

Autores	**Jogo**	**Público-alvo**	**Algoritmos de ordenação**	**Tecnologias**	**Complexidade assintótica dos algoritmos?**
Fritsch et al. (2016)	*Ordena*	Estudantes de graduação	*Bubble Sort* *Selection Sort* *Insertion Sort*	Unity 3D Android	Exibe o pior caso de cada algoritmo utilizando a notação *Big O*
Rolim et al. (2024)	*Isle Sort*	Estudantes do ensino médio e superior	*Bubble Sort* *Selection Sort* *Insertion Sort*	Unity	-
Battistella et al. (2016)	*SORTIA 2.0*	Estudantes de graduação	*Heap Sort*	JavaScript PHP HTML	-
Raia et al. (2023)	*FruitSort*	Crianças (7 a 11 anos) com deficiência auditiva	*Merge Sort*	JavaScript HTML CSS	-
Fürlinger (2024)	Sem nome definido	Crianças	*Bubble Sort* *Selection Sort* *Insertion Sort*	Vue.js JavaScript HTML CSS	-

Conforme apresentado na Tabela 3.1, os jogos educacionais analisados abrangem públicos variados, atendendo desde crianças até estudantes de graduação. Esses jogos têm como objetivo introduzir e/ou reforçar os conceitos relacionados aos algoritmos de ordenação. Dentre os algoritmos abordados, destacam-se *Bubble Sort*, *Selection Sort* e *Insertion Sort*, devido à simplicidade e facilidade de visualização.

Apesar da existência de um jogo que mencione a complexidade assintótica (Fritsch et al., 2016), observa-se que ainda há uma lacuna quanto a abordagens que explorem esse conceito de maneira mais representativa. Assim, este trabalho busca contribuir com as pesquisas na área de Jogos Educacionais, a partir do desenvolvimento de um jogo digital que explore o conceito de complexidade assintótica. Além disso, o jogo irá abordar os algoritmos *Bubble Sort* e *Merge Sort* e um algoritmo de ordenação não baseado em comparação, no caso o *Radix Sort*, destacando suas principais diferenças e contextos mais adequados para sua aplicação.

4 MATERIAIS E MÉTODOS

O jogo Sorteus foi desenvolvido utilizando o modelo incremental, no qual o sistema evolui por meio de versões sucessivas que são avaliadas e aprimoradas continuamente. Conforme descrito por Sommerville (2011), esse modelo permite alternar especificação, desenvolvimento e validação em ciclos curtos, facilitando ajustes conforme o projeto avança. Seguindo essa abordagem, o processo de criação do jogo foi organizado em três etapas principais: levantamento de requisitos, prototipação e implementação.

4.1 LEVANTAMENTO DE REQUISITOS

Primeiramente, foram definidos os requisitos funcionais e não funcionais que orientaram o escopo e os objetivos do jogo. Cada um desses requisitos serão detalhados nas Subseções 4.1.1 e 4.1.2.

4.1.1 Requisitos Funcionais

A seguir, são listados os requisitos funcionais que descrevem as principais funcionalidades do jogo Sorteus:

1. Apresentar de forma visual e clara o passo a passo dos algoritmos de ordenação *Bubble Sort* (com critério de parada), *Merge Sort* e *Radix Sort*.

2. Possibilitar que o jogador interaja com a interface a fim de ordenar conjuntos de livros, frutas ou cartas, utilizando algoritmos de ordenação específicos, como *Bubble Sort*, *Merge Sort* ou *Radix Sort*.

3. Fornecer *feedback* imediato e instrutivo quando o jogador cometer algum erro ao tentar ordenar incorretamente, utilizando algum dos algoritmos de ordenação.

4. Introduzir os algoritmos de ordenação de forma gradual, começando pelo *Bubble Sort*, depois o *Merge Sort* e, por fim, o *Radix Sort*.

5. Mostrar de maneira lúdica o conceito de complexidade assintótica exibindo: (1) a quantidade de comparações/movimentos realizadas pelo algoritmo e (2) a quantidade de energia gasta pelo personagem, representando o esforço do algoritmo, ou seja, o sistema exibe uma barra de energia e o estado afetivo do personagem, de modo que esse estado varia de acordo com a quantidade de comparações ou movimentos.

6. Após a execução de cada algoritmo de ordenação, o jogo fornece uma análise de desempenho em duas etapas: (1) um pré-quiz que apresenta conceitos de complexidade assintótica e explica o pior caso (notação *Big O*) do algoritmo em questão; (2) um quiz com perguntas que ajudam a fixar o entendimento sobre a complexidade assintótica e sobre o próprio algoritmo analisado.

4.1.2 Requisitos Não Funcionais

Além dos requisitos funcionais, também foram definidos requisitos não funcionais para assegurar a qualidade e o bom funcionamento do sistema, tais como fornecer uma interface gráfica intuitiva e acessível, bem como a estabilidade e o bom desempenho do jogo.

A estrutura narrativa do jogo é centrada no personagem Teus, que recebe a missão de levar um medicamento para a casa de sua avó. A progressão do jogador ocorre ao longo desse trajeto, atravessando três cenários distintos: uma biblioteca, uma feira de comidas e uma rua onde encontra um carteiro. Em cada etapa, o protagonista interage com personagens secundários para solucionar desafios específicos de organização (livros, frutas e cartas), correspondendo, respectivamente, ao aprendizado dos algoritmos *Bubble Sort*, *Merge Sort* e *Radix Sort*.

4.2 PROTOTIPAÇÃO

Antes da implementação do jogo, foi realizada a prototipação de baixa fidelidade, utilizando a ferramenta Figma, com o objetivo de estruturar a interface gráfica e planejar a experiência do usuário. Neste protótipo, o foco foi estruturar os *layouts* que compõem o ciclo principal de cada nível. Foram desenhadas as telas correspondentes às três etapas fundamentais da jogabilidade: a execução prática do algoritmo, a visualização da análise de desempenho (pré-quiz) e o questionário de fixação (quiz). O objetivo foi buscar antecipar a maneira como os elementos visuais seriam dispostos ao jogador, de modo a facilitar a compreensão das mecânicas e do conteúdo didático.

4.3 IMPLEMENTAÇÃO

O jogo foi implementado na *engine* Godot utilizando o modelo de desenvolvimento incremental (Sommerville, 2011). O processo de implementação iniciou pelas funcionalidades básicas, como a construção do cenário, o controle geral do personagem e a ordenação utilizando o algoritmo *Bubble Sort*. Em seguida, o sistema evoluiu progressivamente para incorporar os outros algoritmos de ordenação (*Merge Sort* e *Radix Sort*) e a análise da complexidade assintótica, assegurando que, ao final, todos os requisitos funcionais e não funcionais fossem atendidos.

5 RESULTADOS

Este capítulo dedica-se à apresentação dos resultados obtidos com o desenvolvimento do jogo Sorteus. Serão mostradas as principais funcionalidades implementadas e as interfaces desenvolvidas, incluindo as telas iniciais do jogo (Seção 5.1), a aplicação prática dos conceitos de complexidade assintótica e dos algoritmos de ordenação *Bubble Sort* com critério de parada (Seção 5.2), *Merge Sort* (Seção 5.3), *Radix Sort LSD* (Seção 5.4) e telas finais do jogo (Seção 5.5).

5.1 APRESENTAÇÃO DAS TELAS INICIAIS

A Figura 5.1 exibe a tela inicial do jogo Sorteus, na qual é possível visualizar os botões "Iniciar Jogo", "Opções" e "Créditos".

Figura 5.1: Tela inicial do jogo Sorteus.

Ao clicar no botão "Opções" da Figura 5.1, será mostrado o menu com os botões "Ajuda" e "Configuração", como pode ser visto na Figura 5.2.

Figura 5.2: Menu com os botões “Ajuda” e “Configuração”.

Ao clicar no botão “Ajuda” da Figura 5.2, o jogador pode visualizar instruções sobre o jogo, como as teclas de movimentação e a interação com as portas, conforme apresentado na Figura 5.3.

Figura 5.3: Menu com informações de como jogar, após clicar no botão “Ajuda”.

Ao clicar no botão “Configuração” da Figura 5.2, é exibida a tela que permite ajustar o volume da música do jogo ou desativá-lo, como ilustrado na Figura 5.4.

Figura 5.4: Tela que permite ao jogador ajustar ou desativar o volume da música do jogo, após clicar no botão "Configuração".

Ao clicar no botão "Créditos" da Figura 5.1, são exibidas as informações das músicas utilizadas no jogo, como pode ser visto na Figura 5.5. A inclusão desta tela tem o intuito de garantir a conformidade com as licenças *Creative Commons* (CC BY 3.0 e 4.0) sob as quais as trilhas sonoras foram disponibilizadas, cumprindo assim o requisito obrigatório de atribuição de autoria aos compositores.

Figura 5.5: Tela com informações das músicas utilizadas no jogo, após clicar no botão "Créditos".

Por fim, ao clicar no botão "Iniciar Jogo" (Figura 5.1), o jogador é levado a uma residência formada por dois quartos, uma cozinha, duas salas e um banheiro (Figura 5.6). A história começa com o protagonista, Teus, conversando com sua mãe dentro de casa. Durante o diálogo, ela pede que Teus leve o medicamento que a avó esqueceu em sua residência. A avó de Teus mora do outro lado da vila, e, ao longo do trajeto, ele encontrará diferentes habitantes, cada um apresentando um desafio de ordenação e oferecendo oportunidades valiosas de aprendizagem.

Figura 5.6: Tela em que o personagem Teus recebe uma tarefa de sua mãe.

5.2 TELAS DE CONTEXTUALIZAÇÃO E EXECUÇÃO DO *BUBBLE SORT*

No início de sua jornada, Teus encontra seu pai (Figura 5.7), o dono da biblioteca da vila, que se disponibilizou a ajudar na reconstrução da ponte. Antes de partir, ele pede a Teus que auxilie José, o bibliotecário, a organizar um conjunto de livros que acabou de chegar na biblioteca.

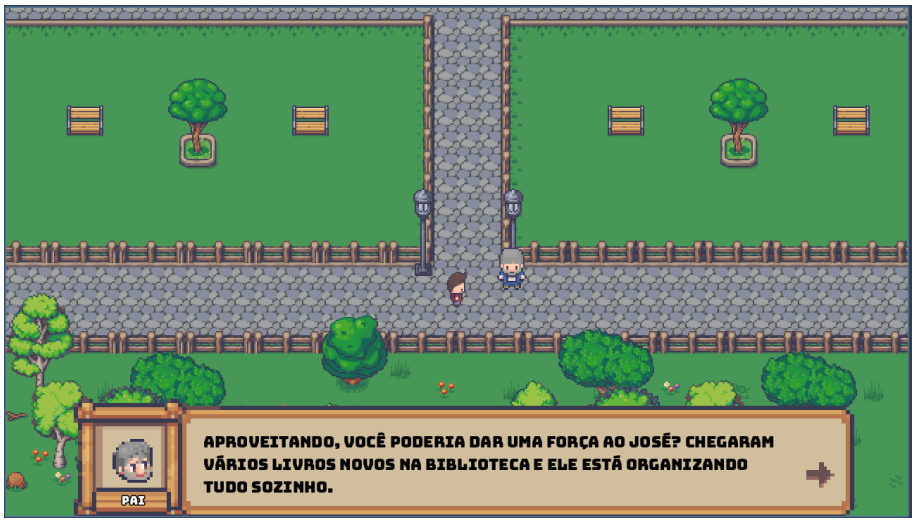


Figura 5.7: Encontro de Teus com seu pai.

Ao seguir em frente, Teus encontrará a ponte bloqueada (Figura 5.8) até que o jogador finalize os desafios de organizar os livros na biblioteca.

Figura 5.8: Ponte bloqueada até que o jogador complete os desafios de organizar os livros da biblioteca.

Atendendo ao pedido do pai, Teus dirige-se à biblioteca para ajudar José na organização dos livros recém-chegados (Figura 5.9). Durante a conversa, José explica que eles utilizam o algoritmo *Bubble Sort* como estratégia para ordenar os livros de forma crescente, de acordo com o número de páginas.

Figura 5.9: Encontro de Teus e José na biblioteca da vila.

Após a interação inicial com José, o jogador é redirecionado ao *minigame* do *Bubble Sort* (Figura 5.10), que é formado por duas fases: tutorial e aprendizado ativo. Na fase de tutorial, José ensina a execução do algoritmo *Bubble Sort*, com um conjunto de quatro livros, explicando passo a passo seu funcionamento. Além disso, no canto superior direito da tela, o jogador acompanha a contagem de comparações realizadas entre o número de páginas dos livros. Essa contagem será utilizada, posteriormente, na análise de desempenho e na apresentação da complexidade assintótica do algoritmo *Bubble Sort*.

Figura 5.10: Fase de tutorial no *minigame* do *Bubble Sort*.

Após finalizar a fase de tutorial, o jogador passa para a fase de aprendizado ativo, na qual é convidado a ordenar um conjunto de cinco livros utilizando o algoritmo *Bubble Sort* (Figura 5.11). Observe que os livros se encontram em ordem decrescente pelo número de páginas.

Figura 5.11: Fase de tutorial no *minigame* do *Bubble Sort*.

Na fase de aprendizado ativo, foram introduzidos dois novos elementos à interface: as vidas e a energia, localizadas no canto superior esquerdo da tela. As vidas indicam quantas vezes o jogador pode errar. Se cometer três erros, ele será redirecionado para a fase de tutorial para revisar os conceitos. A energia, por sua vez, representa o esforço do jogador para ordenar um conjunto de elementos em ordem crescente e funciona como uma analogia ao custo (esforço) do algoritmo para realizar a ordenação.

No contexto do *Bubble Sort*, a energia (esforço) do personagem diminui a cada comparação realizada. Conforme o nível de energia varia, o estado afetivo do personagem altera, podendo apresentar até três estados diferentes (Figura 5.12): rosto feliz (energia alta), rosto cansado (energia intermediária) e rosto triste (energia baixa). Além disso, o cansaço de Teus é enfatizado visualmente por meio de caixas de diálogo específicas que surgem em momentos de transição: a primeira ocorre ao atingir o nível de energia intermediário (Figura 5.13) e a segunda sinaliza o estado de exaustão ao alcançar o nível de energia baixa (Figura 5.14).

Figura 5.12: Nível de energia do jogador associado a três estados afetivos: felicidade, cansaço e tristeza.

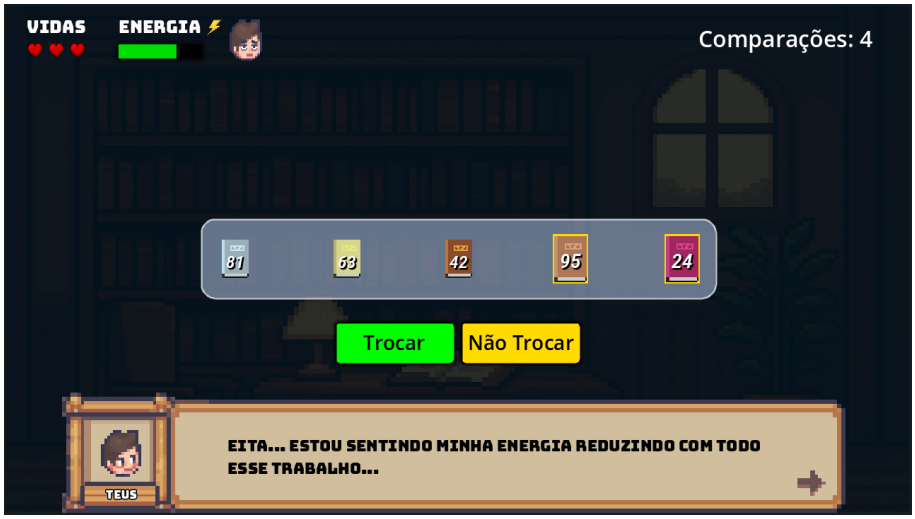


Figura 5.13: Caixa de diálogo apresentada ao atingir o nível de energia intermediário.

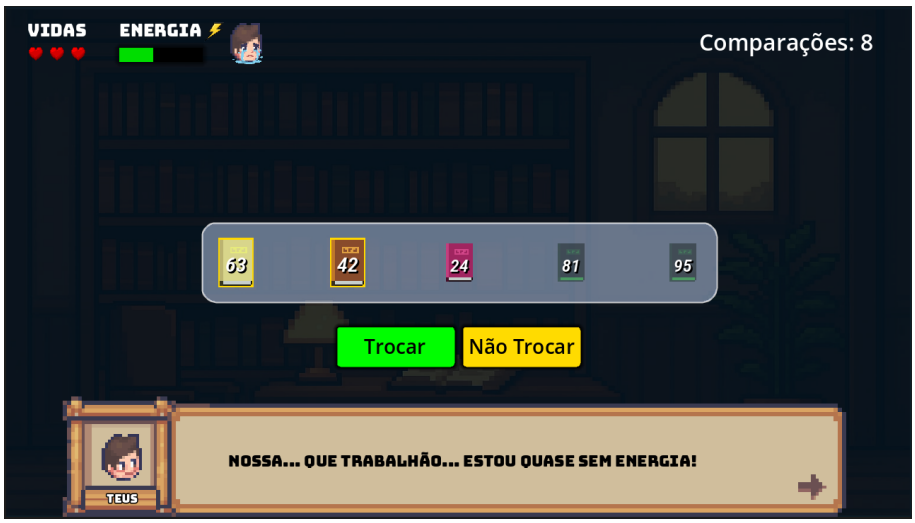


Figura 5.14: Caixa de diálogo apresentada ao atingir o nível de energia baixa.

Ainda na fase de aprendizado ativo, durante a execução do algoritmo *Bubble Sort*, o jogador deverá selecionar os livros de dois em dois para compará-los e decidir se devem ser trocados de posição ou não (Figura 5.15).

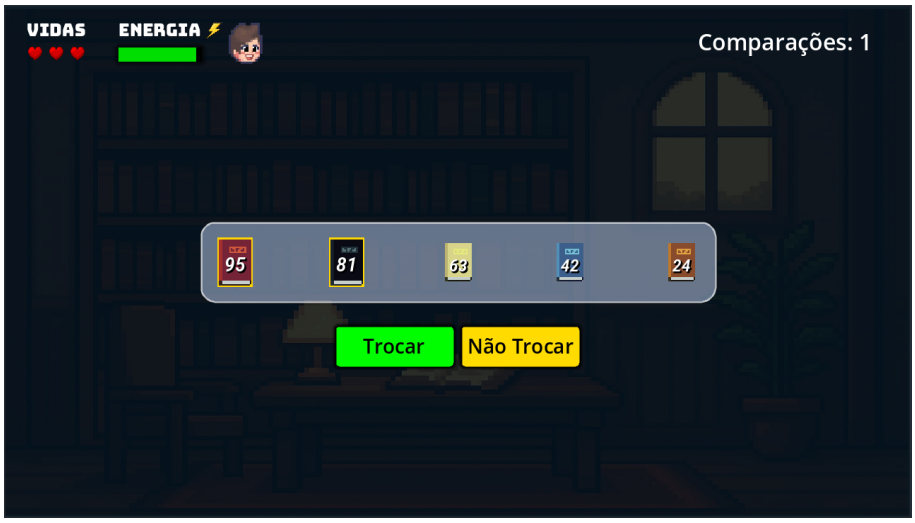


Figura 5.15: Comparação entre os livros no *minigame* do *Bubble Sort*.

A fase de aprendizado ativo também oferece *feedbacks* ao jogador por meio de caixa de diálogo, após a ocorrência de erros, conforme fala de José apresentada na Figura 5.16. No caso desse exemplo, o jogador clicou no botão "Não Trocar" da Figura 5.15, mas o correto seria trocar o livro com 95 páginas e o livro com 81 páginas. Note também que o jogador, após o erro, perdeu uma vida.

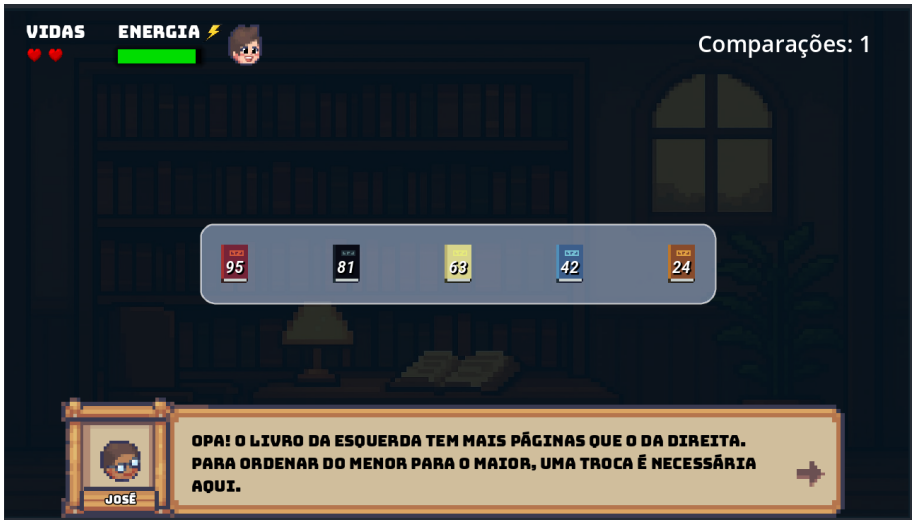


Figura 5.16: Mensagem de *feedback* em caso de erro na fase de aprendizado ativo do *Bubble Sort*.

Ao finalizar a ordenação do primeiro conjunto de livros, pode-se verificar que a energia do personagem ficou muito baixa devido à ineficiência do *Bubble Sort*, sobretudo por se tratar de um conjunto inicialmente em ordem decrescente (Figura 5.16), ou seja, representando o pior caso, conforme relata José na Figura 5.17.

Figura 5.17: Finalização do primeiro desafio: exemplo de pior caso do *Bubble Sort*.

Após concluir o primeiro desafio, o jogador é convidado a ordenar um novo conjunto de cinco livros, inicialmente dispostos na seguinte sequência: 24, 42, 95, 63 e 81. Esse segundo desafio corresponde a um caso intermediário do algoritmo *Bubble Sort*. Nesse caso, o jogador finaliza a ordenação com expressão de cansaço e energia baixa (Figura 5.18), enquanto no desafio anterior, ele termina com expressão de tristeza e energia muito baixa.

Figura 5.18: Finalização do segundo desafio: exemplo de caso intermediário do *Bubble Sort*.

Por fim, José apresenta um caso particular do *Bubble Sort* no qual o conjunto de livros já se encontra em ordem crescente, representando o melhor caso deste algoritmo (Figura 5.19).

Figura 5.19: Explicação do cenário de melhor caso do *Bubble Sort*.

Após a conclusão dos desafios, tem início a fase de pré-quiz. Inicialmente, José apresenta ao jogador informações sobre o comportamento do *Bubble Sort* com diferentes tipos de conjuntos e sobre o desempenho do algoritmo em cada cenário, considerando o número de comparações entre os elementos do conjunto (Figura 5.20).

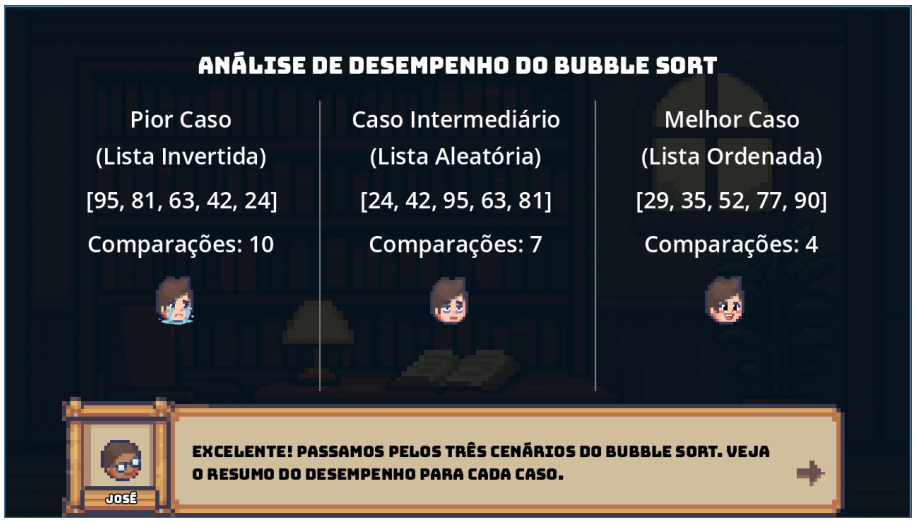


Figura 5.20: Fase de pré-quiz: José apresenta informações para avaliar o desempenho do *Bubble Sort*.

Depois disso, José mostra como o aumento da entrada de dados impacta no número de comparações do *Bubble Sort*, considerando o cenário de pior caso (Figura 5.21), e explica a complexidade assintótica (Figura 5.22), focando na notação *Big O*.

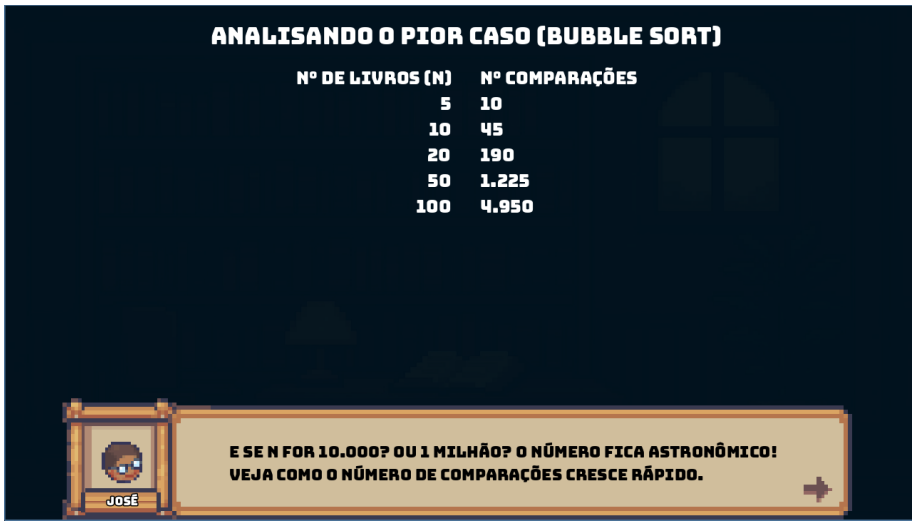


Figura 5.21: Fase de pré-quiz: José mostra o impacto do aumento da entrada para o número de comparações, no cenário de pior caso do *Bubble Sort*.

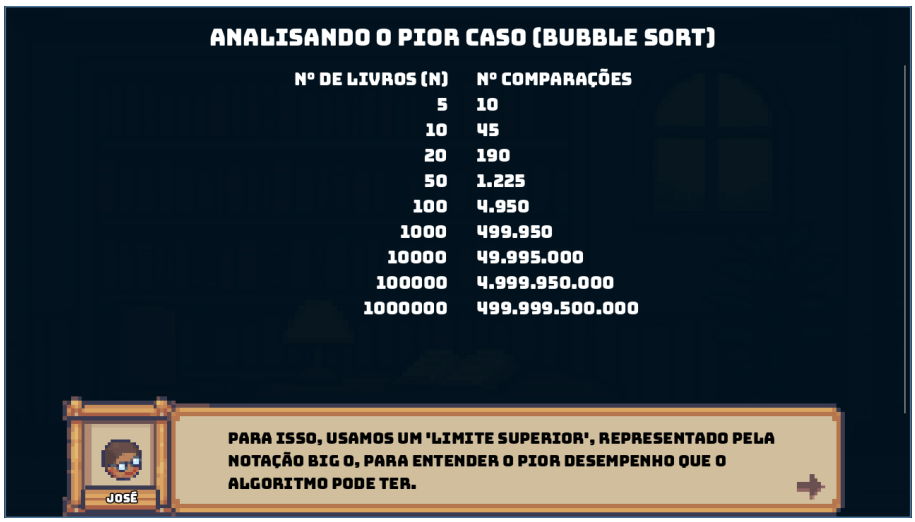


Figura 5.22: Fase de pré-quiz: José explica o conceito de complexidade assintótica focando na notação *Big O*.

Depois de explicar os conceitos de complexidade assintótica focados na notação *Big O*, José finaliza a fase de pré-quiz explicando a importância de se considerar o pior caso e mostrando um gráfico para reforçar o quão ruim fica o *Bubble Sort* para entradas muito grandes no cenário de pior caso (Figura 5.23).

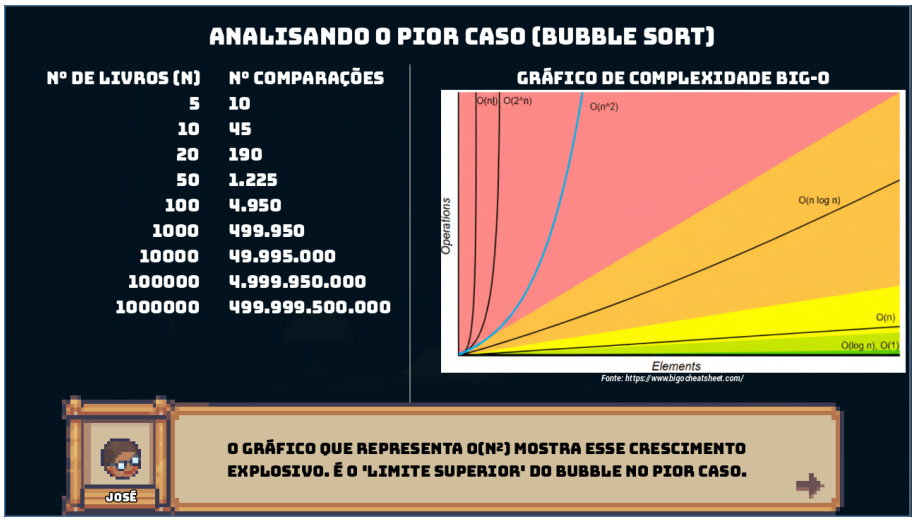


Figura 5.23: Fase de pré-quiz: José mostra um gráfico para reforçar o quão ruim é o *Bubble Sort* no cenário de pior caso para entradas muito grandes.

Após as explicações de José na fase de pré-quiz (Figura 5.23), o jogador deve clicar no botão “Iniciar Quiz” (Figura 5.24) que irá direcioná-lo à fase de quiz.

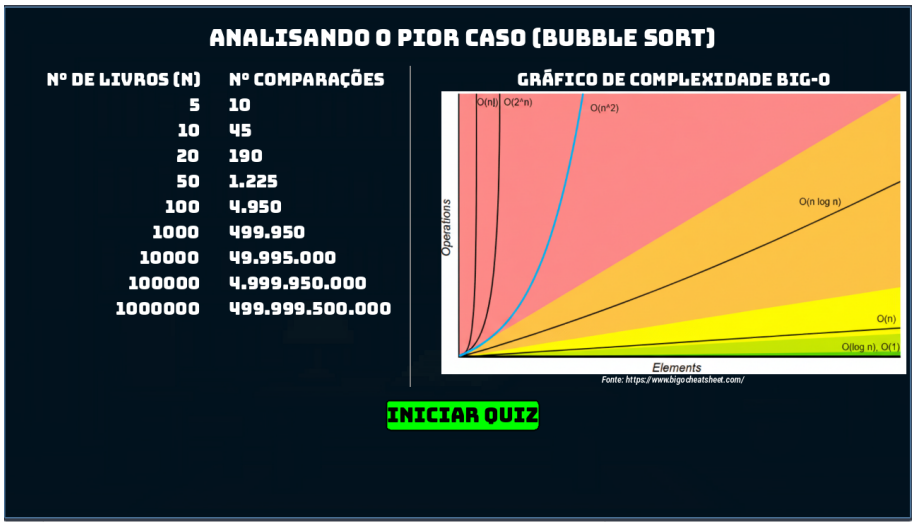


Figura 5.24: Tela que apresenta o botão “Iniciar Quiz”, o qual conduz o jogador a uma série de perguntas para avaliar os conhecimentos adquiridos.

O quiz é formado por cinco perguntas de múltipla escolha, cada uma contendo entre três a quatro alternativas. A cada resposta, o jogador recebe *feedback* imediato, seja em caso de acerto ou de erro. A Figura 5.25 apresenta um exemplo de *feedback* quando a resposta está correta e a Figura 5.26 quando está incorreta.

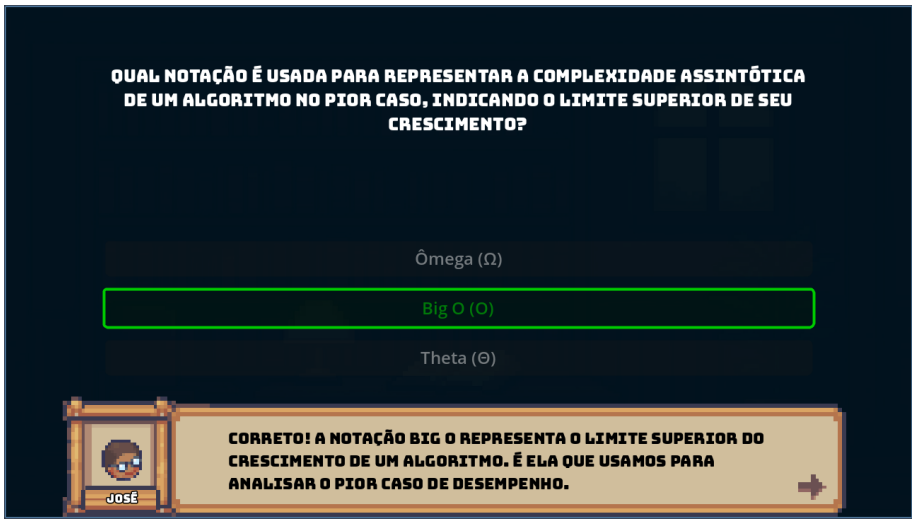


Figura 5.25: Exemplo de *feedback* para resposta correta no quiz.

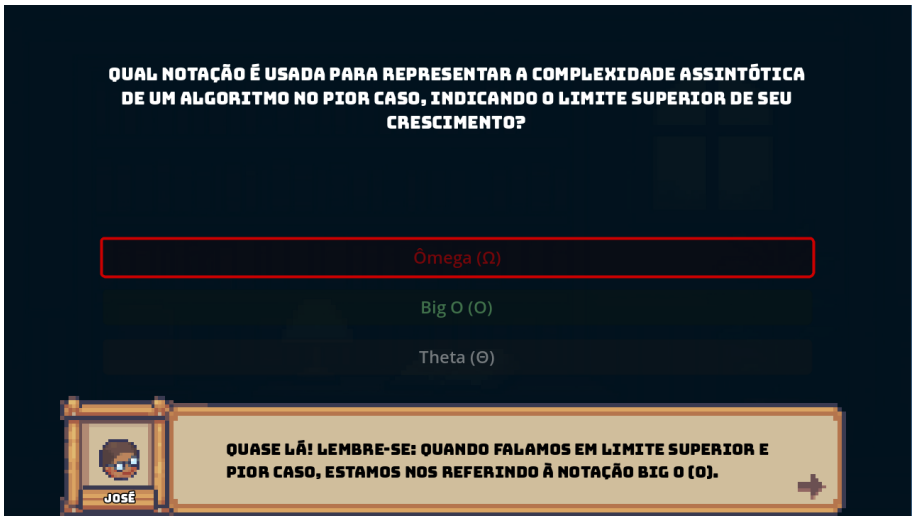


Figura 5.26: Exemplo de *feedback* para resposta incorreta no quiz.

Além da pergunta mostrada nas Figuras 5.25 e 5.26, a Tabela 5.1 exibe mais quatro perguntas do primeiro quiz. Para cada uma delas, são exibidas as alternativas corretas e incorretas. O objetivo dessas questões é ajudar o jogador a construir o entendimento sobre o conceito de complexidade assintótica e como ele é aplicado no contexto do *Bubble Sort*.

Ao final do primeiro quiz, o jogador retorna ao cenário da biblioteca. Nesse cenário, os livros que estavam espalhados pelo chão desapareceram, pois Teus ajudou José a organizá-los nas estantes (Figura 5.27).

Tabela 5.1: Quatro perguntas presentes no primeiro quiz do jogo Sorteus, além da pergunta apresentada nas Figuras 5.25 e 5.26.

Pergunta	**Alternativa correta**	**Alternativas incorretas**
O que representa a complexidade assintótica de um algoritmo?	Mostra como o desempenho do algoritmo se comporta quando a entrada de dados aumenta muito.	- Mostra como o desempenho do algoritmo se comporta quando a entrada de dados aumenta pouco. - Mostra como o desempenho do algoritmo se comporta quando a entrada nunca aumenta.
Na análise de algoritmos, o que significa dizer que a notação Big O fornece um limite superior?	Que ela mostra o pior desempenho possível do algoritmo.	- Que ela mostra o melhor desempenho possível do algoritmo. - Que ela mostra o desempenho exato do algoritmo.
Qual a complexidade assintótica do algoritmo *Bubble Sort* no pior caso?	$O(n^2)$ - Quadrático	- $O(logn)$ - Logarítmico - $O(n)$ - Linear - $O(n!)$ - Fatorial
Se tivermos uma lista muito grande (n) de itens e aplicarmos o *Bubble Sort*, dizer que sua complexidade assintótica é $O(n^2)$ significa que:	No pior caso, o número de comparações cresce no máximo de forma proporcional a n^2.	- No pior caso, o número de comparações será sempre maior que n^2. - No pior caso, o número de comparações será sempre exatamente igual a n^2.

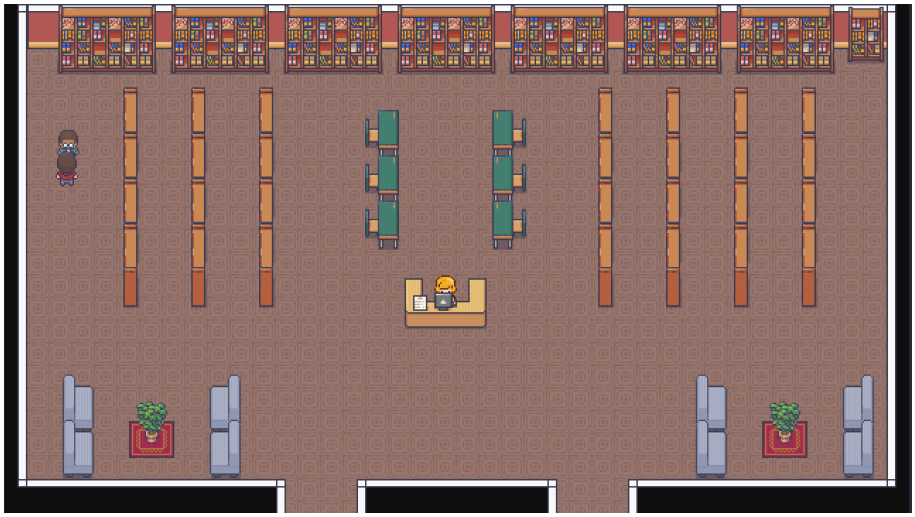


Figura 5.27: Cenário da biblioteca, sem os livros espalhados no chão, após Teus ajudar José a organizá-los.

Assim, conclui-se o cenário de contextualização e execução do algoritmo *Bubble Sort*. Nesse momento, a ponte é exibida sem bloqueios, indicando que foi consertada (Figura 5.28). Com isso, o jogador está pronto para avançar para a próxima fase do jogo, dedicada ao algoritmo *Merge Sort*.

Figura 5.28: Ponte sem bloqueio após conclusão do *minigame* do *Bubble Sort*.

5.3 TELAS DE CONTEXTUALIZAÇÃO E EXECUÇÃO DO *MERGE SORT*

Ao atravessar a ponte, o jogador chega a uma feira com diversos itens sendo vendidos, como frutas, pães e queijo. Nesse local, ocorre o encontro entre Teus e Maria, uma feirante preocupada com a grande quantidade de frutas que precisa organizar para colocar na bancada, em ordem crescente de peso. Com isso, ela pede a ajuda de Teus e diz que utiliza o *Merge Sort* no processo de ordenação (Figura 5.29).

Figura 5.29: Encontro entre Teus e Maria na feira.

Após o diálogo inicial, o jogador é direcionado ao *minigame* do *Merge Sort*. A primeira etapa é a de tutorial, em que Maria ensina o procedimento do *Merge Sort* para organizar um conjunto de frutas em ordem crescente de acordo com o peso.

Assim como no *Bubble Sort*, foi utilizada a métrica “Comparação” para possibilitar a análise de desempenho do algoritmo *Merge Sort* posteriormente. As Figuras 5.30 a 5.45 ilustram a sequência didática deste algoritmo no jogo, demonstrando desde a apresentação inicial dos elementos desordenados, passando pela aplicação da estratégia de divisão e conquista, até a conclusão com as frutas devidamente ordenadas pelo peso.

Figura 5.30: Início do tutorial do *Merge Sort*: as frutas aparecem desordenadas pelo peso.

Figura 5.31: Início do tutorial do *Merge Sort*: Maria apresenta a tática de ordenação conhecida como “Dividir para Conquistar”.

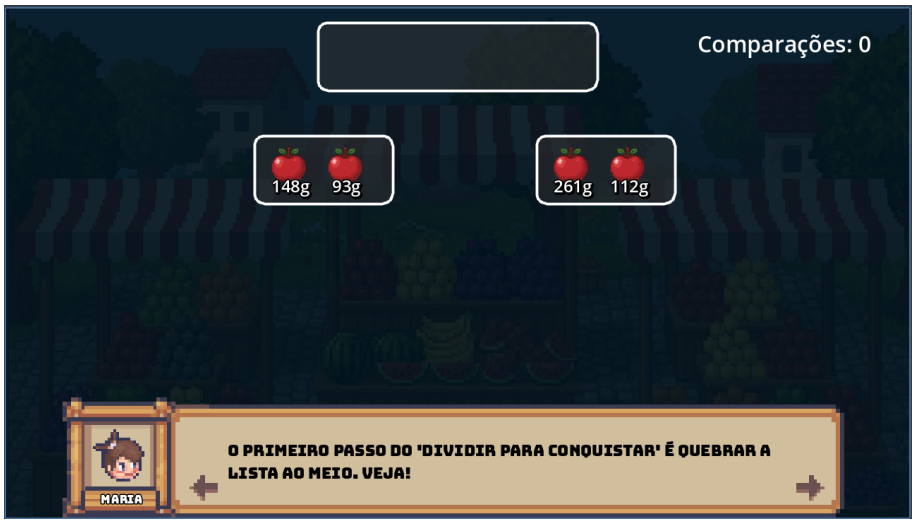


Figura 5.32: Fase de divisão do *Merge Sort*: Maria apresenta o primeiro passo da tática do “Dividir para Conquistar”, momento em que ocorre a divisão inicial.

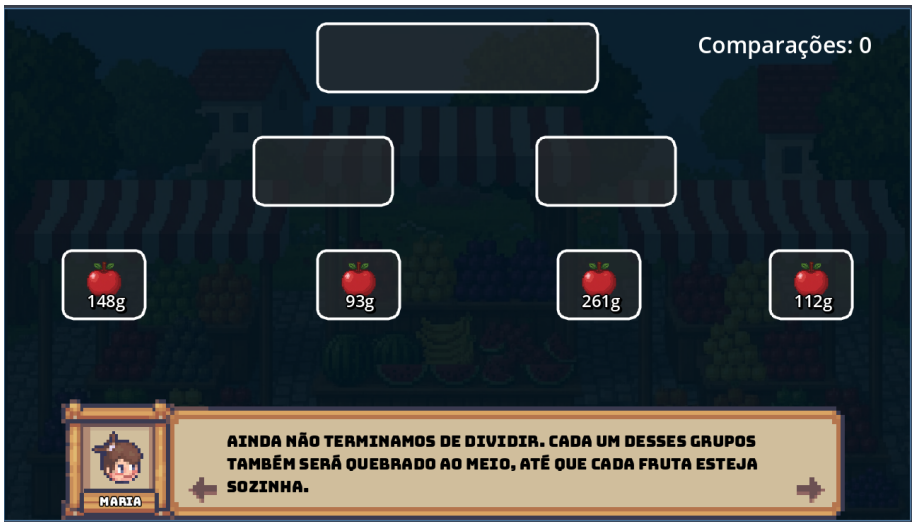


Figura 5.33: Fase de divisão do *Merge Sort*: Maria exibe a conclusão do processo de separação, tornando cada fruta um elemento independente.

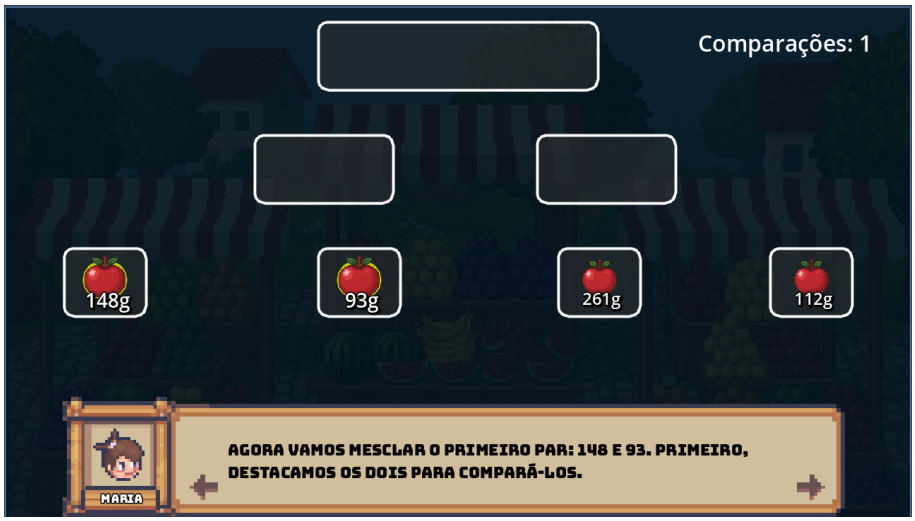


Figura 5.34: Fase de conquista do *Merge Sort*: Maria começa comparando as frutas dos grupos mais à esquerda, no caso, as frutas de peso 148g e 93g

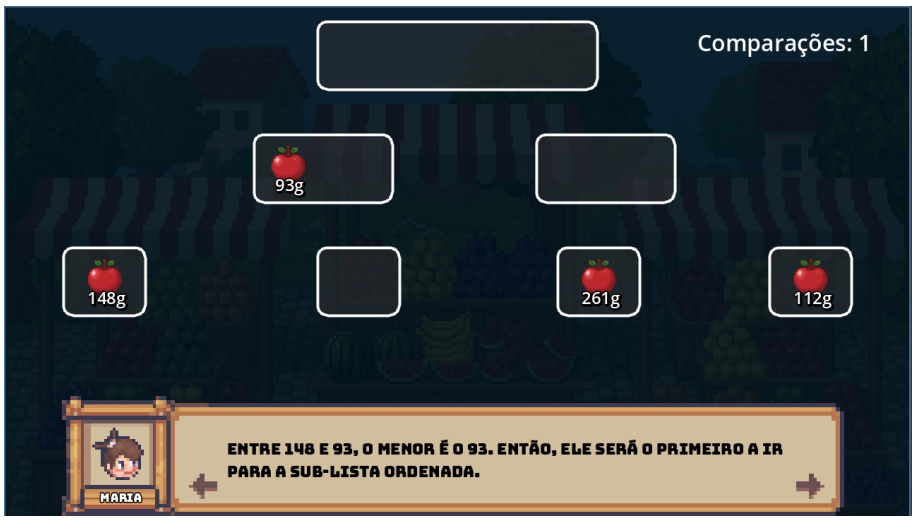


Figura 5.35: Fase de conquista do *Merge Sort*: Após a comparação inicial, o elemento de 93g é selecionado para compor a sub-lista ordenada, dado que seu valor é inferior ao de 148g.

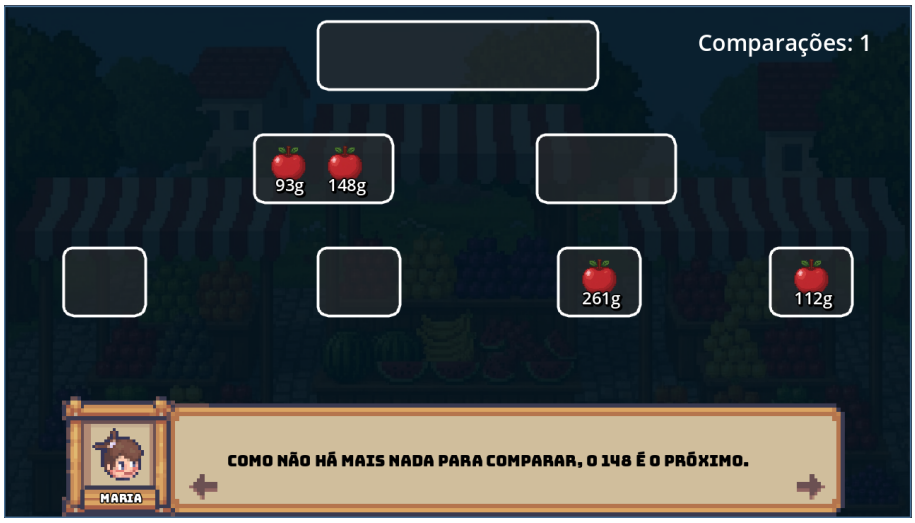


Figura 5.36: Fase de conquista do *Merge Sort*: A fruta remanescente (148g) é transferida para a sub-lista ordenada, visto que o grupo adjacente não possuía frutas para comparação.

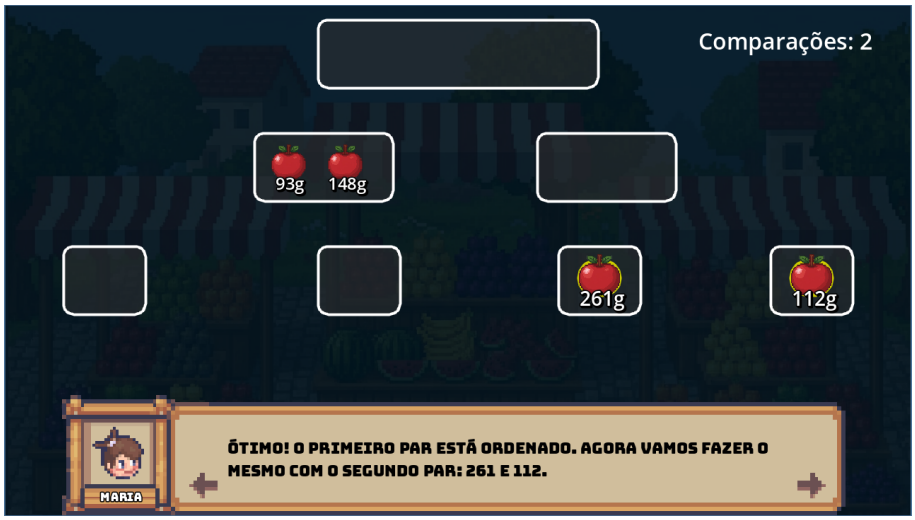


Figura 5.37: Fase de conquista do *Merge Sort*: Após a organização das sub-listas à esquerda, o foco desloca-se para a direita, no qual Maria realiza a comparação inicial entre as frutas de pesos 261g e 112g.

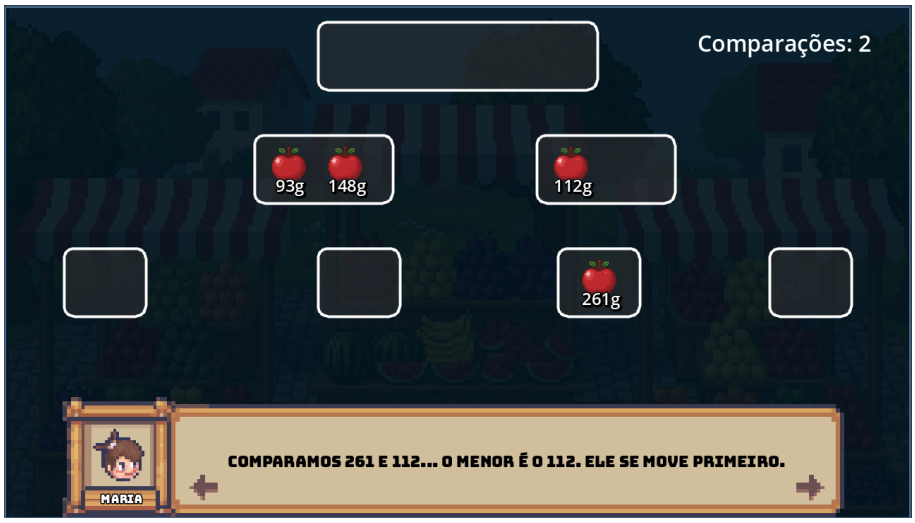


Figura 5.38: Fase de conquista do *Merge Sort*: Após a comparação, o elemento de 112g é selecionado para compor a sub-lista ordenada, dado que seu valor é inferior ao de 261g.

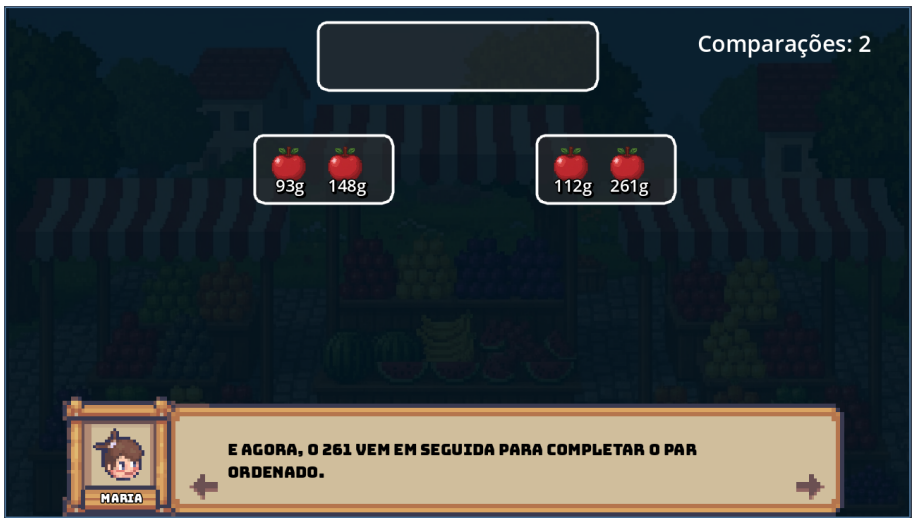


Figura 5.39: Fase de conquista do *Merge Sort*: A fruta remanescente (261g) é transferida para a sub-lista ordenada, visto que o grupo adjacente não possuía frutas para comparação.

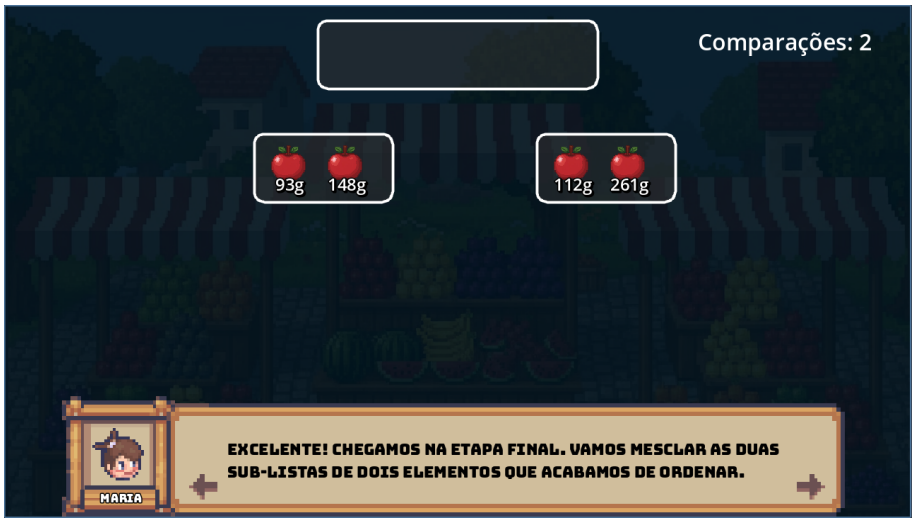


Figura 5.40: Etapa final da fase de conquista do *Merge Sort*: Com as sub-listas intermediárias já ordenadas, inicia-se o processo de mesclagem final para obter o conjunto de frutas ordenadas de maneira crescente por peso.

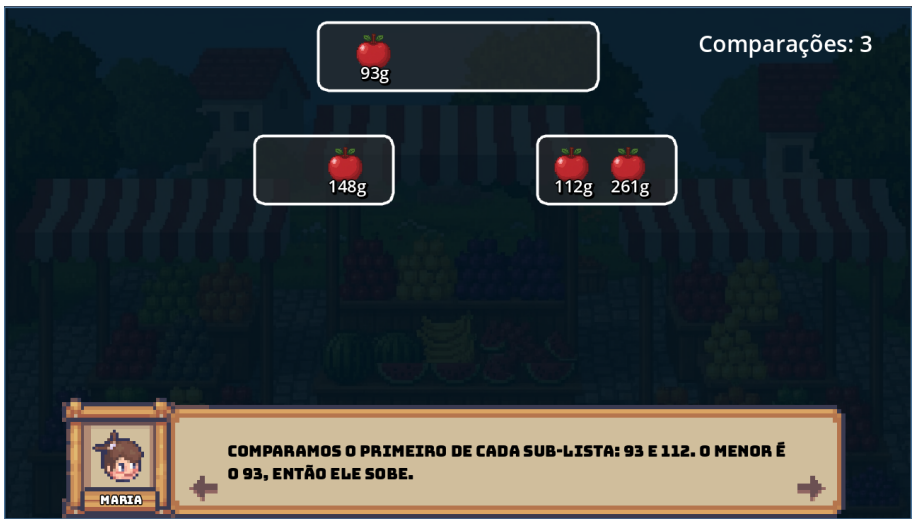


Figura 5.41: Etapa final da fase de conquista do *Merge Sort*: Maria compara os primeiros itens de cada grupo. A fruta de 93g é transferida para a lista ordenada primeiro, dado que seu valor é inferior ao de 112g.

Figura 5.42: Etapa final da fase de conquista do *Merge Sort*: Maria compara os primeiros itens remanescentes de cada grupo, que agora são as frutas de pesos 148g e 112g. A fruta de 112g é transferida para a lista ordenada, dado que seu valor é inferior ao de 148g.

Figura 5.43: Etapa final da fase de conquista do *Merge Sort*: Maria compara os primeiros itens remanescentes de cada grupo, que agora são as frutas de pesos 148g e 261g. A fruta de 148g é transferida para a lista ordenada, dado que seu valor é inferior ao de 261g.

Figura 5.44: Etapa final da fase de conquista do *Merge Sort*: A fruta remanescente (261g) é transferida para a lista ordenada, visto que o grupo adjacente não possuía frutas para comparação.

Figura 5.45: Fim da execução do *Merge Sort*, com Maria apresentando as frutas ordenadas pelo peso.

Após finalizar a fase de tutorial, o usuário é convidado a ordenar um conjunto de quatro frutas (Figura 5.46), utilizando a mesma estratégia de "Divisão e Conquista" apresentada no tutorial. Da mesma maneira que no *Bubble Sort*, a energia do personagem no *minigame* do *Merge Sort* e as vidas são apresentadas no canto superior esquerdo da tela. Além disso, a energia diminui a cada comparação realizada e as vidas são reduzidas quando o jogador comete algum erro. Para iniciar o processo de ordenação, o jogador deverá clicar no botão "Dividir Lista".

Figura 5.46: Fase de aprendizado ativo no *minigame* do *Merge Sort*: primeiro desafio.

Nessa fase de aprendizado ativo, o usuário seleciona as frutas de cada grupo, da esquerda para a direita, e decide qual delas será direcionada para a sub-lista ordenada. Dessa forma, o usuário executa progressivamente a etapa de conquista até que as frutas estejam ordenadas por peso de forma crescente. Ao finalizar o primeiro desafio, Maria pede que o jogador ordene um novo conjunto de frutas, $\{148, 93, 261, 112, 305, 54\}$, utilizando o mesmo procedimento.

Após a fase de aprendizagem ativa, o jogador é encaminhado à fase de pré-quiz. Nessa fase serão apresentadas as informações sobre o desempenho do algoritmo de ordenação *Merge Sort* (Figura 5.47), comparando os desafios com quatro frutas (Desafio 1) e com seis frutas (Desafio 2), que resultaram em cinco e dez comparações, respectivamente. Essa visualização, aliada ao diálogo da Maria, busca instigar o jogador a relacionar o aumento da entrada com o crescimento do custo computacional.

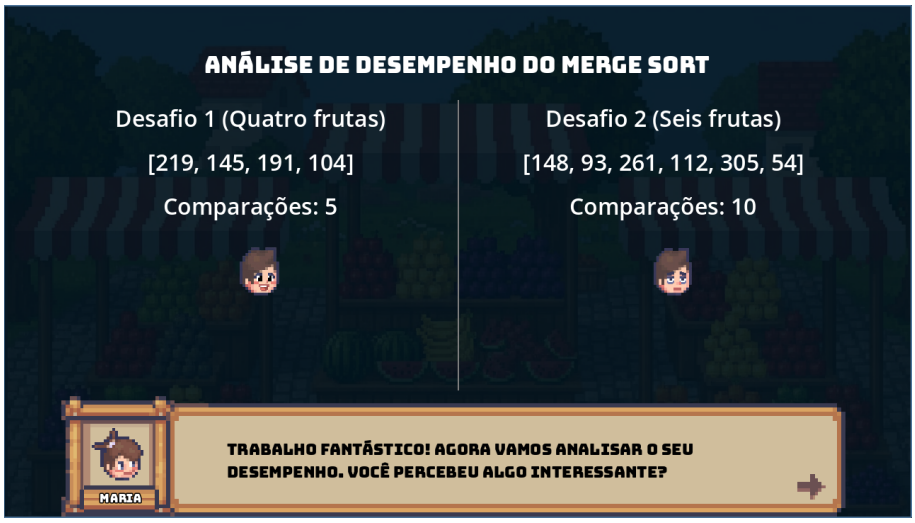


Figura 5.47: Fase de pré-quiz do *Merge Sort*: início da análise de desempenho.

Na Figura 5.48, Maria mostra como o aumento da quantidade de dados de entrada impacta o número de comparações realizadas pelo *Merge Sort* no cenário de pior caso. Na explicação, ela compara o desempenho do algoritmo *Bubble Sort* para a entrada $n = 128$, ressaltando o quanto o *Merge Sort* é mais eficiente.

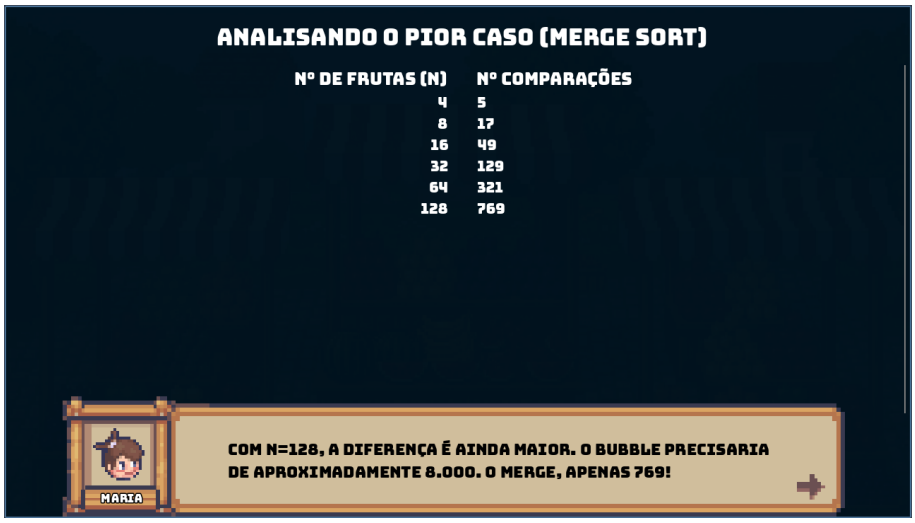


Figura 5.48: Fase de pré-quiz: Maria mostra como o aumento do tamanho da entrada afeta o número de comparações no pior caso do *Merge Sort*.

Além disso, na Figura 5.49, Maria apresenta a complexidade assintótica do algoritmo *Merge Sort*, utilizando a notação *Big O*.

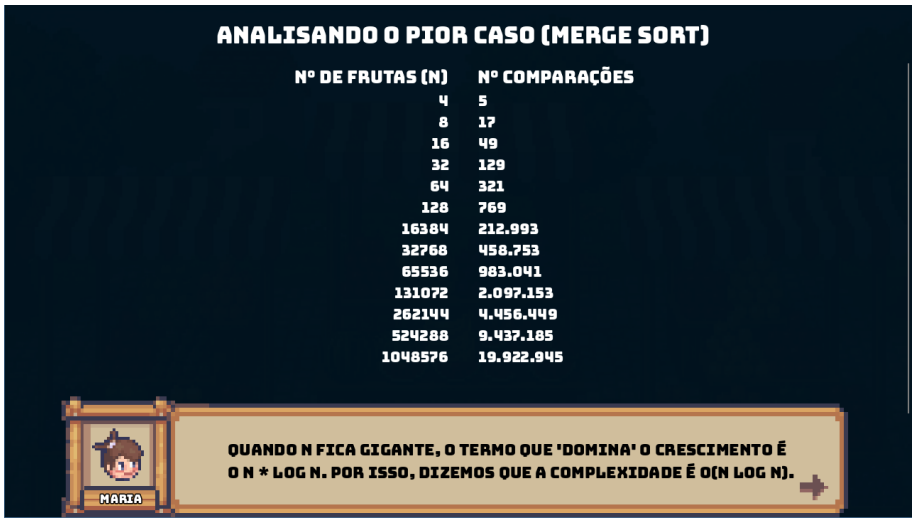


Figura 5.49: Fase de pré-quiz: Maria apresenta a complexidade assintótica do *Merge Sort* utilizando a notação *Big O*.

Depois de explicar os conceitos de complexidade assintótica focados na notação *Big O*, Maria finaliza a fase de pré-quiz mostrando um gráfico para reforçar o quão mais eficiente o *Merge Sort* é em relação ao *Bubble Sort* para entrada grande de dados (Figura 5.50).

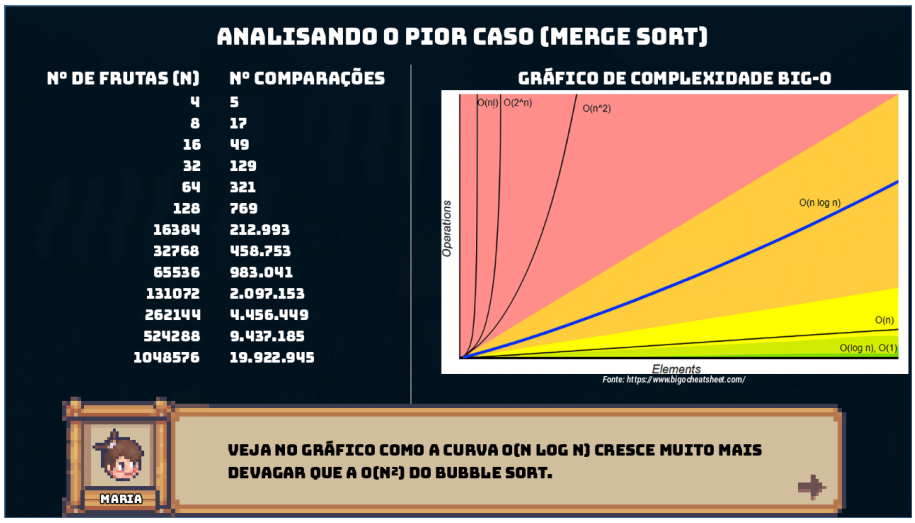


Figura 5.50: Fase de pré-quiz: Maria exibe um gráfico para reforçar que o *Merge Sort* é mais eficiente que o *Bubble Sort* para grandes entradas.

Após a conclusão da fase de pré-quiz, o botão “Iniciar Quiz” é exibido na tela, permitindo que o jogador avance para a fase de quiz, cuja proposta é testar os conhecimentos adquiridos durante a prática dos desafios. Na fase de quiz, o jogador deverá responder cinco perguntas de múltipla escolha, cada uma com três ou quatro alternativas. Para cada pergunta, o jogador receberá *feedback*, tanto no caso de acerto, quanto no caso de erro, conforme ilustrado nas Figuras 5.51 e 5.52.

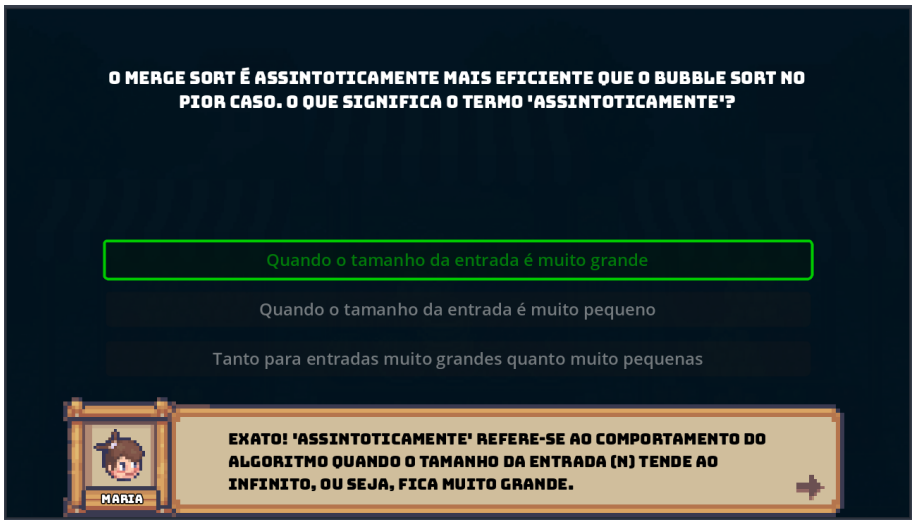


Figura 5.51: Fase de quiz: exemplo de *feedback* para resposta correta sobre o *Merge Sort*.

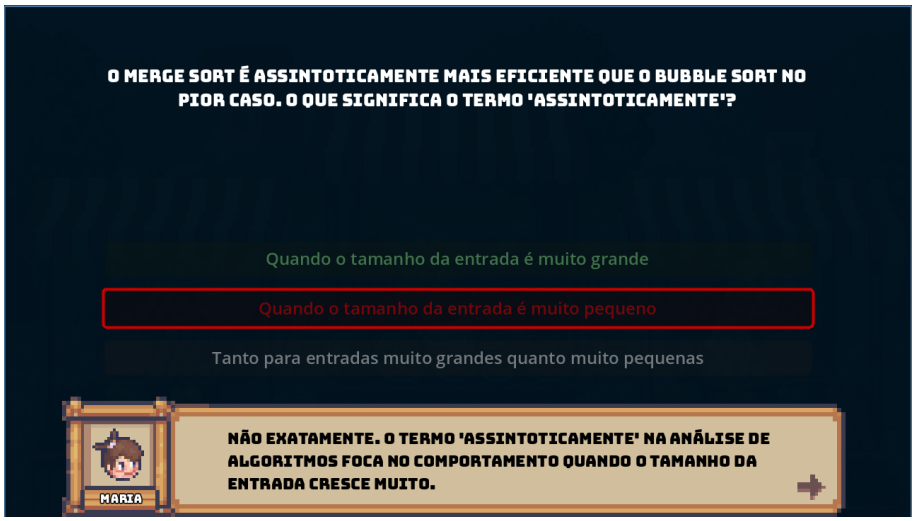


Figura 5.52: Fase de quiz: exemplo de *feedback* para resposta incorreta sobre o *Merge Sort*.

Além da pergunta mostrada nas Figuras 5.51 e 5.52, a Tabela 5.2 exibe mais quatro perguntas para a fase de quiz do *Merge Sort*. Para cada uma delas, são exibidas as alternativas corretas e incorretas.

Tabela 5.2: Quatro perguntas presentes no segundo quiz do jogo Sorteus, além da pergunta apresentada nas Figuras 5.51 e 5.52.

Pergunta	**Alternativa correta**	**Alternativas incorretas**
O *Merge Sort* é assintoticamente mais eficiente que o *Bubble Sort* no pior caso. O que isso significa?	Para entradas grandes, o *Merge Sort* realiza menos comparações no pior caso.	- Para entradas grandes, o *Bubble Sort* realiza menos comparações no pior caso. - Para entradas grandes, ambos realizam o mesmo número de comparações no pior caso.
O *Merge Sort* é mais eficiente que o *Bubble Sort* para listas grandes. Qual é sua complexidade no pior caso?	$O(nlogn)$	- $O(n!)$ - $O(n)$ - $O(n^2)$
Se tivermos uma lista muito grande (n) de itens e aplicarmos o *Merge Sort*, dizer que sua complexidade assintótica é $O(nlogn)$ significa que:	No pior caso, o número de comparações cresce no máximo de forma proporcional a $nlogn$.	- No pior caso, o número de comparações será sempre exatamente igual a $nlogn$. - No pior caso, o número de comparações será sempre maior que $nlogn$.
Você viu que o *Merge Sort* quebra a lista em pedaços menores e depois os junta. Qual é o nome dessa estratégia de algoritmo?	Divisão e Conquista	- Estratégia Gulosa - Programação Dinâmica - Força Bruta

Ao final da fase de quiz do *Merge Sort*, o jogador sai do contexto do *minigame* e retorna ao mundo principal. No cenário da feira, é possível perceber que as frutas agora estão dispostas na bancada da Maria, após Teus tê-la ajudado a organizá-las por peso (Figura 5.53).

Figura 5.53: Cenário da feira com as frutas dispostas na bancada da Maria.

Assim, conclui-se o cenário de contextualização e execução do algoritmo *Merge Sort*, permitindo que Teus siga adiante em sua jornada (Figura 5.54). Ao sair da praça, ele se prepara para explorar a fase final do jogo dedicada ao algoritmo *Radix Sort*.

Figura 5.54: Teus em direção à saída leste da praça.

5.4 TELAS DE CONTEXTUALIZAÇÃO E EXECUÇÃO DO *RADIX SORT*

Ao sair da praça, Teus encontra Jaiminho, o carteiro da vila, que foi atacado pelos cachorros da Dona Xica que estavam soltos, e acabou derrubando todas as cartas que carregava, deixando-as completamente bagunçadas. Jaiminho pede a ajuda de Teus para organizá-las e explica que gostaria de mostrá-lo como realizar essa ordenação utilizando o método *Radix Sort* (Figura 5.55).

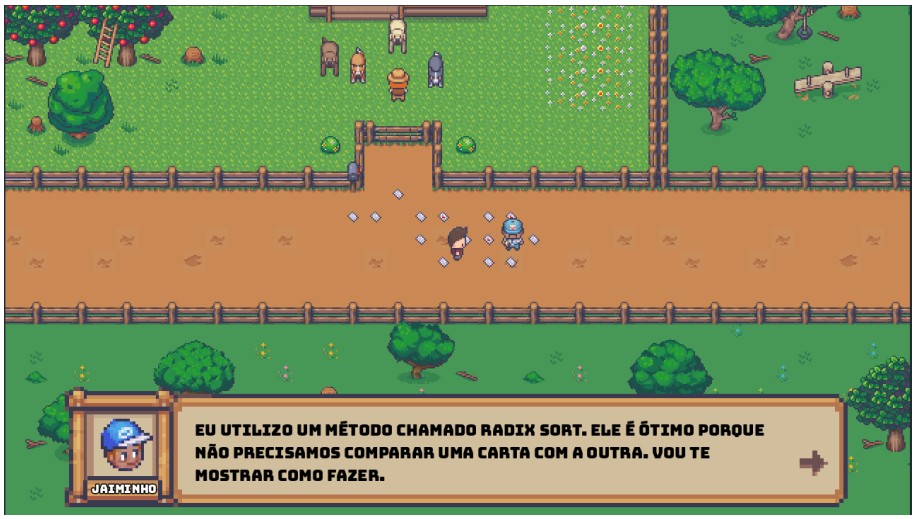


Figura 5.55: Encontro entre Teus e o carteiro da vila, Jaiminho, antes de iniciar o *minigame* do *Radix Sort*.

Após o diálogo inicial, o jogador inicia o *minigame* do *Radix Sort*. Na fase de tutorial, Jaiminho ensina a Teus como aplicar o *Radix Sort* para ordenar um conjunto de quatro cartas em ordem crescente, de acordo com o número da residência mostrada nas cartas. Diferente dos métodos de ordenação *Bubble Sort* e *Merge Sort*, no *Radix Sort* foi utilizada a métrica "Movimento" para análise de desempenho, tendo em vista que esse método não é baseado em comparação (Figura 5.56).

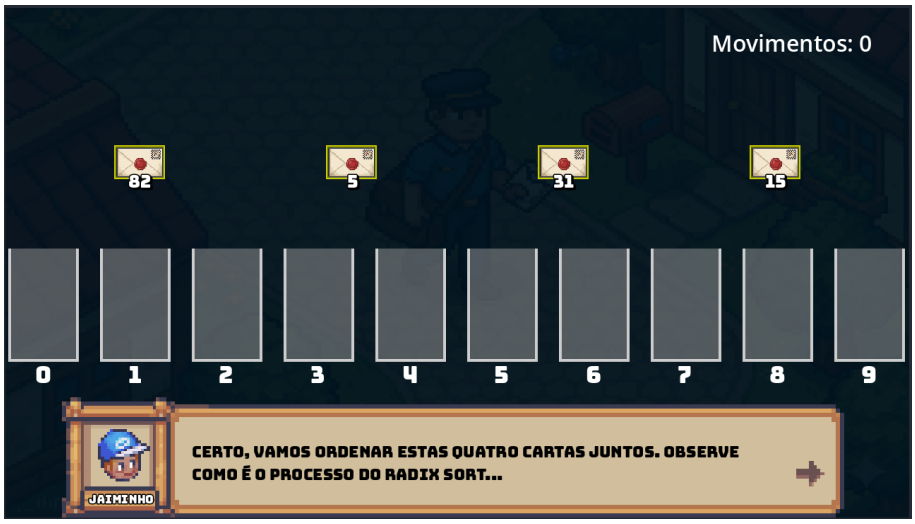


Figura 5.56: Fase de tutorial no *minigame* do *Radix Sort*.

Ao usar o método de ordenação *Radix Sort*, os elementos (cartas) são analisados da esquerda para a direita, começando pelo algarismo da unidade. Com isso, cada carta é colocada no "balde" correspondente ao algarismo que está sendo analisado. No caso da Figura 5.57, como o número de residência 82 possui o algarismo 2 na casa das unidades, ele é direcionado para o balde 2. O mesmo ocorre com os demais elementos, ou seja, o 5 vai para o balde 5, o 31 vai para o balde 1 e o 15 vai para o balde 5.

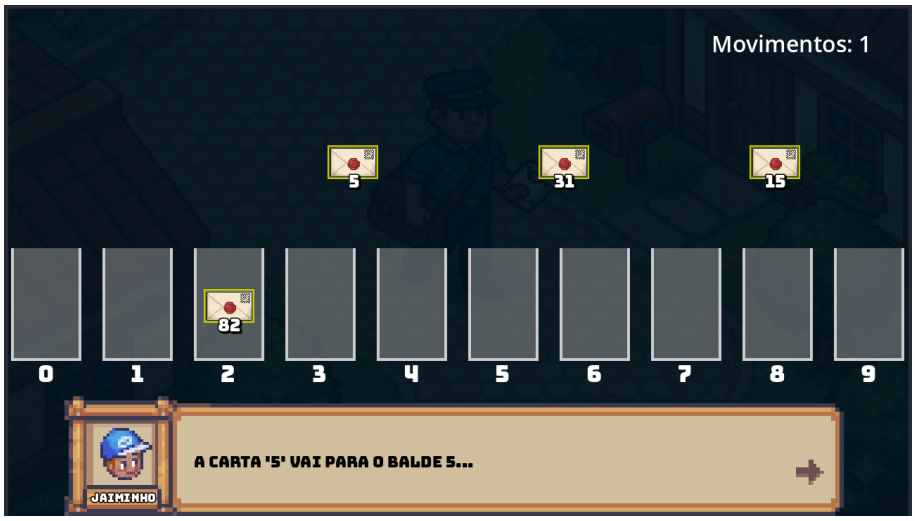


Figura 5.57: Fase de tutorial no *minigame* do *Radix Sort*: iniciando o processo de ordenação das cartas.

Após todas as cartas serem colocadas nos baldes (Figura 5.58)[1], o próximo passo consiste em recolhê-las começando pelo balde de menor valor e mantendo a mesma ordem em que foram inseridas em cada um deles, ou seja, primeiro a carta com o número de residência 31 é retirada, seguida da 82, depois 5 e, por fim, 15 (Figura 5.59). Dessa forma, preserva-se a estabilidade do algoritmo.

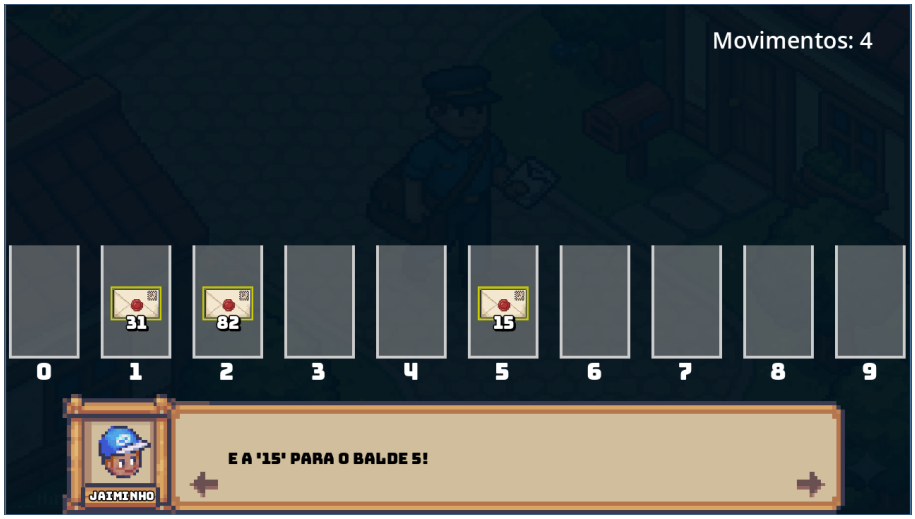


Figura 5.58: Fase de tutorial no *minigame* do *Radix Sort*: todas as cartas estão em seus devidos baldes durante a primeira iteração.

[1]A carta com número 15 foi adicionada depois da carta com valor 5 e, por isso, a de número 5 não está sendo mostrada no balde.

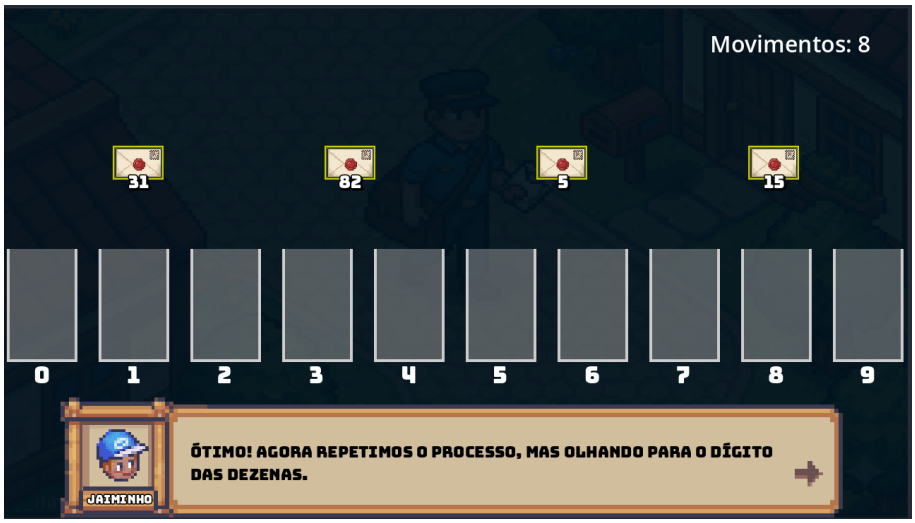


Figura 5.59: Fase de tutorial no *minigame* do *Radix Sort*: retirada das cartas dos baldes após a primeira iteração.

Na segunda iteração, o dígito analisado será o da casa das dezenas. Com isso, caso uma carta não tenha esse dígito, o mesmo será considerado com valor zero. Por exemplo, a carta com o número de residência 5 só possui o dígito das unidades, logo o dígito da dezena nesse caso será computado como zero.

Conforme mostrado na Figura 5.60, após repetir o processo de inserir as cartas nos baldes e recolhê-las, obtém-se a lista final de cartas ordenadas de maneira crescente.

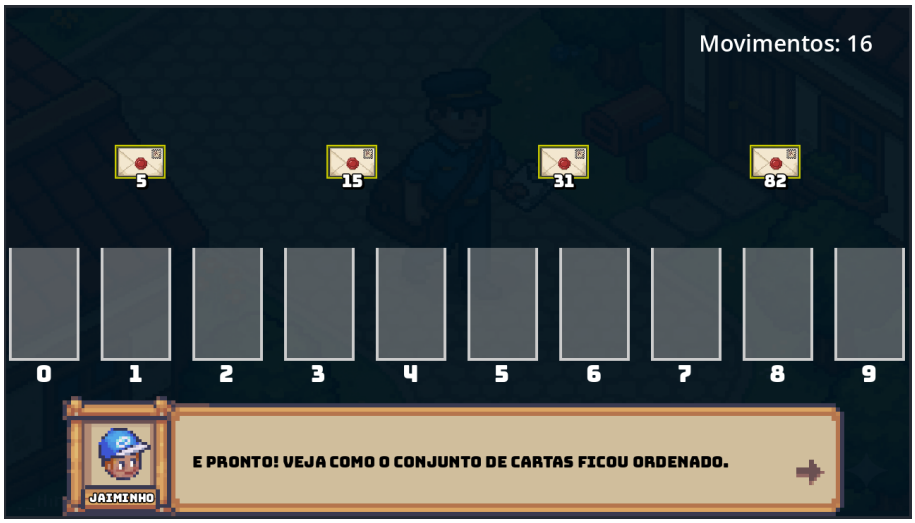


Figura 5.60: Fim da fase de tutorial no *minigame* do *Radix Sort*: conjunto de cartas ordenadas de maneira crescente.

Ao final da fase de tutorial, o jogador inicia a fase de aprendizado ativo e é convidado a ordenar um conjunto de seis cartas seguindo o mesmo procedimento apresentado no tutorial. No contexto do *Radix Sort*, a energia do personagem diminui a cada movimento realizado pelo jogador e o número de vidas é reduzido sempre que um movimento incorreto é realizado (Figura 5.61).

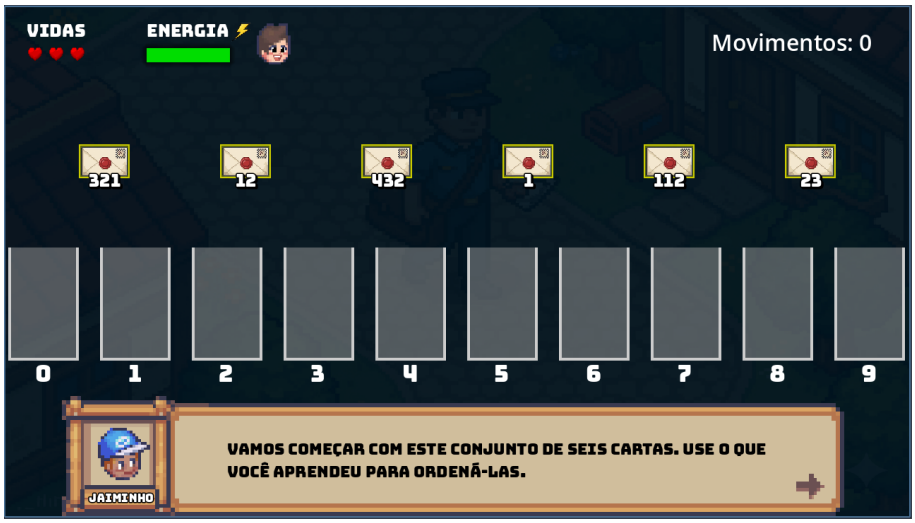


Figura 5.61: Fase de aprendizado ativo no *minigame* do *Radix Sort*: ordenação de seis cartas.

Na fase de aprendizado ativo, assim como mostrado no tutorial, o jogador deve colocar as cartas nos baldes correspondentes de acordo com o dígito analisado. O processo de coleta é automático e resulta, ao final da última iteração, nas seis cartas dispostas em ordem crescente (Figura 5.62).

Figura 5.62: Fase de aprendizado ativo no *minigame* do *Radix Sort*: fim da ordenação das seis cartas.

Após ordenar as seis cartas, Jaiminho solicita que o jogador realize a ordenação de um novo conjunto, agora com oito cartas. O procedimento é o mesmo realizado anteriormente. Com isso, a fase de aprendizado ativo é concluída e tem início a fase de pré-quiz do *Radix*, em que Jaiminho apresenta ao jogador informações sobre o desempenho e a complexidade do *Radix Sort* (Figura 5.63).

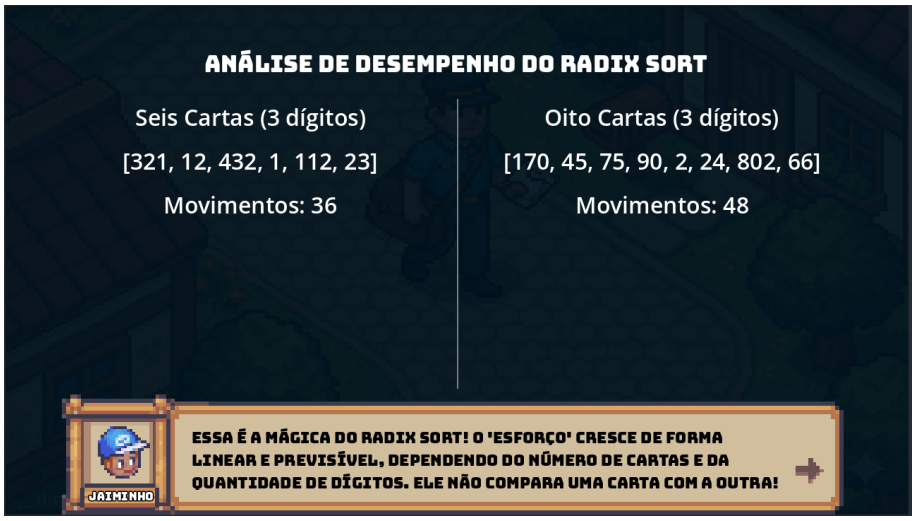


Figura 5.63: Início da fase de pré-quiz do *minigame Radix Sort*: Jaiminho apresenta informações sobre o desempenho e a complexidade do algoritmo.

Ainda na fase de pré-quiz, Jaiminho mostra ao jogador como o aumento da entrada de dados (número da residência) tem impacto no número de movimentos do *Radix Sort*, considerando o cenário de pior caso (Figura 59).

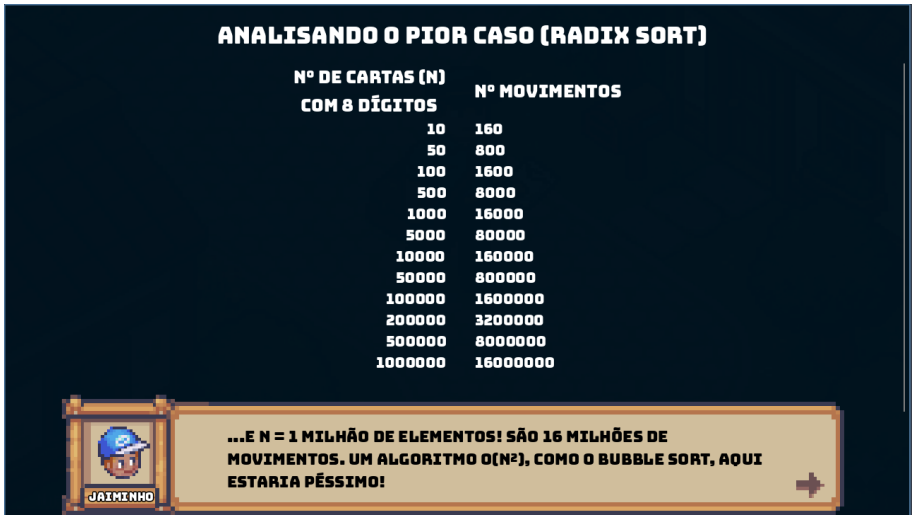


Figura 5.64: Fase de pré-quiz: Jaiminho mostra ao jogador como o aumento do número da residência tem impacto no número de movimentos, no cenário de pior caso do *Radix Sort*.

Para finalizar a fase de pré-quiz, Jaiminho exibe um gráfico para que o jogador possa visualizar o quão eficiente é o algoritmo de ordenação *Radix Sort* (Figura 60).

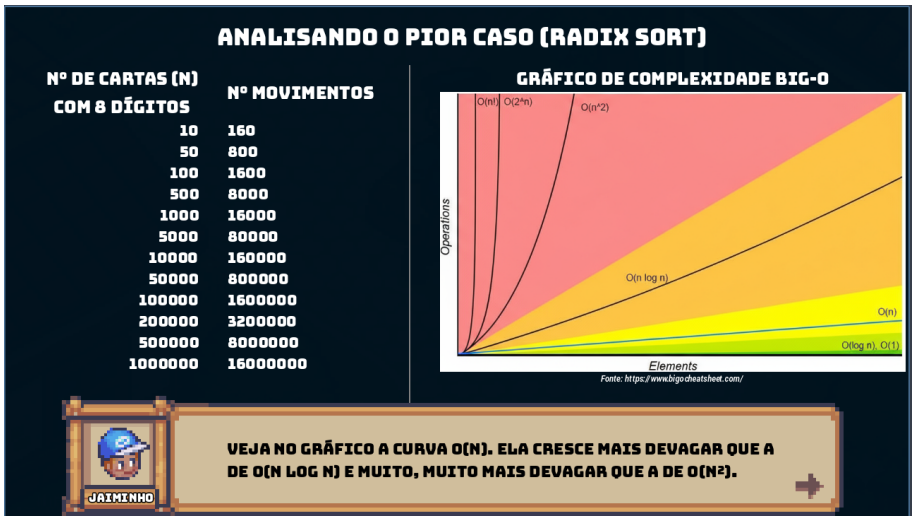


Figura 5.65: Fase de pré-quiz: Jaiminho exibe um gráfico mostrando o quão eficiente é o *Radix Sort* no cenário de pior caso para entradas muito grandes.

Ao finalizar a fase de pré-quiz, o jogador deve clicar no botão de “Iniciar Quiz” para ser direcionado à fase de quiz. Nessa fase serão apresentadas três perguntas de múltipla escolha com três ou quatro alternativas, sendo a Figura 5.66 referente à resposta correta e a Figura 5.67 à resposta incorreta.

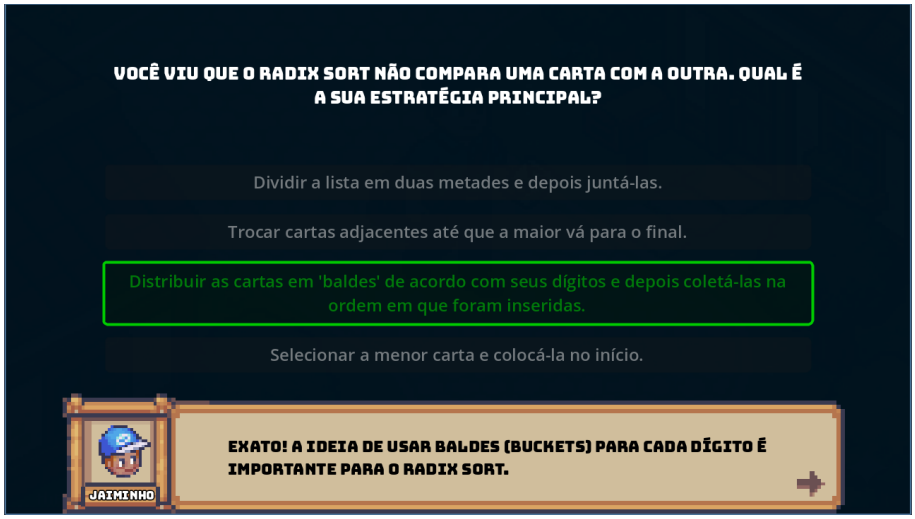


Figura 5.66: Exemplo de *feedback* para resposta correta no quiz.

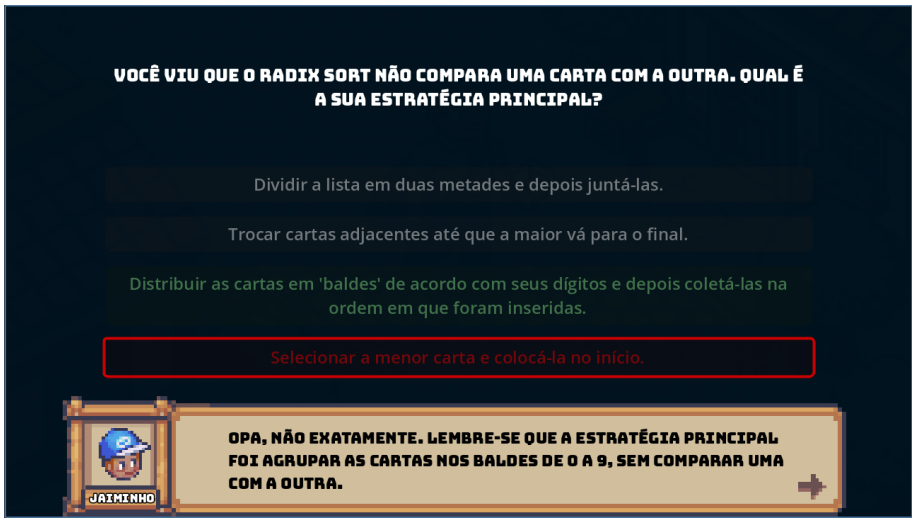


Figura 5.67: Exemplo de *feedback* para resposta incorreta no quiz.

Além da pergunta mostrada nas Figuras 5.66 e 5.67, a Tabela 5.3 exibe mais duas perguntas da fase de quiz do *Radix Sort*. Para cada uma delas, são exibidas as alternativas corretas e incorretas. O objetivo dessas questões é ajudar o jogador a construir o entendimento sobre o conceito de complexidade assintótica e como ele é aplicado no contexto do *Radix Sort*.

Tabela 5.3: Perguntas presentes na fase de quiz do *Radix Sort*, além da pergunta apresentada nas Figuras 5.66 e 5.67.

Pergunta	**Alternativa correta**	**Alternativas incorretas**
Na análise, você viu que o número de movimentos cresceu de forma previsível (linear) com o número de cartas. Qual é a complexidade assintótica do *Radix Sort*?	$O(dn)$ - Linear em relação à quantidade de dígitos e de elementos	- $O(n^2)$ - Quadrático - $O(nlogn)$ - Log-linear - $O(1)$ - Constante
Na análise de algoritmos, o que significa dizer que a notação Big O fornece um limite superior?	Que ela mostra o pior desempenho possível do algoritmo.	- Que ela mostra o melhor desempenho possível do algoritmo. - Que ela mostra o desempenho exato do algoritmo.

Após finalizar a fase de quiz do *Radix Sort*, o jogador retorna ao cenário do encontro de Teus com o carteiro Jaiminho. Nesse cenário, as cartas que estavam espalhadas pelo chão desaparecem, pois Teus ajudou Jaiminho a organizá-las (Figura 5.68).

Figura 5.68: Cenário de encontro entre Teus e Jaiminho: as cartas não estão mais espalhadas no chão após ajuda de Teus.

Assim, conclui-se a etapa de contextualização e execução do algoritmo *Radix Sort*, permitindo que Teus avance para a etapa final de sua missão.

5.5 TELAS FINAIS DO JOGO SORTEUS

Após o encontro com o carteiro, Teus segue caminhando até encontrar sua avó na área externa da residência, conforme ilustrado na Figura 5.69.

Figura 5.69: Teus encontra sua avó.

A avó de Teus explica que se dirigia à casa dele para buscar o remédio, pois o horário de tomar sua medicação estava próximo. Agradecida pela atitude do neto, ela o abraça e o convida para tomar um café e comer um bolo recém-feito. Teus aceita o convite, encerrando sua jornada no jogo, conforme mostra a Figura 5.70. Nesta tela, encontra-se também o botão "Voltar", responsável por redirecionar o jogador ao menu principal.

Figura 5.70: Fim da jornada de Teus.

6 CONCLUSÃO

Este trabalho teve como objetivo apresentar o jogo educacional Sorteus, de forma a auxiliar no ensino de algoritmos de ordenação com foco no conceito de complexidade assintótica. A principal contribuição desta pesquisa consiste na disponibilização de um jogo educacional, que introduz o jogador aos fundamentos básicos de complexidade assintótica, além de explicar o funcionamento dos algoritmos de ordenação *Bubble Sort*, *Merge Sort* e *Radix Sort*.

Além disso, a mecânica do jogo (ex.: energia do jogador), em conjunto com as fases de pré-quiz e quiz, foi planejada para apoiar tanto a construção quanto a consolidação dos conceitos introdutórios da notação assintótica *Big O*, permitindo ao jogador compreender as diferenças de desempenho entre os algoritmos *Bubble Sort*, *Merge Sort* e *Radix Sort* no contexto de pior caso. Vale ressaltar que os custos de energia foram ponderados para evidenciar as diferenças de desempenho que seriam imperceptíveis para poucos elementos. Assim, configurou-se uma escala onde as ações do *Bubble Sort* foram as mais custosas e as do *Radix Sort* as mais econômicas, com o *Merge Sort* situando-se em um nível intermediário.

Como trabalhos futuros, sugere-se a validação do jogo em turmas de graduação, como Ciência da Computação, para verificar tanto a parte pedagógica quanto a usabilidade do sistema. Este trabalho pode ser ampliado com a inclusão de novos algoritmos de ordenação, como o *Heap Sort*, para a classe de ordenação por comparação, e o *Counting Sort*, para a classe de algoritmos de tempo linear; além da portabilidade para dispositivos móveis, visando ampliar o acesso ao jogo.

REFERÊNCIAS

Battistella, P. E., Petri, G., von Wangenheim, C. G., von Wangenheim, A. e Martina, J. E. (2016). Sortia 2.0: Um jogo de ordenação para o ensino de estrutura de dados. Em *Anais do XII Simpósio Brasileiro de Sistemas de Informação (SBSI)*, páginas 558–565, Florianópolis, Brasil. Sociedade Brasileira de Computação.

Bhargava, A. Y. (2017). *Entendendo algoritmos: um guia ilustrado para programadores e outros curiosos*. Novatec.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. e Stein, C. (2012). *Algoritmos: Teoria e Prática*. Elsevier.

Fritsch, H. F., Silva, E. M. e Souza, J. G. (2016). Ordena - um jogo educacional para auxílio ao ensino de métodos de ordenação. Em *Anais do ENCOINFO - Congresso de Computação e Tecnologias da Informação*, páginas 123–131, Palmas, TO. CEULP/ULBRA.

Fürlinger, J. (2024). An educational browser game about sorting algorithms.

Goldman, S. A. e Goldman, K. J. (2008). *A practical guide to data structures and algorithms using Java*. Chapman & Hall/CRC.

Liu, Y. A. (2013). *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press.

Raia, M. L., Martins, A. G., Kas, G. P. A. e Eliseo, M. A. (2023). Fruitsort: the educational computational thinking game with accessibility for hearing-impaired children. Em *2023 18th Iberian Conference on Information Systems and Technologies (CISTI)*, páginas 1–6.

Rolim, I., Coelho, A. e Duarte, J. C. (2024). Isle sort: Um jogo voltado para o aprendizado de algoritmos de ordenação e busca. Em *Anais do 13.º Concurso Apps.Edu - Categoria Protótipo - Congresso Brasileiro de Informática na Educação (CBIE)*, páginas 277–280, Rio de Janeiro, RJ. Sociedade Brasileira de Computação.

Roughgarden, T. (2017). *Algorithms Illuminated: Part 1 - The Basics*. Soundlikeyourself Publishing.

Savi, R. e Ulbricht, V. R. (2008). Jogos digitais educacionais: benefícios e desafios. *RENOTE*, 6(1).

Sedgewick, R. e Wayne, K. (2011). *Algorithms*. Addison-Wesley.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Prentice Hall.

Szwarcfiter, J. L. e Markezon, L. (2010). *Estruturas de dados e seus algoritmos*. LTC.