**Relatório dissertativo da Atividade Avaliativa 5**

**(26/09/2025)**

**Nome:** Davi Gabriel Domingues, Pedro Martins de Oliveira

**N° USP:** 15447497, 13696213

**Objetivo do documento:** comparar o desempenho dos códigos produzidos, a partir do exercício “Entrega 04”, do dia 26/09/2025, com as devidas especificações. No caso, será discutido o desempenho final observado (notação “Big O”), além dos pontos nos quais foram pertinentes ao desenvolvimento do código, para a resolução da situação problema em si.

**Versão Final)**

Temos a seguinte versão:

Texto

O conteúdo gerado por IA pode estar incorreto.

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

Texto

O conteúdo gerado por IA pode estar incorreto.

Interface gráfica do usuário, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

Texto

O conteúdo gerado por IA pode estar incorreto.

**Discussão técnica:** O algoritmo lida com uma situação mais extensa dessa vez, já que deve tratar a ordenação das informações dos brinquedos (cor, comprimento, nota e id), a partir de quatro perspectivas, ou seja, algoritmos, sendo eles, respectivamente enumerados de 1 a 4: InsertionSort, BubbleSort, MergeSort e QuickSort. Essas metodologias de ordenação podem ter seus desempenhos lidados separadamente, dado às suas construções diferenciadas, sendo, então notório destacar o seguinte:  
**I) InsertionSort:**

Tem seu funcionamento embasado no princípio da inserção do elemento — denominado *chave* — em uma posição apropriada para, justamente, ordenar o vetor. A cada iteração o algoritmo extrai essa chave (brinquedo[i]) do vetor e a insere na posição correta dentro do subvetor já ordenado 0..i-1. Em vez de procurar essa posição elemento a elemento, a rotina usa busca binária (com os índices inicio, fim, meio e a variável posInsercao) para decidir rapidamente em qual metade do subvetor a chave deve ficar; ao localizar posInsercao realiza-se um deslocamento prévio do bloco de elementos à direita dessa posição para abrir espaço e então a chave é escrita. O invariante útil para raciocinar sobre o algoritmo é: ao iniciar a iteração com índice i, o subvetor 0..i-1 já está ordenado — essa propriedade é preservada a cada passo.

Quanto ao desempenho, percebe-se que a busca binária reduz o número de comparações necessárias para localizar a posição de inserção — para cada elemento a busca custa O(log i), resultando em O(n log n) comparações no total. Contudo, após encontrar posInsercao ainda é preciso deslocar os elementos do subvetor para a direita, e esses deslocamentos custam linearmente no tamanho do bloco deslocado; somando todos os deslocamentos ao longo do algoritmo o custo de movimentação é O(n²). Assim, embora haja uma melhoria nas comparações graças ao comportamento logarítmico da busca, o tempo total do algoritmo permanece O(n²) devido às movimentações.

**II) BubbleSort:**

Tem seu funcionamento baseado no princípio de varreduras sucessivas por pares adjacentes: o algoritmo percorre o vetor comparando vizinhos e os trocando quando estão na ordem errada, de modo que valores maiores “sobem” em cada varredura e valores menores “descem”. No código foi adotada a variação chamada ShakeSort (ou *Cocktail Shaker Sort*), que alterna varreduras da esquerda para a direita e da direita para a esquerda usando as variáveis inicio, fim e a flag trocou. Na varredura da esquerda para a direita o maior elemento do intervalo atual é empurrado até fim; na varredura da direita para a esquerda o menor elemento é empurrado até inicio. Após cada par de varreduras, os limites inicio e fim são atualizados, reduzindo o intervalo ativo. O invariante que facilita o raciocínio é: ao término de cada varredura completa (ida e volta), todos os elementos fora do intervalo inicio..fim já estão nas posições finais corretas.

Quanto aos impactos e ao desempenho, o ShakeSort tenta amortizar o comportamento do bubble sort clássico ao mover os extremos em ambas as direções, o que pode reduzir o número de passadas necessárias em entradas onde grandes valores estão no começo e pequenos valores no fim. Além disso, a flag trocou torna o algoritmo *adaptativo* — se uma passada não produzir nenhuma troca, o algoritmo interrompe cedo, alcançando custo O(n) no melhor caso (vetor já ordenado). Ainda assim, no pior caso (ex.: vetor inversamente ordenado) tanto comparações quanto trocas continuam na ordem de O(n²), porque cada elemento ainda pode precisar ser trocado muitas vezes. Em termos práticos, o ShakeSort frequentemente reduz constantes (menos passadas efetivas) e pode ser mais rápido que o bubble sort em alguns padrões de dados, mas não altera a classe de complexidade assintótica no pior caso. Uma das propriedades adicionais relevantes ao algoritmo está relacionada com o fato dele ser in-place (uso de memória extra O(1)).

**III) MergeSort:**

**IV) QuickSort:**

Tabela

O conteúdo gerado por IA pode estar incorreto.

Tempo de execução dos 5 casos padrão do runcodes (casos gerais/médios)

Obs¹: nos cinco casos de teste definidos pela interface do runcodes, o primeiro é para o InsertionSort, o segundo para o BubbleSort, o terceiro para o MergeSort, por fim, o quarto e o quinto remetem ao QuickSort.

Obs²: as implementações dos algoritmos InsertionSort , BubbleSort, e MergeSort são, naturalmente estáveis, menos o QuickSort. Porém, nessa situação, a função compararBrinquedos foi definida, justamente, para delimitar claramente os critérios de igualdade e a inserção por deslocamento e/ou troca adequada, preservando a ordem relativa dos elementos equivalentes, garantindo, dessa maneira, que elementos semelhantes mantenham sua ordem original.