**Exercício prático individual**: Tempo de execução de algoritmos

**Aluno**: Luiz Felipe Vieira

1. Hardware:

**Nome do sistema operacional**: Microsoft Windows 11 Home

**Versão do sistema operacional**: 10.0.22621 N/A compilação 22621

**Tipo de sistema**: x64-based PC

**Processador(es)**: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz

**Memória física total**: 16.215 MB

1. Resultados obtidos (MergeSort e QuickSort):





Tanto o QuickSort quanto o MergeSort exibem complexidade de tempo O(n \* log n), geralmente analisada em termos de comparações entre elementos nesses algoritmos. Embora a literatura sugira que o MergeSort seja superior ao QuickSort para volumes de dados mais elevados, **meus resultados mostraram que o QuickSort teve um desempenho superior em termos de tempo**.

Ambos os algoritmos demonstraram comportar-se conforme o esperado. Conforme o tamanho da entrada de dados aumentou, observamos um aumento correspondente no tempo de execução, na quantidade de operações e nas trocas de elementos. Essa relação é um **indicativo de que os algoritmos são estáveis**.

É importante notar que o aumento percentual no tempo de execução é um pouco maior do que o aumento na entrada de dados, o que está alinhado com a complexidade O(n \* log n), como esperado.

**Ao validar a complexidade O(n \* log n) com entradas de tamanho n, observamos valores razoáveis que se aproximam das comparações teóricas**. No entanto, é importante ressaltar que o desempenho dos algoritmos na prática pode variar um pouco em relação ao que a literatura descreve, devido a fatores específicos da implementação e do ambiente de execução.

Além da análise de tempo, é importante considerar o uso de espaço pelos algoritmos. **O QuickSort tende a ser mais eficiente em termos de espaço quando comparado ao MergeSort**. Isso ocorre porque o QuickSort é um algoritmo de ordenação "in-place", o que significa que ele não requer espaço adicional significativo para armazenar dados temporários durante a ordenação.

Por outro lado, o MergeSort é um algoritmo "out-of-place", o que implica que ele geralmente requer espaço adicional para armazenar os novos arrays durante o processo de ordenação. Em situações onde a memória disponível é um recurso crítico, o QuickSort pode ser preferível devido ao seu uso mais eficiente de espaço.

1. Resultados obtidos QuickSort (ordenado e não ordenado):



A análise do desempenho do algoritmo QuickSort revelou diferenças significativas entre sua eficiência ao lidar com vetores ordenados e vetores com dados aleatórios. **Para vetores com elementos embaralhados, o QuickSort demonstrou um desempenho que se aproximou da complexidade teórica esperada, que é O(n \* log n)**. No entanto, ao lidar com **vetores ordenados, o algoritmo atingiu um desempenho próximo do pior caso, que é O(n²)**. Isso ocorre devido à escolha do pivô, que desempenha um papel crucial quando os dados já estão parcialmente ou totalmente ordenados.

O QuickSort opera selecionando um elemento como pivô e rearranjando o vetor de modo que os elementos menores que o pivô fique à esquerda e os elementos maiores fiquem à direita. Quando o pivô é consistentemente escolhido como o primeiro elemento (ou o último, que é comum em muitas implementações), e o vetor já está ordenado, o QuickSort realiza muitas comparações e trocas redundantes.

Essa ineficiência ocorre porque, em um vetor ordenado, o pivô sempre será o menor ou o maior elemento, dependendo da escolha do pivô. Isso resulta em partições desbalanceadas, levando a múltiplas chamadas recursivas com apenas um elemento à esquerda ou à direita do pivô, o que cria a complexidade quadrática.