Relatório de Desempenho de Algoritmos de Ordenação

Vinicius Bittencourt Chinoli

November 23, 2023

https://github.com/ViniciusBittencourtChinoli/Ordena-o

1 Introdução

Neste relatório, apresentamos uma análise detalhada do desempenho dos algoritmos de ordenação: *Insertion Sort*, *Quick Sort* e *Merge Sort*. O objetivo é avaliar o desempenho desses algoritmos em termos de tempo de execução, número de trocas e número de iterações, utilizando vetores de diferentes tamanhos.

2 Metodologia

Os algoritmos foram implementados e testados em vetores de inteiros gerados aleatoriamente com tamanhos de 50, 500, 1000, 5000 e 10000. Cada configuração foi repetida cinco vezes para garantir resultados mais precisos. As formas para avaliar o desempenho incluem o tempo de execução em nanossegundos, o número médio de trocas e o número médio de iterações.

3 Resultados

3.1 Insertion Sort

Tamanho do Vetor	**Média Tempo (ns)**	**Média Trocas**	**Média Iterações**
50	18600.0	634	49
500	491780.0	62563	499
1000	737420.0	249015	999
5000	3564760.0	6218221	4999
10000	14270840.0	24941084	9999

Table 1: Resultados para o algoritmo *Insertion Sort*.

3.2 Quick Sort

Tamanho do Vetor	**Média Tempo (ns)**	**Média Trocas**	**Média Iterações**
50	13340.0	124	246
500	148900.0	2690	5045
1000	89280.0	5814	11231
5000	457040.0	41207	72810
10000	561680.0	113217	177216

Table 2: Resultados para o algoritmo *Quick Sort*.

3.3 Merge Sort

Tamanho do Vetor	Média Tempo (ns)	Média Trocas	Média Iterações
50	51380.0	608	115
500	246260.0	60724	1952
1000	173080.0	247168	4390
5000	909840.0	6206893	28119
10000	1888740.0	24854878	61254

Table 3: Resultados para o algoritmo *Merge Sort*.

4 Análise

4.1 Insertion Sort

O algoritmo de *Insertion Sort* apresentou desempenho adequado para conjuntos de dados menores (50 e 500 elementos), mas sua eficiência diminuiu significativamente com tamanhos maiores. O número de trocas e iterações aumentou proporcionalmente ao tamanho do vetor, indicando um comportamento quadrático.

O Insert sort funciona da seguinte maneira, o algoritmo começa com o segundo elemento do vetor e o compara com o primeiro elemento,se o segundo elemento for menor ou igual ao primeiro elemento, ele é deixado no lugar, se o segundo elemento for maior que o primeiro elemento, ele é trocado com o primeiro elemento, o algoritmo então repete esses passos para o terceiro elemento do vetor, e assim por diante.

No exemplo fornecido, o algoritmo começa com o segundo elemento, 123. Como 123 é menor que 379, ele é deixado no lugar. O algoritmo então passa para o terceiro elemento, 1. Como 1 é menor que 123, ele é trocado com 123. O algoritmo então passa para o quarto elemento, 755. Como 755 é maior que 1, ele é trocado com 1. O processo continua até que todos os elementos do vetor estejam ordenados.

```
[379, 655, 123, 1, 755, 726, 593, 994]
[123, 379, 655, 1, 755, 726, 593, 994]
[1, 123, 379, 655, 755, 726, 593, 994]
[1, 123, 379, 655, 755, 726, 593, 994]
[1, 123, 379, 655, 726, 755, 593, 994]
[1, 123, 379, 593, 655, 726, 755, 994]
[1, 123, 379, 593, 655, 726, 755, 994]
```

4.2 Quick Sort

O *Quick Sort* demonstrou uma excelente eficiência em comparação com o *Insertion Sort*. O tempo de execução aumentou de forma relativamente linear com o tamanho do vetor. O número de trocas e iterações também aumentou, mas em uma taxa menor que o *Insertion Sort*, indicando uma complexidade média menor.

Quick Sort é um algoritmo de ordenação divide e impera ele escolhe um elemento do vetor, chamado de pivô, divide o vetor em duas metades, uma com todos os elementos menores ou iguais ao pivô e outra com todos os elementos maiores ou iguais ao pivô, recursivamente ordena cada metade do veto. algoritmo divide o vetor em duas metades[363, 342, 391] e [546, 632, 674, 884, 954] entao ocorrem as iteraçoes e trocas de pivot que passa a ser 546 para 363 e entao por 342, e por fim a ordenação esta completa.

```
[546, 363, 342, 391, 632, 674, 884, 954]
[363, 342, 391, 546, 632, 674, 884, 954]
[342, 363, 391, 546, 632, 674, 884, 954]
[342, 363, 391, 546, 632, 674, 884, 954]
[342, 363, 391, 546, 632, 674, 884, 954]
```

4.3 Merge Sort

O *Merge Sort* exibiu um desempenho consistente e eficiente em todos os tamanhos de vetor testados. O tempo de execução aumentou de forma um pouco mais pronunciada em comparação com o *Quick Sort*, mas o número de trocas e iterações foi significativamente menor, sugerindo uma complexidade menor.

O Merge sort divide o vetor em duas metades, até que cada metade contenha apenas um elemento, em seguida, o algoritmo merge as metades ordenadas, uma a uma, para obter um vetor ordenado. as duas metades são as seguintes [928, 973] e [567, 783]. A primeira metade é ordenada da seguinte forma: o algoritmo começa comparando os dois elementos, 928 e 973, como 928 é menor que 973, ele é colocado na frente do vetor. A segunda começa comparando os dois elementos, 567 e 783. Como 567 é menor que 783, ele é colocado na frente do vetor. A segunda parte funciona da mesma forma [293, 721] e [13, 448] são as metades depois de estarem organizados o algoritmo merge as duas metades ordenadas de 4 elementos cada. O vetor resultante é o seguinte: [13, 293, 448, 567, 721, 783, 928, 973]

```
[928, 973]
[567, 783]
[567, 783, 928, 973]
[293, 721]
[13, 448]
[13, 293, 448, 721]
[13, 293, 448, 567, 721, 783, 928, 973]
```

5 Gráficos e Tabelas

Tamanho do Vetor	Merge Sort	Quick Sort	Insertion Sort
50	51380	13340	18600
500	246260	148900	491780
1000	173080	89280	737420
5000	909840	457040	3564760
10000	1888740	561680	14270840

Table 4: Tempo de execução médio (ns) para cada algoritmo.

Tamanho do Vetor	Merge Sort	Quick Sort	Insertion Sort
50	608	124	634
500	60724	2690	62563
1000	247168	5814	249015
5000	6206893	41207	6218221
10000	24854878	113217	24941084

Table 5: Número médio de trocas para cada algoritmo.

Tamanho do Vetor	Merge Sort	Quick Sort	Insertion Sort
50	115	246	49
500	1952	5045	499
1000	4390	11231	999
5000	28119	72810	4999
10000	61254	177216	9999

Table 6: Número médio de iterações para cada algoritmo.

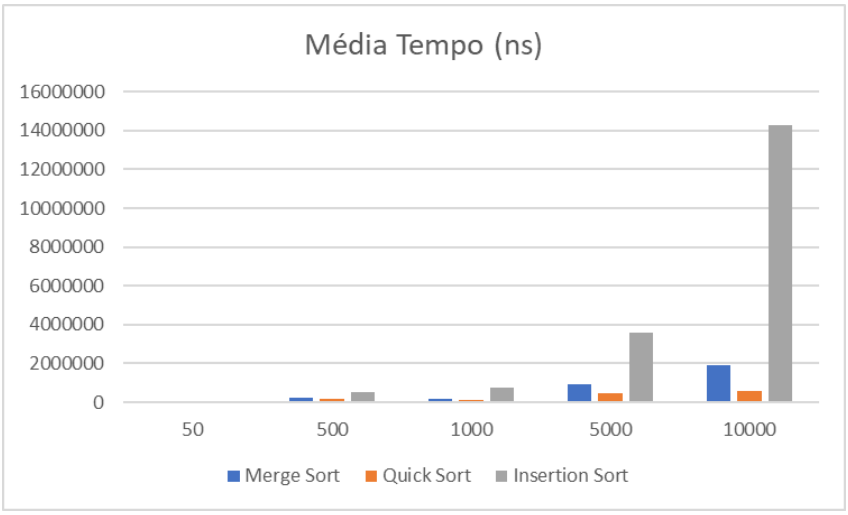

Figure 1: Média tempo

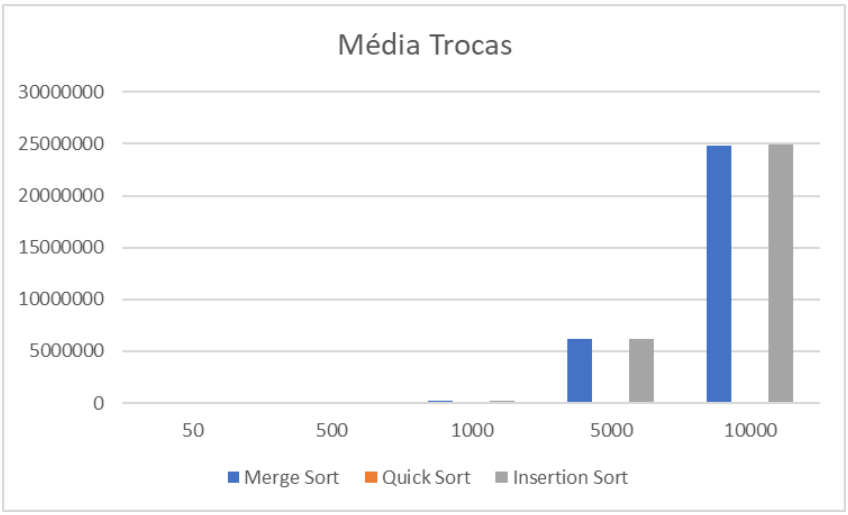

Figure 2: Média Trocas

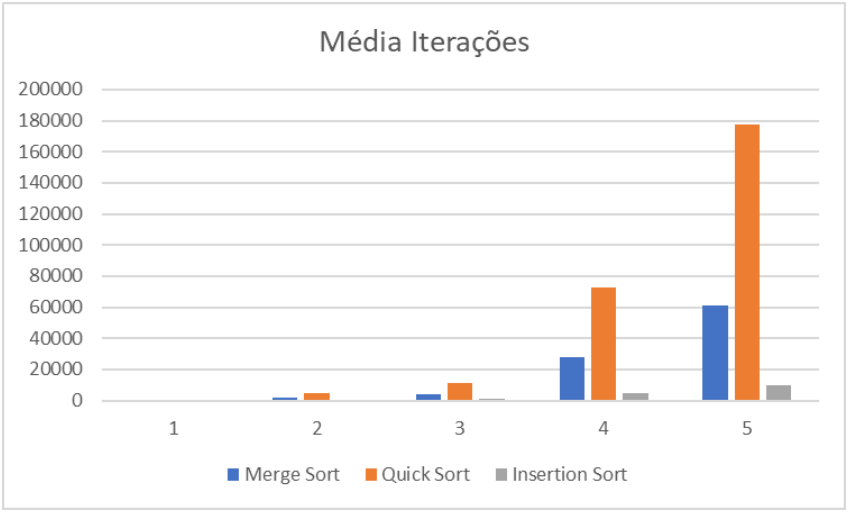

Figure 3: Meia iterações