UNIVERSIDADE ANHEMBI MORUMBI

DAVY DA SILVA FARIA, DOUGLAS ALMEIDA SILVA, MATHEUS SILVA ESPINOSA, RAFAEL SANTOS DE ALMEIDA

RELATÓRIO DE PESQUISA E ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

São Paulo - SP

2025

SUMÁRIO

1. Funcionamento dos Algoritmos

1.1. Counting Sort

O Counting Sort é um algoritmo que dispensa comparações e se baseia na contagem das ocorrências de cada valor para determinar suas posições finais. Ele preserva a ordem entre elementos iguais, sendo estável, mas não é in-place devido ao uso de vetores auxiliares. É eficiente quando o intervalo de valores é pequeno e bem definido.

Seu uso é ideal para ordenar chaves inteiras em faixas reduzidas, como idades, categorias discretas ou como parte do Radix Sort. Contudo, quando o intervalo é muito amplo, o custo de memória cresce demais, prejudicando sua viabilidade. Ele também se aplica apenas diretamente a inteiros ou valores facilmente mapeáveis.

Segue o algoritmo do Counting Sort:

1.1.1. Determinação do Intervalo de Valores

A primeira etapa do algoritmo consiste em identificar o menor e o maior valor presentes na lista de entrada. Isso permite determinar o intervalo total de valores possíveis, informação essencial para definir o tamanho do vetor auxiliar de contagem. Sem essa identificação inicial, não seria possível mapear corretamente cada valor às posições que irão representar suas frequências.

1.1.2. Construção do Vetor de Contagem

Após descobrir o intervalo, é criado um vetor auxiliar em que cada posição representa um valor possível da lista. Todas as posições começam inicialmente com zero, sendo incrementadas conforme cada elemento é encontrado durante a varredura da lista original. Dessa forma, esse vetor passa a armazenar quantas vezes cada valor aparece, substituindo a necessidade de comparar elementos entre si.

1.1.3. Cálculo das Contagens Acumuladas

Em seguida, as contagens simples são transformadas em contagens acumuladas. Cada posição passa a indicar o limite superior da região onde determinado valor deve ser inserido no vetor final. Esse acúmulo é essencial para determinar a posição exata dos elementos no vetor ordenado, garantindo a estabilidade do algoritmo quando implementado de forma apropriada.

1.1.4. Construção do Vetor Ordenado

Por fim, o algoritmo percorre a lista original de trás para frente, inserindo cada elemento em sua posição correta no vetor ordenado conforme indicado pelo vetor de contagens acumuladas. A cada inserção, o valor acumulado é decrementado, mantendo o controle das próximas posições disponíveis. Esse processo resulta em um vetor completamente ordenado sem necessidade de comparação entre elementos.

1.2. Insertion Sort

O Insertion Sort insere cada novo elemento na posição adequada da sublista já ordenada, funcionando de maneira intuitiva e direta. Ele é estável e in-place, utilizando pouquíssima memória extra e preservando a ordem relativa dos elementos iguais. Seu desempenho é excelente em listas pequenas ou quase ordenadas.

É usado em algoritmos híbridos e em situações em que simplicidade e baixa memória são desejáveis. Entretanto, apresenta complexidade $O(n^2)$ no pior caso, sendo pouco eficiente para listas grandes e completamente desordenadas. Assim, costuma servir melhor como complemento de outros métodos.

Segue o algoritmo do Insertion Sort:

1.2.1. Seleção do Elemento e Definição da Chave

O algoritmo inicia analisando o vetor a partir do segundo elemento, pois o primeiro já é considerado ordenado. Em cada iteração, o valor atual é armazenado em uma variável chamada chave, representando o elemento que deverá ser inserido na posição adequada dentro da parte já ordenada do vetor.

1.2.2. Comparação com Elementos Anteriores

Após definir a chave, o algoritmo começa a compará-la com os elementos anteriores. Enquanto esses elementos forem maiores (na ordenação crescente) ou menores (na ordenação decrescente), eles são deslocados uma posição para a direita, abrindo espaço para que a chave seja inserida corretamente. Esse movimento garante que a sublista à esquerda permaneça sempre ordenada durante o processo.

1.2.3. Inserção da Chave na Posição Correta

Quando não houver mais elementos que devam ser deslocados, o espaço correto para a inserção é alcançado. Nesse ponto, a chave é finalmente posicionada, consolidando a ordenação da sublista analisada. Esse procedimento se repete para todos os elementos do vetor, ampliando gradualmente o trecho ordenado até que toda a lista esteja organizada.

1.3. Merge Sort

O Merge Sort utiliza divisão e conquista, separando a lista em partes menores e depois intercalando-as de forma ordenada. Ele é estável, mas não in-place, pois necessita de memória auxiliar proporcional ao tamanho dos dados. Seu desempenho O(n log n) é garantido em qualquer cenário.

É adequado para aplicações que demandam estabilidade, paralelização ou ordenações externas, como bancos de dados e grandes volumes de informação. Sua principal limitação é o uso elevado de memória, o que pode ser inviável em sistemas com recursos restritos ou listas muito extensas.

Segue o algoritmo do Merge Sort:

1.3.1. Etapa da Divisão

O array é dividido repetidamente em duas partes iguais. Esse processo continua até que cada lista subordinada possua apenas um único elemento, pois um array com um elemento é considerado ordenado por definição.

1.3.2. Etapa de Conquista

Após as subdivisões, o algoritmo começa a resolver cada pequena parte individualmente. Como cada lista subordinada possui apenas um elemento,

elas já estão ordenadas, e então o algoritmo começa a “subir” na árvore de recursão, seguindo para a etapa de combinação as listas ordenadas.

1.3.3. Etapa de Combinação (Merge)

O algoritmo recebe duas listas já ordenadas e as combina em uma única lista também ordenada. Para isso, são usados dois ponteiros, um para a primeira metade e outro para a segunda. A cada comparação, o menor elemento é copiado para o array final. Esse processo se repete até que todas as listas subordinadas tenham sido combinadas novamente, produzindo um array final completamente ordenado.

1.4. Quick Sort

O Quick Sort divide o problema a partir de um pivô, colocando elementos menores à esquerda e maiores à direita, aplicando recursão às sublistas. Ele tem excelente desempenho médio e é in-place, mas não é estável, pois o particionamento pode alterar a ordem de elementos iguais. Aproveita bem o cache e costuma ser o mais rápido na prática.

Apesar do bom desempenho médio O(n log n), seu pior caso é $O(n^2)$, principalmente quando o pivô é mal escolhido. Por isso, técnicas como pivô aleatório ou mediana de três são usadas para reduzir riscos. Ainda assim, continua sendo um dos métodos mais eficientes para ordenação interna.

Segue o algoritmo Quick Sort:

1.4.1. Verificação do Tamanho do Vetor

Se o vetor tem mais de um único elemento, segue para as próximas etapas, senão encerra o Quick Sort.

1.4.2. Escolha do pivô

Nesta etapa é feita a escolha do pivô que pode ser feita de diversas maneiras como selecionando o primeiro ou o último elemento do vetor, escolhendo um elemento aleatoriamente ou então através da mediana de três elementos, o objetivo consiste em selecionar um pivô adequado, isto é, aquele capaz de

particionar a lista em duas partes cujos tamanhos sejam o mais equilibrados possível.

O método utilizado no código implementado foi o da mediana de três elementos, que é calculado a mediana entre o valor do primeiro elemento, o do meio e o último, aquele elemento que for a mediana é utilizado como pivô, este método reduz o risco do pior cenário, que seria a lista estar previamente ordenada.

1.4.3. Particionamento

O particionamento é a reordenação dos elementos do vetor em torno do pivô, para que fiquem os menores que o pivô ao lado esquerdo da divisão e os maiores que o pivô ao lado direito da divisão, novamente nesta etapa existe diversos métodos que executam essa mesma operação e cada um deles com suas particularidades, como o método de Lomuto, método de Hoare, três vias etc.

Vale destacar o método utilizado no código implementado nesta pesquisa que foi o de Hoare, inicialmente o pivô é movimentado para a primeira posição do vetor, após isso são definidos dois índices, um que irá percorrer o vetor da esquerda para direita e outro que irá da direita para a esquerda, eles irão fazer isso dentro de um loop while com a condição igual a true, além disso é utilizado para incrementar e decrementar os índices criados dois loops “do while”, o índice que começa da esquerda é incrementado até que o elemento indicado por ele seja maior ou igual ao pivô e o índice que começa da direita é decrementado até que o elemento indicado por ele seja menor ou igual ao pivô, após isso se o índice da esquerda for maior ou igual que o índice da direita o método irá retornar o índice da direita, senão irá realizar a troca do elemento do índice da esquerda com o índice da direita.

Esse método irá garantir que a lista ficará dividida em duas partes, a parte da esquerda do pivô com os valores menores que ele e os da direita com os valores maiores que o dele.

1.4.4. Recursão

Nesta etapa o quick sort é aplicado nos subintervalos da lista que foram gerados pela divisão feita no método de partição, o primeiro intervalo é do índice inicial até o índice do pivô e o segundo intervalo é do índice seguinte do pivô até o índice final da lista.

Além disso a recursão irá parar quando o subintervalo tiver apenas um único elemento, assim no final de todo o processo a lista inicial estará completamente ordenada.

2. Resultados dos Testes

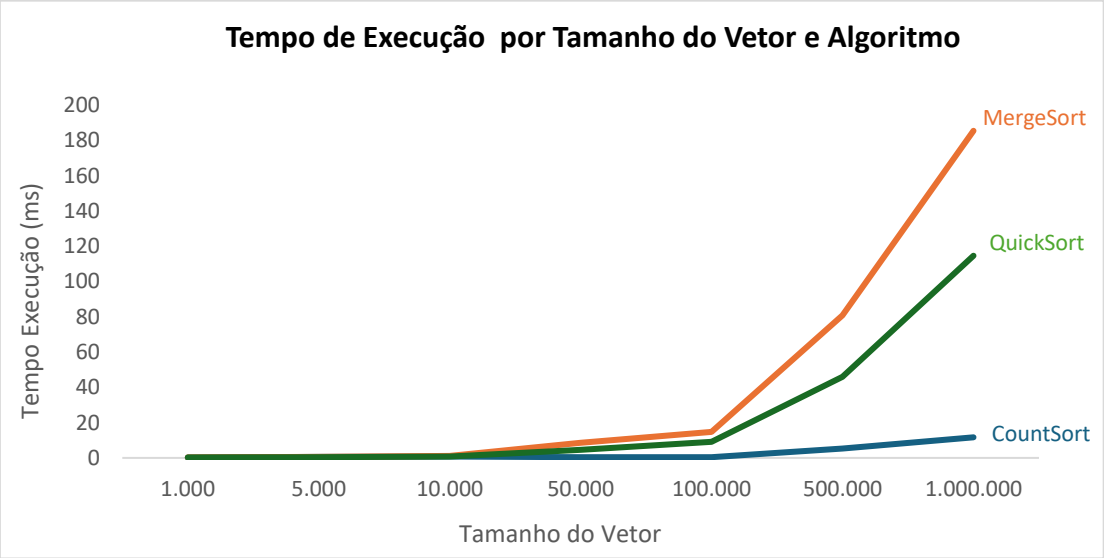
2.1. Resultados com Vetor Aleatório não Ordenado

2.1.1. Tempo Médio de Execução

Segue a tabela e o gráfico com os resultados do tempo médio de execução dos algoritmos de ordenação para cada tamanho de vetor:

Tempo médio (ms) / Tamanho do Vetor	Algoritmos / CountSort	InsertionSort	MergeSort	QuickSort
1.000	0,1527	1,6194	0,2965	0,2718
5.000	0,5544	13,5062	0,5617	0,4378
10.000	0,8316	16,6728	1,0967	0,8153
50.000	0,4446	504,9108	8,6122	4,5714
100.000	0,4668	2195,9009	14,8209	9,1104
500.000	5,3117	59899,1137	80,8966	45,8911
1.000.000	11,6839		185,4686	114,6402

Obs: O teste do InsertionSort com o vetor de 1 milhão de elementos não foi feito, pois estava travando a execução do código.



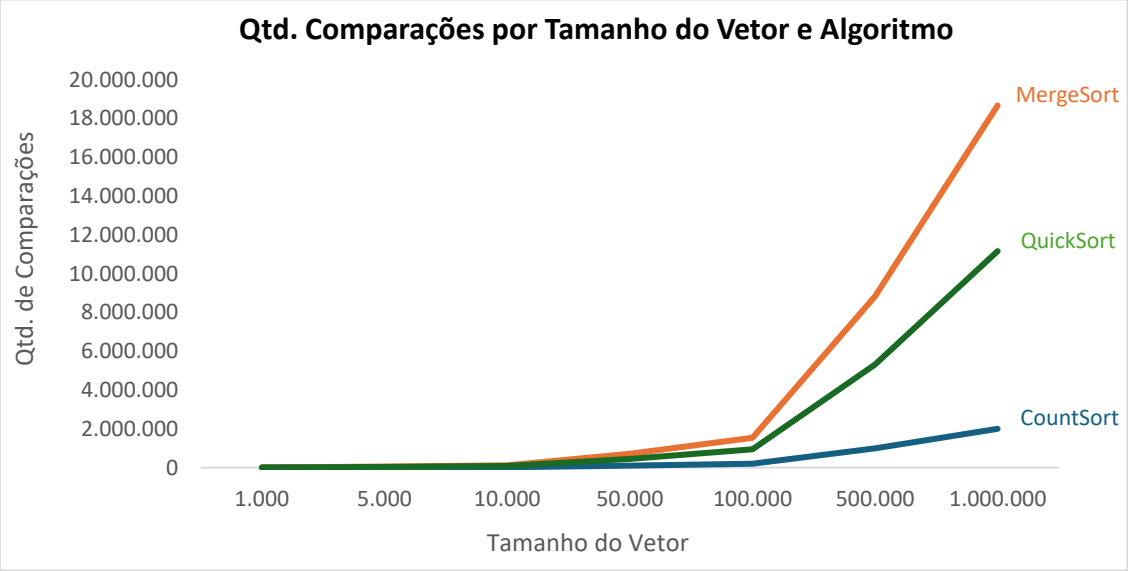
Obs: InsertionSort foi retirado do gráfico, pois seu resultado foi muito discrepante e isso atrapalhou a visualização dos demais.

2.1.2. Quantidade Média de Comparações

Segue a tabela e o gráfico com os resultados da quantidade de comparações dos algoritmos de ordenação:

Qtd. De Comparações	Rótulos de Coluna			
Rótulos de Linha	CountSort	InsertionSort	MergeSort	QuickSort
1.000	1.998	256.159	8.712	6.822
5.000	9.998	6.256.660	55.183	37.713
10.000	19.998	24.897.335	120.458	79.656
50.000	99.998	625.178.959	718.172	451.145
100.000	199.998	2.493.769.933	1.536.133	950.272
500.000	999.998	62.457.696.397	8.835.557	5.325.687
1.000.000	1.999.998		18.670.251	11.161.850

Obs: O teste do InsertionSort com o vetor de 1 milhão de elementos não foi feito, pois estava travando a execução do código.



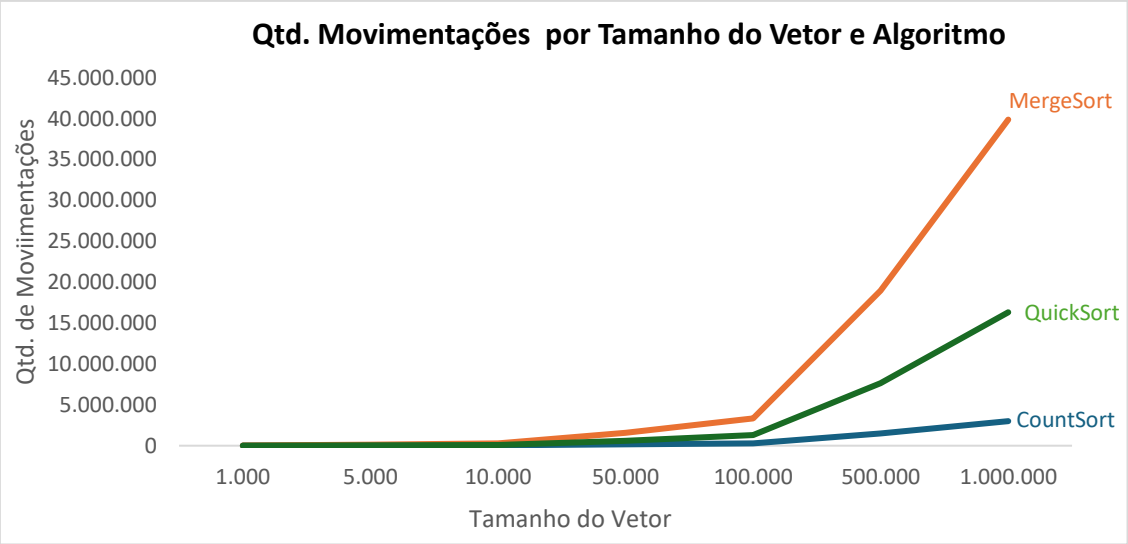
Obs: InsertionSort foi retirado do gráfico, pois seu resultado foi muito discrepante e isso atrapalhou a visualização dos demais.

2.1.3. Quantidade Média de Movimentações

Segue a tabela e o gráfico com os resultados da quantidade de comparações dos algoritmos de ordenação:

Qtd. de Movimentações	Rótulos de Coluna			
Rótulos de Linha	CountSort	InsertionSort	MergeSort	QuickSort
1.000	4.000	256.159	19.952	7.648
5.000	16.000	6.256.660	123.616	45.431
10.000	31.000	24.897.335	267.232	99.317
50.000	151.000	625.178.959	1.568.928	602.294
100.000	301.000	2.493.769.933	3.337.856	1.300.548
500.000	1.501.000	62.457.696.397	18.951.424	7.651.378
1.000.000	3.001.000		39.902.848	16.323.704

Obs: O teste do InsertionSort com o vetor de 1 milhão de elementos não foi feito, pois estava travando a execução do código.



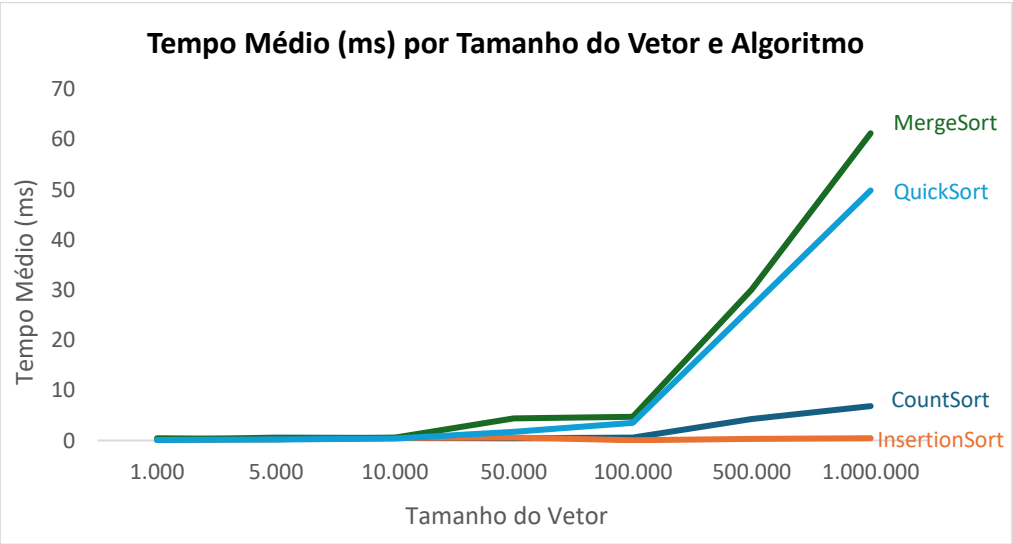
Obs: InsertionSort foi retirado do gráfico, pois seu resultado foi muito discrepante e isso atrapalhou a visualização dos demais.

2.2. Resultados com Vetor Aleatório e Ordenado Crescente

2.2.1. Tempo Médio de Execução

Segue a tabela e o gráfico com os resultados do tempo médio de execução dos algoritmos de ordenação para cada tamanho de vetor:

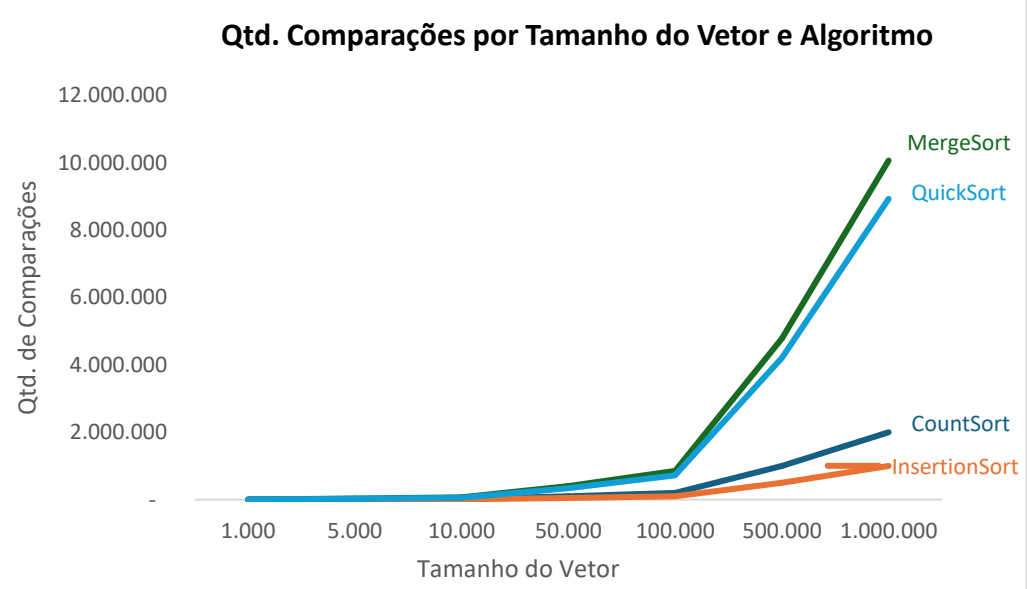
Tempo Médio (ms) / Tamanho do Vet	Algoritn / CountSort	InsertionSort	MergeSort	QuickSort
1.000	0,1616	0,0574	0,4264	0,0973
5.000	0,5467	0,2454	0,2541	0,2029
10.000	0,5074	0,5053	0,5536	0,3632
50.000	0,4699	0,5915	4,3993	1,7105
100.000	0,5467	0,0453	4,7490	3,4878
500.000	4,2849	0,3040	30,0293	26,6274
1.000.000	6,8485	0,4252	61,2448	49,8489



2.2.2. Quantidade Média de Comparações

Segue a tabela e o gráfico com os resultados da quantidade média de comparações dos algoritmos para cada tamanho de vetor:

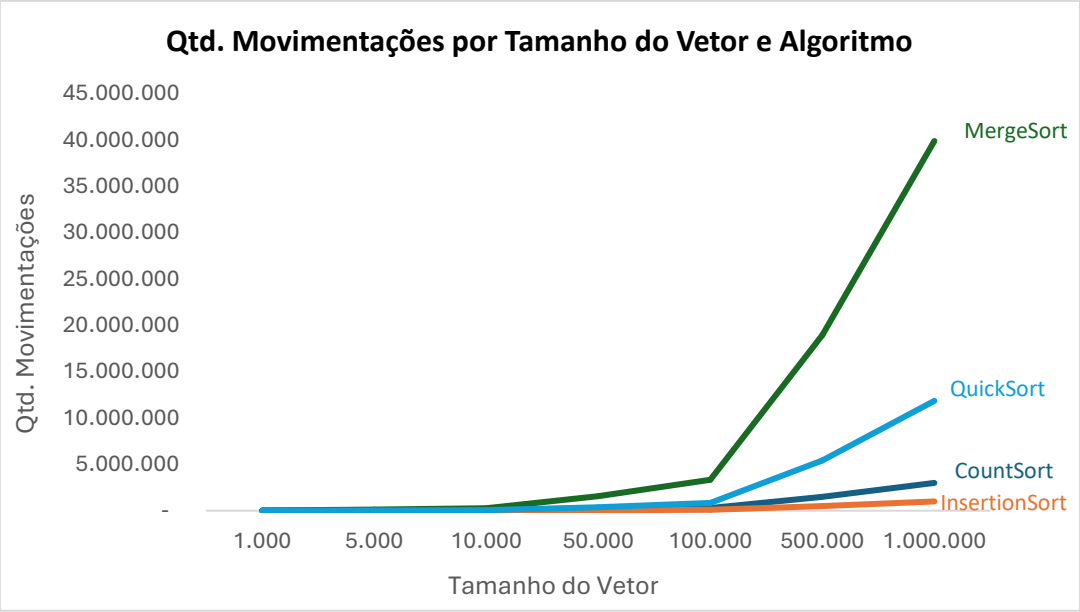
Qtd. Comparações	Algoritmo			
Tamanho do Vetor	CountSort	InsertionSort	MergeSort	QuickSort
1.000	1.998	999	5.044	5.099
5.000	9.998	4.999	32.004	27.408
10.000	19.998	9.999	69.008	57.865
50.000	99.998	49.999	401.952	337.169
100.000	199.998	99.999	853.904	720.219
500.000	999.998	499.999	4.783.216	4.210.650
1.000.000	1.999.998	999.999	10.066.432	8.928.606



2.2.3. Quantidade Média de Movimentações

Segue a tabela e o gráfico com os resultados da quantidade média de movimentações dos algoritmos para cada tamanho de vetor:

Qtd. Movimentações	Algoritmo			
Tamanho do Vetor	CountSort	InsertionSort	MergeSort	QuickSort
1.000	3.999	999	19.952	4.202
5.000	16.000	4.999	123.616	24.821
10.000	31.000	9.999	267.232	55.735
50.000	151.000	49.999	1.568.928	374.342
100.000	301.000	99.999	3.337.856	840.442
500.000	1.501.000	499.999	18.951.424	5.421.304
1.000.000	3.001.000	999.999	39.902.848	11.857.216

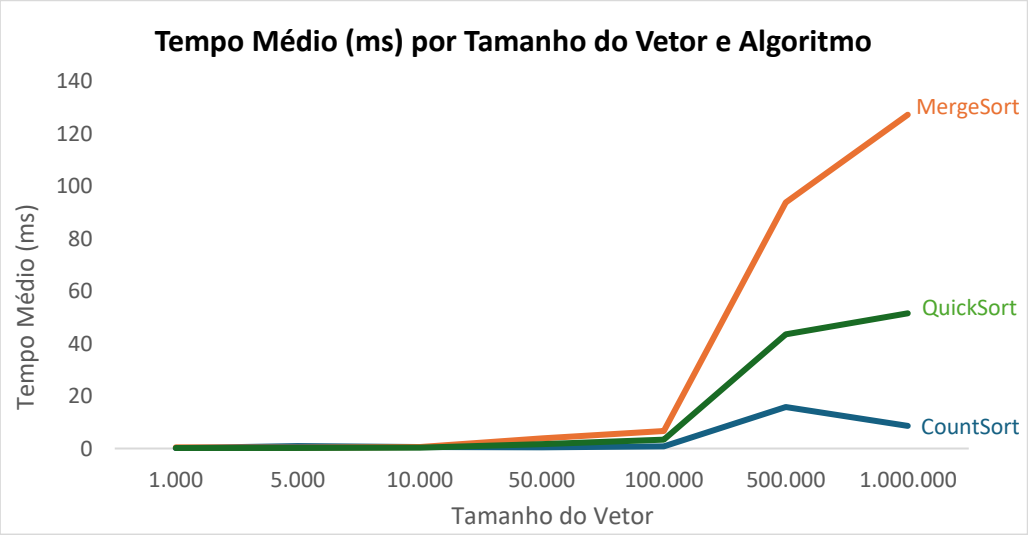


2.3. Resultados com Vetor Aleatório e Ordenado Decrescente

2.3.1. Tempo Médio de Execução

Segue a tabela e o gráfico com os resultados do tempo médio de execução dos algoritmos de ordenação para cada tamanho de vetor:

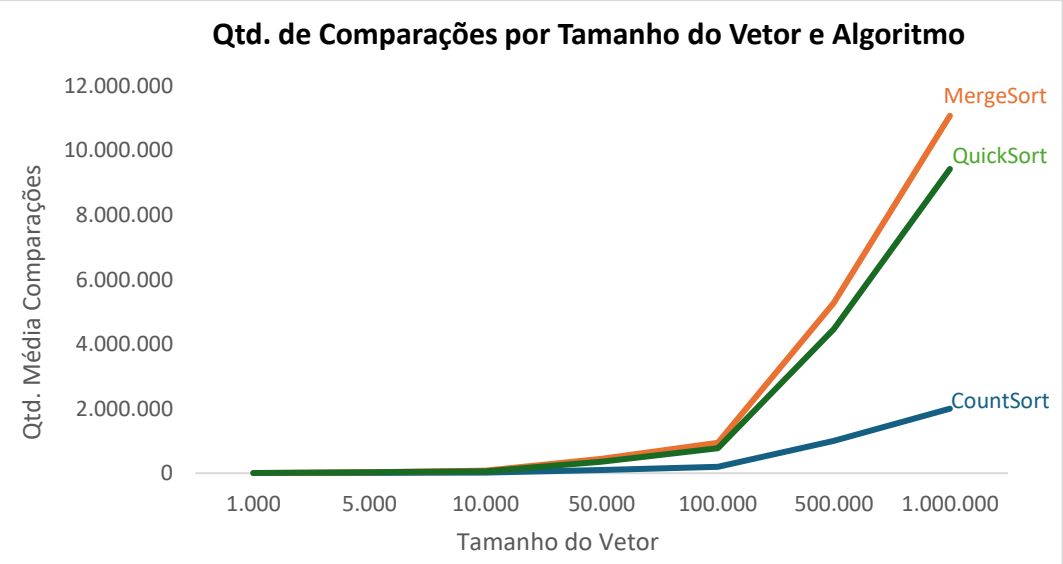
Tempo Médio (ms)	Algoritmos			
Tamanho do Vetor	CountSort	InsertionSort	MergeSort	QuickSort
1.000	0,1527	3,8527	0,4338	0,2296
5.000	0,8780	17,5047	0,3370	0,2085
10.000	0,5145	51,8123	0,5788	0,3703
50.000	0,3948	1539,6720	3,8638	1,6568
100.000	0,8251	5840,6096	6,6310	3,3552
500.000	15,8170	191382,2193	93,8492	43,5771
1.000.000	8,6046		127,2829	51,5568



2.3.2. Quantidade Média de Comparações

Segue a tabela e o gráfico com os resultados da quantidade média de comparações dos algoritmos para cada tamanho de vetor:

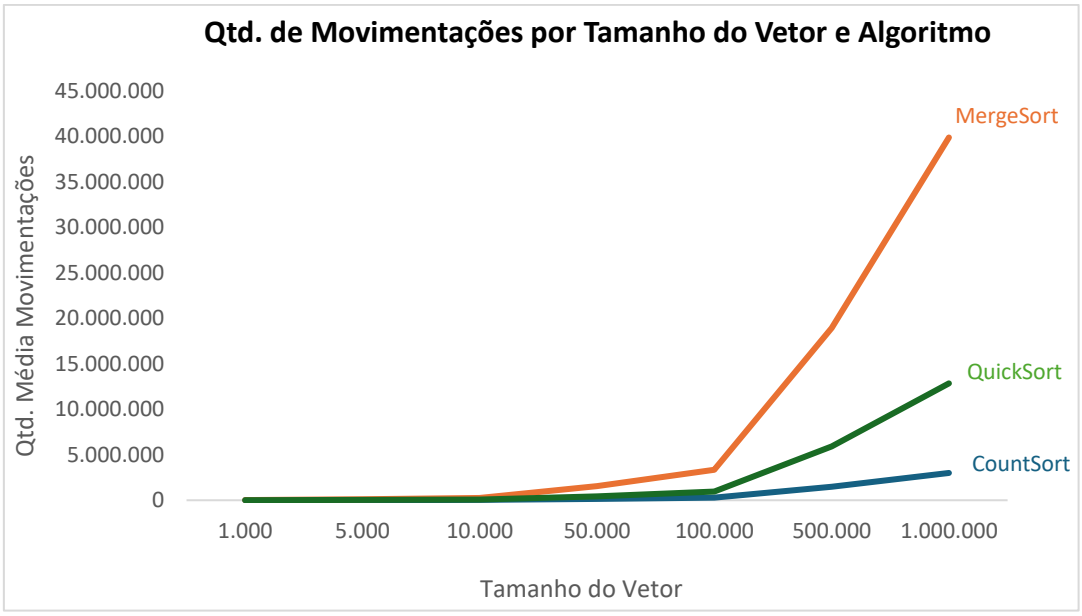
Qtd. Média Comparações	Algoritmos			
Tamanho do Vetor	CountSort	InsertionSort	MergeSort	QuickSort
1.000	1998	500005	5142	5584
5.000	9998	12490062	33884	29896
10.000	19998	49954867	75091	62865
50.000	99998	1248776471	447345	362149
100.000	199998	4995056280	945717	770141
500.000	999998	1,24875E+11	5275474	4460766
1.000.000	1999998		11076445	9427463



2.3.3. Quantidade Média de Movimentações

Segue a tabela e o gráfico com os resultados da quantidade média de movimentações dos algoritmos para cada tamanho de vetor:

Qtd. Média Movimentações	Algoritmos			
Tamanho do Vetor	CountSort	InsertionSort	MergeSort	QuickSort
1.000	3998	500005	19952	5172
5.000	16000	12490062	123616	29797
10.000	31000	49954867	267232	65735
50.000	151000	1248776471	1568928	424303
100.000	301000	4995056280	3337856	940286
500.000	1501000	1,24875E+11	18951424	5921536
1.000.000	3001000		39902848	12854931


Qtd. de Movimentações por Tamanho do Vetor e Algoritmo
Qtd. Média Movimentações
45.000.000
40.000.000
35.000.000
30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
MergeSort
QuickSort
CountSort
1.000
5.000
10.000
50.000
100.000
500.000
1.000.000
Tamanho do Vetor

3. Conclusão

Os algoritmos avaliados apresentam diferenças importantes quanto às suas complexidades e necessidades de memória. O Counting Sort é extremamente rápido com complexidade O(n + k), mas depende de um intervalo de valores pequeno para ser viável. O Insertion Sort mantém $O(n^2)$ no pior caso, funcionando bem apenas para listas pequenas ou quase ordenadas. O Merge Sort tem O(n log n) garantido, mas exige memória auxiliar proporcional ao tamanho da entrada. Já o Quick Sort costuma ser o mais rápido na prática, com O(n log n) médio, embora possa chegar a $O(n^2)$ se o pivô for mal escolhido.

Os testes experimentais confirmam o comportamento esperado pela teoria. O Counting Sort apresentou desempenho excelente quando aplicável, enquanto o Merge Sort manteve tempos consistentes independentemente da ordem inicial dos dados. O Quick Sort se destacou pela rapidez em vetores grandes e variados, sobretudo com a técnica da mediana de três, que reduziu o risco de pior caso. O Insertion Sort, por outro lado, demonstrou limitação clara em listas extensas, sendo inviável para escalas maiores. Assim, os resultados reforçam que a eficiência prática depende diretamente do tipo de entrada.

Quanto à escolha do algoritmo, ela deve considerar tanto as características dos dados quanto os requisitos do sistema. O Counting Sort é ideal quando o intervalo de valores é pequeno e conhecido; já o Insertion Sort é mais vantajoso em listas pequenas ou parcialmente ordenadas e em algoritmos híbridos. O Merge Sort se destaca quando estabilidade e paralelização são essenciais, além de ser adequado para ordenação externa. O Quick Sort, com boa escolha de pivô, é a melhor opção geral para grandes volumes de dados em memória. Dessa forma, o algoritmo ideal varia conforme o contexto e as restrições do problema.