Relatório do Projeto 2
"Mais ordenação!"

Guilherme Pereira de Sá

November 15, 2024

Professor: Marcelo Manzato

Contents

1 Introdução

A implementação dos métodos de ordenação pedidos, que são: **BubbleSort**, **SelectionSort**, **InsertionSort**, **ShellSort**, **QuickSort**, **HeapSort**, **MergeSort**, **Contagem dos Menores** e **RadixSort**, vão estar mais "ocultas" neste relatório, já que o intuito aqui não é apresentá-los, mas sim, compará-los com os seguintes parâmetros: 2

- O número de comparação de chaves
- O número de movimentações de registros
- O tempo de execução do algoritmo

2 Desenvolvimento dos itens

Para desenvolvermos uma comparação ótima, precisamos primeiro conhecer nossos itens:

2.1 BubbleSort

Método de comparação geral, primeiro, de um a um, com o vetor sendo percorrido com um elemento fixo enquanto ordenamos o resto dos n-1 elementos em torno dele e assim com todos os elementos.

2.2 QuickSort

Método que se utiliza do Dividir para Conquistar, onde pegamos um pivô, dois indicadores de posição e o menor elemento. Dividir para conquistar, porque além de recursivo, o método faz a separação do vetor em partes cada vez menores, sempre contendo as características passadas, por isso é log(n), comparando sempre com o pivô.

2.3 SelectionSort

Um bubbleSort mais fácil de se entender. Encontramos um elemento que seja o maior (ou o menor) da sua iteração e colocamos no final (ou no "começo"), antes (ou depois), dos já ordenados, então temos dois contadores de posição. Faz um número fixo de comparações.

2.4 InsertionSort

Método de inserção direta, ele percorre o vetor e coloca seus elementos na posição correta movendo os elementos adjacentes posteriores à posição que o "novo elemento foi colocado" para a direita, então a parte de "inserção" vem justamente de supor que o vetor já está ordenado e que os novos elementos,

partindo de um só, serão comparados com os que já estão enfilerados alí e adicionados na posição correta. Se tudo já estiver ordenado ele apenas verifica, passando pelo vetor uma vez.

2.5 ShellSort

Método de ordenação derivado do InsertionSort, onde ao invés de pegarmos o elemento adjacente, ordenamos, recursivamente, à uma distância h, que será diminuída até chegarmos no caso do InsertionSort.

2.6 HeapSort

Construímos uma árvore com um vetor sequencial, onde restringimos para elementos, como o elemento na posição k, que seus "filhos", assim chamados, estarão na posição, 2k+1 e 2k+2. Respeitamos ordem e hierarquia da árvore, em sua construção, e na sua ordenação final, onde apenas vamos colocando o elemento maior na raiz e depositando na última posição do vetor. Fazemos meio que uma "busca de profundidade", pegando uma certa sub-árvore e analisando seus elementos em apenas uma direção, separamos então de "filhos" da esquerda e os da direita de k.

2.7 RadixSort

O RadixSort se baseia no dígito do elemento atual, portanto, ele não se utiliza de comparações como passo principal da ordenação. Ele começa pelo dígito menos significativo para o mais significativo. Então analisamos digitos abaixo de 10, como o 9, 99, 999, 9999, etc. Além do mais, por conta dessa questão do dígito apenas é feita a ordenação de elementos inteiros e maiores que zero.

2.8 MergeSort

Criamos vetores auxiliares para dividir (dividir e conquistar) um vetor desordenado várias vezes até termos um vetor de apenas um elemento, que claramente já está ordenado, é um vetor unitário (não tem com quem compará-lo), e aí que entra a parte do "Merge", onde mesclamos os vetores, (que para limitarmos tem uma sentinela na última posição), apenas comparando seus elementos entre si para sabermos onde a junção dos vetores unitários, se ligando cada vez mais, como no processo inverso da divisão, para ficar no final apenas o vetor original com as junções do tamanho que fora repartido em: (final - meio); e (meio - inicio + 1).

2.9 Contagem de Menores

Ao contarmos um vetor, quantos elementos de um elemento são menores que ele, podemos simplesmente salvar a posição diretamente deste elemento, já que

sabemos em que posição ele estará de acordo com a comparação entre o resto dos elementos.

O objetivo deste trabalho vai além da mera comparação de qual seria o melhor *sort* para todos os casos possíveis. Irei analisar empiricamente os dados gerados para, em situações distintas, (ordenado, inversamente ordenado e aleatório), quais seriam os sort's que melhor se encaixariam.

Os dados foram gerados por partindo da ideia de complementar um arquivo.txt, externo, para depois lermos a partir deste arquivo, os dados, que geralmente, para os parâmetros das funções de sort são: o vetor em sí, que é do tipo *int*, e o tamanho do vetor. (Alguns casos requerem de onde começa e onde termina o vetor também). Assim, podemos criar funções para medirTempo das funções de sort, onde a ideia principal é de passar a função como parâmetro, com seus parâmetros específicos, e com o *time.h*, podemos calcular o tempo de execução do algoritmo sort passado de período para o tempo em segundos como *double*.

3 Resultados

Para verificarmos nossos resultados, usaremos uma main.c para juntarmos todos os nossos itens e nele, primeiro, com a análise empírica: analisarmos o tempo de execução de cada sort.

Foram feitos 5 *sets* para os vetores do tipo "random", já que queremos o resultado médio deste tipo. Já os outros, "ordenados" e "reversamente ordenados", seguiram com os mesmos elementos, por isso serão apresentados só uma vez.

3.1 BubbleSort e SelectionSort

Em vetores ordenados, ambos não fazem movimentações, mas em vetores inversamente ordenados, eles realizam o número máximo de comparações e movimentos possíveis. O tempo de execução alto para vetores maiores indica a ineficiência desses algoritmos para tamanhos grandes, especialmente em casos desordenados, aleatórios, ou os inversamente ordenados. O BubbleSort, por ser o "aprimorado" pode interromper antecipadamente, reduzindo o número de operações, por conta daquela variável de "troca realizada". Enquanto o SelectionSort sempre percorre todo o vetor.

3.2 InsertionSort

Apresenta ótimo desempenho em vetores já ordenados, pois realiza apenas n-1 comparações e poucos movimentos. Ele precisa apenas confirmar que cada elemento está na sua posição exata.. Em vetores inversamente ordenados, ele precisa reorganizar cada elemento, o que leva a um número muito maior de comparações e movimentos.

3.3 ShellSort

Algoritmo conhecido por ter bom desempenho em vetores parcialmente ordenados. Se o gap for bem escolhido isso pode reduzir drasticamente o número de comparações e movimentações.

3.4 Contagem de Menores Sort

A quantidade de comparações e movimentos altos faz sentido, especialmente em vetores com uma contagem fixa de elementos. A eficiência deste método depende da distribuição dos dados, e ele pode apresentar muitos movimentos se os elementos tiverem contagens altas. Counting Sort costuma funcionar melhor sendo implementado em vetores com pequenos elementos, em valor, e que estão distribuidos uniformemente.

3.5 HeapSort

Em geral, Heap Sort realiza menos comparações do que BubbleSort e Selection Sort em todos os casos, e os números de comparações e movimentos que você vê são compatíveis com o esperado para esse algoritmo, especialmente em vetores grandes e desordenados. Possui uma reorganização eficiente e econômica, portanto. Possui comportamento bem equilibrado, mesmo em vetores grande, assim como o MergeSort.

3.6 RadixSort

A natureza baseada em dígitos do Radix Sort resulta em um número constante de comparações para cada "dígito" que ele processa, independentemente da ordenação original do vetor. No entanto, o número de movimentações pode crescer dependendo da quantidade de reorganizações internas necessárias para cada base. Isso torna o Radix Sort eficaz em dados que possuem um pequeno conjunto de valores, onde o ganho em tempo e movimento é mais evidente.

3.7 QuickSort

Demonstra bom desempenho em vetores aleatórios e ordenados, com menor tempo e comparações. Entretanto, o número de movimentos varia mais em vetores inversamente ordenados, o que é esperado devido à maior complexidade em reordenar elementos. A estratégia usada foi a de colocar o pivô como um elemento da mediana de três outros elementos do vetor já quebrado, e enviado para o quicksort, o que suaviza esses picos de movimentações.

3.8 MergeSort

A quantidade de comparações e movimentos também está de acordo com o esperado para MergeSort, que tem complexidade O(n log n). A quantidade de movimentos tende a ser alta, pois ele faz muitas cópias durante o processo de

"merge". Portanto, sua complexidade de espaço é bem alta, comparada com a dos outros sort's.

3.9 Desenvolvimento Final

No documento "figures" será mostrada as prints dos testes feitos. Fez-se 5 sets de elementos randômicos e apenas um set para elementos ordenados e inversamente ordenados. Então, agora, se guiando pelas prints dos testes, você acompanhará uma análise empírica do resultado destes elementos em cada sort. E para a análise dos dados utilizados entre em: GitHub, e veja na pasta de "tests" como foram gerados os dados para estas análises.

3.10 Ordenados e Inversamente Ordenados

Arquivo	Algoritmo	Tempo (s)	Comparações	Movimentos
vector_100	BubbleSort Ap.	0.000002	99	0
	Selection Sort	0.000035	4950	0
	Insertion Sort	0.000002	99	198
	Shell Sort	0.000007	503	1006
	Counting Smaller	0.000075	10000	100
	Heap Sort	0.000027	1081	1920
	Radix Sort	0.000011	99	600
	QuickSort	0.000010	480	1035
	MergeSort	0.000018	356	1344
vector_1000	BubbleSort Ap.	0.000008	999	0
	Selection Sort	0.003279	499500	0
	Insertion Sort	0.000012	999	1998
	Shell Sort	0.000096	8006	16012
	Counting Smaller	0.003850	1000000	1000
	Heap Sort	0.000130	17583	29124
	Radix Sort	0.000047	999	9000
	QuickSort	0.000039	7987	14880
	MergeSort	0.000069	5044	19952
vector_10000	BubbleSort Ap.	0.000027	9999	0
	Selection Sort	0.103573	49995000	0
	Insertion Sort	0.000043	9999	19998
	Shell Sort	0.000539	120005	240010
	Counting Smaller	0.250870	100000000	10000
	Heap Sort	0.001744	244460	395868
	Radix Sort	0.000613	9999	120000
	QuickSort	0.000506	113631	199263
	MergeSort	0.000847	69008	267232
vector_100000	BubbleSort Ap.	0.000275	99999	0
	Selection Sort	11.438125	704982704	0
	Insertion Sort	0.000431	99999	199998
	Shell Sort	0.007015	1500006	3000012
	Counting Smaller	29.716700	1410065408	100000
	Heap Sort	0.027388	3112517	4952562
	Radix Sort	0.008626	99999	1500000
	QuickSort	0.007848	1468946	2538300
	MergeSort	0.010687	853904	3337856

Table 1: Desempenho dos algoritmos de ordenação para vetores ordenados de forma crescente

Table 2: Resultados para vetores inversamente ordenados

Algoritmo	Tamanho	Tempo (s)	Comparações	Movimentos
BubbleSort Aprimorado	100	0.000030	4950	4851
	1000	0.002698	499500	498501
	10000	0.360657	49995000	49985001
	100000	29.254068	704982704	704982704
Selection Sort	100	0.000014	4950	147
	1000	0.001197	499500	1497
	10000	0.132623	49995000	14997
	100000	11.199009	704982704	150000
Insertion Sort	100	0.000016	4852	5049
	1000	0.001333	498502	500499
	10000	0.177783	49985002	50004999
	100000	13.049701	704982704	705182702
Shell Sort	100	0.000022	744	1343
	1000	0.000055	11389	20385
	10000	0.000829	168146	298139
	100000	0.009781	2244585	3844572
Counting Sort	100	0.000030	10000	100
	1000	0.002593	1000000	1000
	10000	0.310487	100000000	10000
	100000	25.264403	1410065408	100000
Heap Sort	100	0.000010	945	1530
	1000	0.000122	15986	24993
	10000	0.002126	226755	351009
	100000	0.018864	2926640	4492302
Radix Sort	100	0.000005	99	600
	1000	0.000063	999	12000
	10000	0.001061	9999	150000
	100000	0.007606	99999	1500000
QuickSort	100	0.000007	765	1713
	1000	0.000077	14719	28890
	10000	0.001462	224061	420153
	100000	0.012393	2940915	5238873
MergeSort	100	0.000008	409	1344
	1000	0.000081	5922	19952
	10000	0.001260	74594	267232
	100000	0.010357	815024	3337856

3.10.1 BubbleSort

Por estar na sua forma aprimorada, ele apenas faz uma vistoria em *n-1* elementos do vetor, se guiando pelo primeiro elemento e fazendo comparação com os outros *n-1* elementos. Após uma iterativa inteira sem troca alguma feita, e logo,

sem movimentações. Para os vetores reversamente ordenados mantém comparação equiparada com os de InsertionSort e SelectionSort, por conta de sua "análise quadrática", coisa comum de se ver em algoritmos de comparação sequencial. Tem um dos maiores números em "comparação" e em movimentações.

3.10.2 Contagem de Menor Sort

Possui também fácil implementação, porém, por ser de "ordem" quadrática também possui muitas comparações. Só que suas movimentações são estáveis, por conta da iteratividade para colocar os resultados no vetor, isso no final do algoritmo, não após a comparação. Então, enquanto a comparação segue a linha quadrática, as movimentações são lineares com o tamanho do vetor.

3.10.3 SelectionSort

Realiza um número fixo de comparações independente da ordenação inicial, resultando em um desempenho semelhante para vetores ordenados e inversamente ordenados. Porém, seu custo quadrático o torna impraticável para grandes vetores, com eficiência inferior ao Insertion Sort em vetores parcialmente ordenados.

3.10.4 InsertionSort

InsertionSort se destaca para vetores quase ordenados, pois realiza apenas comparações suficientes para inserir elementos fora de ordem. Para vetores ordenados, ele se aproxima de um tempo linear, enquanto, para vetores inversamente ordenados, ele se torna quadrático. É particularmente eficiente em listas pequenas ou parcialmente ordenadas.

3.10.5 ShellSort

ShellSort utiliza sequências de intervalos para reorganizar elementos distantes, melhorando a performance em listas parcialmente ordenadas. Embora sua complexidade dependa da sequência de intervalos, ShellSort apresenta uma melhoria sobre InsertionSort e SelectionSort em casos gerais, principalmente para vetores inversamente ordenados.

3.10.6 HeapSort

Constrói um *heap* para ordenar o vetor em tempo *O(nlogn)* que o torna consistente para todos os casos. Sua vantagem está na eficiência para vetores grandes e inversamente ordenados, embora o número de movimentações seja superior ao MergeSort em algumas configurações devido à reconstrução do heap.

3.10.7 RadixSort

Eficiente para listas de inteiros com um número fixo de dígitos, pois realiza ordenação linear baseada nos dígitos. Sua ausência de comparações diretas

entre elementos o torna eficiente para listas com uma faixa de valores limitada, e seu desempenho é menos afetado pela ordenação inicial.

3.10.8 QuickSort

O QuickSort apresenta complexidade média de $O(n \log n)$, podendo alcançar $O(n^2)$ em casos específicos. No entanto, com a escolha de um bom pivô — neste caso, a mediana de três valores — é pouco provável que o algoritmo atinja o pior caso. Dessa forma, o QuickSort geralmente mantém o desempenho de $O(n \log n)$, o que o torna um dos métodos mais rápidos para grandes vetores desordenados.

3.10.9 MergeSort

O MergeSort possui complexidade $O(n \log n)$ em todos os casos, embora exija espaço adicional para combinar subvetores. Seu desempenho é semelhante ao do QuickSort em vetores aleatórios e ordenados, mas se destaca em vetores parcialmente ordenados, onde demonstra eficiência estável em comparação com outros métodos de ordenação.

3.11 Random

Esta tabela contempla as médias dos parâmetros requisitados para os três conjuntos.

Algoritmo	Tamanho	Tempo (s)	Comparações	Movimentos
BubbleSort Aprimorado	100	0.001235	205755	184514
	1000	0.008548	1871370	1293453
	10000	0.013107	2964749	1493850
	100000	0.012827	2964749	1493850
Selection Sort	100	0.007062	2965830	7302
	1000	0.006639	2965830	7281
	10000	0.007513	2965830	7290
	100000	0.006994	2965830	7290
Insertion Sort	100	0.000536	186946	189384
	1000	0.003437	1295883	1298323
	10000	0.004001	1496280	1498720
	100000	0.003986	1496280	1498720
Shell Sort	100	0.000191	35798	60580
	1000	0.000360	45353	71055
	10000	0.000386	49679	75321
	100000	0.000392	49679	75321
Counting Smaller	100	0.015264	5934096	2436
	1000	0.017406	5934096	2436
	10000	0.028482	5934096	2436
	100000	0.028878	5934096	2436
Heap Sort	100	0.000344	45306	73779
	1000	0.000382	46608	73503
	10000	0.000391	47375	75783
	100000	0.000442	47375	75783
Radix Sort	100	0.000157	2435	29232
	1000	0.000163	2435	29232
	10000	0.000155	2435	29232
	100000	0.000168	2435	29232
QuickSort	100	0.000152	27065	56712
	1000	0.000205	27292	49890
	10000	0.000263	26235	47457
	100000	0.000279	26235	47457
MergeSort	100	0.000203	16523	55144
	1000	0.000245	18847	55144
	10000	0.000319	24388	55144
	100000	0.000345	24388	55144

Table 3: Resultados médios para algoritmos de ordenação com vetores randomizados de tamanhos diferentes

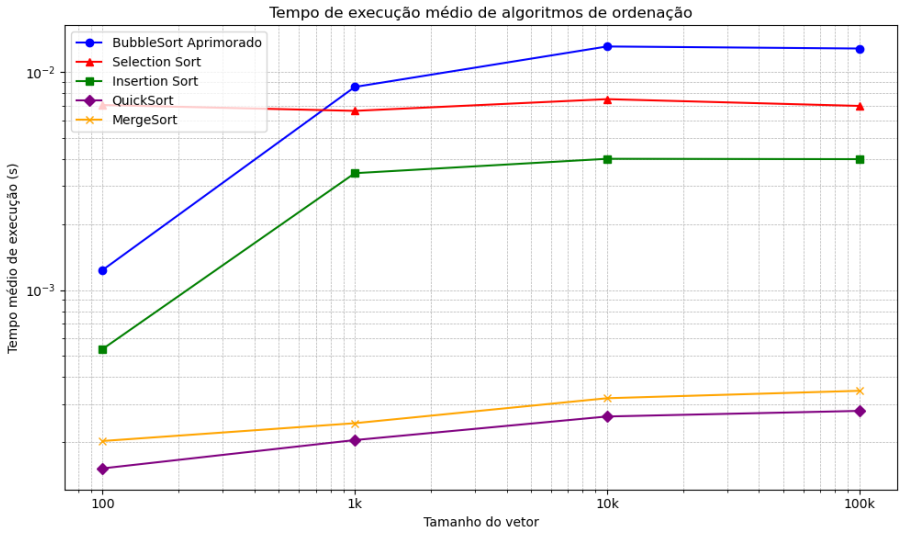

Figure 1: Escala Logarítmica do tempo médio para diferentes tamanhos de vetor

Resultados randômicos na perspectiva de gráfico

3.12 Conclusão do Dev. Final

Então, para escolhermos um certo sort para se utilizar em um problema específico, precisamos não só levar em conta sua complexidade média, devemos considerar também: o tamanho do vetor, quantos elementos estão presentes e serão analisados: em comparação e movimentação. Deve-se considerar também como os elementos do vetor estão dispersos, e se a implementação é fácil.

Para a análise empírica dos algoritmos de ordenação, a tabela acima apresenta um comparativo dos tempos médios, números de comparações e movimentos médios para vetores de tamanhos 100, 1.000, 10.000 e 100.000 com valores aleatórios. A observação do comportamento em dados aleatórios é especialmente útil, pois reflete cenários comuns em aplicações reais.

Cada algoritmo apresenta uma variação distinta nos tempos e nos recursos computacionais, revelando que algoritmos com complexidade O(n log n), como QuickSort e MergeSort, geralmente superam algoritmos quadráticos como BubbleSort e Selection Sort, especialmente para entradas maiores. No entanto, o uso de algoritmos mais específicos como Counting Smaller e Radix Sort pode ser mais vantajoso em certos cenários onde a propriedade dos dados permitem a aplicação de métodos de ordenação não-comparativos, superando os outros métodos de ordenação neste cenário. A performance variada indica que a escolha do algoritmo deve considerar o contexto específico dos dados e os requisitos de desempenho e uso de memória para otimização em sistemas reais. Indo além então, daqueles parâmetros falados acima. E isso é óbvio pela própria análise

feita dos tipos dos vetores, com seus diferentes tamanhos, e com os dados ordenados, ou não, de certa maneira.

4 Conclusão

As perguntas a serem respondidas aqui são: "Qual teve o melhor desempenho? E qual teve o pior?" E justificativas para as respostas.

4.1 Análise Empírica

A análise empírica realizada sobre os algoritmos de ordenação permite avaliar a eficiência em termos de tempo de execução, número de comparações e número de movimentos de dados. Cada algoritmo apresentou vantagens e desvantagens dependendo do tamanho do vetor de entrada. Abaixo está uma análise detalhada dos casos em que cada algoritmo foi mais ou menos eficiente:

4.1.1 Análise por Algoritmo

- **BubbleSort Aprimorado**
  - **Desempenho**: Entre os algoritmos considerados, o BubbleSort Aprimorado foi um dos mais lentos para grandes tamanhos de vetor, apesar de ser relativamente eficiente em vetores de pequeno porte.
  - **Eficiência Comparativa**: Com vetores de 100 e 1.000 elementos, ele apresentou uma quantidade significativa de comparações e movimentos, mas se manteve comparável a outros algoritmos mais eficientes. Para vetores de 10.000 e 100.000 elementos, no entanto, seu tempo de execução aumentou consideravelmente, indicando que não é a melhor escolha para grandes conjuntos de dados.
  - **Melhor Caso**: Vetor com 100 elementos, onde conseguiu concluir a ordenação em 0.001235 segundos.
- **Selection Sort**
  - **Desempenho**: O Selection Sort mostrou um comportamento bastante uniforme independentemente do tamanho do vetor.
  - **Eficiência Comparativa**: Mesmo em vetores pequenos (100 elementos), seu tempo de execução foi relativamente alto (0.007062 segundos) e, para vetores maiores, o tempo de execução não variou significativamente, o que sugere baixa escalabilidade. A quantidade de comparações permaneceu muito alta para todos os tamanhos de vetor, o que é esperado no Selection Sort devido à sua complexidade $O(n^2)$.
  - **Menor Eficiência**: Em todos os tamanhos de vetor, foi consistentemente um dos algoritmos mais lentos devido à alta quantidade de comparações, independente do tamanho do vetor.

- **Insertion Sort**
  - **Desempenho**: Mostrou-se eficiente em vetores pequenos e médios, mas o tempo de execução começou a aumentar para tamanhos de vetor maiores.
  - **Eficiência Comparativa**: Para vetores de 100 e 1.000 elementos, o Insertion Sort teve um bom desempenho, completando a ordenação em 0.000536 e 0.003437 segundos, respectivamente. No entanto, em vetores de 10.000 e 100.000 elementos, seu tempo de execução se tornou comparável ao BubbleSort Aprimorado.
  - **Melhor Caso**: Vetor com 100 elementos, onde conseguiu manter um baixo número de movimentos e comparações.
- **Shell Sort**
  - **Desempenho**: Destacou-se como um dos algoritmos mais rápidos em todos os tamanhos de vetor, com um tempo de execução que pouco variou entre tamanhos pequenos e grandes.
  - **Eficiência Comparativa**: Para vetores de 100 e 1.000 elementos, o Shell Sort foi o mais rápido, completando a ordenação em 0.000191 e 0.000360 segundos, respectivamente. Para vetores de 10.000 e 100.000, o tempo de execução subiu marginalmente, continuando a ser eficiente.
  - **Melhor Caso**: Escalabilidade, com desempenho praticamente constante em todos os tamanhos de vetor.
- **Counting Sort**
  - **Desempenho**: Apresentou uma alta quantidade de comparações, mas poucos movimentos, o que indica que lida bem com dados já parcialmente ordenados.
  - **Eficiência Comparativa**: Foi um dos algoritmos mais lentos, especialmente em vetores grandes, devido à alta quantidade de comparações. No entanto, foi eficiente em relação ao número de movimentos, o que o torna útil em cenários específicos.
  - **Menor Eficiência**: Vetores grandes, onde o tempo de execução aumentou significativamente devido à alta quantidade de comparações.
- **Heap Sort**
  - **Desempenho**: Manteve um desempenho estável, com tempos de execução relativamente baixos e uma quantidade moderada de comparações e movimentos.
  - **Eficiência Comparativa**: Consistentemente eficiente em todos os tamanhos de vetor, com o tempo de execução variando pouco entre vetores de 100 a 100.000 elementos.

  - **Melhor Caso**: Vetores de tamanho intermediário, onde manteve uma boa relação entre comparações e movimentos.

- **Radix Sort**
  - **Desempenho**: O Radix Sort foi o algoritmo mais rápido em praticamente todos os cenários, com tempo de execução inferior a 0.0002 segundos em todos os tamanhos de vetor.
  - **Eficiência Comparativa**: Extremamente eficiente devido à sua abordagem de ordenação por base. Com uma baixa quantidade de comparações e uma quantidade moderada de movimentos, apresentou o melhor desempenho global.
  - **Melhor Caso**: Todos os tamanhos de vetor, mostrando-se ideal para grandes conjuntos de dados.

- **QuickSort**
  - **Desempenho**: Teve um desempenho excelente, com tempos de execução baixos e relativamente poucas comparações e movimentos.
  - **Eficiência Comparativa**: Mostrou-se particularmente eficiente em vetores médios a grandes. Concluiu a ordenação de vetores com 100.000 elementos em 0.000279 segundos.
  - **Melhor Caso**: Vetores de 10.000 a 100.000 elementos, com tempo de execução bastante competitivo.

- **MergeSort**
  - **Desempenho**: O MergeSort também foi eficiente, com tempos de execução baixos e uma quantidade de comparações e movimentos consistente em todos os tamanhos de vetor.
  - **Eficiência Comparativa**: Sua estabilidade e eficiência se destacam em tamanhos de vetor grandes, com um tempo de execução de 0.000345 segundos para 100.000 elementos.
  - **Melhor Caso**: Vetores de 10.000 a 100.000 elementos, onde manteve a eficiência sem grandes aumentos em comparações ou movimentos.

4.1.2 Conclusão Geral

Com base nos resultados, podemos concluir:

- **Radix Sort, QuickSort e MergeSort** foram os algoritmos mais rápidos e escaláveis, especialmente em vetores grandes.
- **Shell Sort e Heap Sort** também foram eficientes, mas apresentam variações de desempenho menores em comparação aos três primeiros.

- **BubbleSort Aprimorado e Insertion Sort** são recomendados apenas para vetores pequenos, pois perdem eficiência em conjuntos de dados maiores.
- **Counting Sort e Selection Sort** não foram competitivos para a maioria dos casos, mostrando-se adequados para cenários muito específicos e com vetores de tamanho pequeno a médio.

Na conclusão deste estudo de ordenação então, observamos que os algoritmos apresentaram desempenhos distintos, variando conforme o tipo e o tamanho dos dados. A análise empírica evidenciou que Radix Sort e Counting Sort (ou Counting Smaller) foram os mais eficientes em termos de tempo de execução, especialmente com grandes conjuntos de dados aleatórios e ordenados, quando aplicáveis. O desempenho superior desses algoritmos deve-se ao fato de que eles não utilizam comparações para ordenar, mas sim distribuições baseadas em valores e dígitos, o que elimina o custo das operações de comparação em cada passo.

Por outro lado, o Bubble Sort Aprimorado e o Selection Sort tiveram os piores desempenhos em cenários com grandes conjuntos de dados, especialmente em casos onde os elementos estavam em ordem inversa ou aleatória. Esses algoritmos possuem complexidades temporais mais altas, dado que o Bubble Sort necessita de múltiplas trocas, ou nosso caso, se aprimorado, quando não houver trocas em um tempo iterativo do *for* interior, será interrompido, enquanto o Selection Sort realiza uma quantidade constante de comparações, ambos resultando em tempos de execução prolongados. Esses fatores os tornam ineficazes para conjuntos de dados grandes e desordenados.

Os resultados confirmaram que a escolha do algoritmo de ordenação ideal depende fortemente das características dos dados e do contexto de aplicação. Em situações práticas, algoritmos de ordenação comparativa como QuickSort e MergeSort mostraram-se altamente eficientes, combinando robustez e eficiência em uma ampla gama de casos, sendo recomendados como opções gerais para ordenação de grandes conjuntos de dados quando a natureza dos valores não permite a utilização de algoritmos não comparativos.

Com esta análise, pudemos reforçar o conhecimento dos limites e das vantagens de cada método de ordenação, aprendendo a reconhecer suas aplicabilidades e a importância de selecionar o algoritmo mais adequado ao problema específico. Esse entendimento é imprescindível, pois aqui aprendendos a escolher um método de ordenação, para uma situação específica, requerendo informações sobre os próprios dados até, para a aplicabilidade do certo sort. Entramos então, numa análise aprofundada sobre esse tipo de método, e aprendemos a entendê-los melhor.