Implementação e comparação empírica de algoritmos de ordenação

Enzo Zanetti Camargo Penteado
Nº USP 15574558

Novembro de 2024

1 ARQUITETURA DA MÁQUINA UTILIZADA

Fabricante: Acer
Modelo: Nitro 5
Processador: AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz
Memória RAM: 8,00 GB (utilizável: 7,36 GB)
Sistema Operacional: Windows 11 (64 bits)

2 METODOLOGIA

Nesse projeto iremos testar 8 algoritmos de ordenação diferentes, em tempo de execução, número de comparações e número de movimentações. Os testes foram feitos utilizando vetores de structs de dois tipos, sendo o primeiro:

```
typedef struct {
    int chave;
    int campoDaEstrutura[1];
} Registro1;
```

Onde iremos ordenar o campo chave da struct, e o campoDaEstrutura será apenas para testes posteriormente. O segundo tipo de struct utilizado é:

```
typedef struct {
    int chave;
    int campoDaEstrutura[1000];
} Registro1000;
```

Onde também iremos ordenar o campo chave da struct, e o campoDaEstrutura será apenas para testes posteriormente.
Abaixo estão os algoritmos utilizados e uma breve explicação sobre seus funcionamentos:

- **Insertion Sort:** Percorre a lista e insere cada elemento em sua posição correta em uma sublista ordenada, expandindo-a progressivamente. Foi utilizado a implementação com sentinela

- **Selection Sort:** Seleciona repetidamente o menor elemento da lista desordenada e o move para sua posição correta.
- **Bubble Sort Melhorado:** Compara pares de elementos adjacentes e os troca se estiverem fora de ordem. Isso é repetido até que a lista esteja ordenada, com os maiores elementos "subindo" para o final da lista, além disso, para cada passagem, verifica se a lista já está ordenada, interrompendo o processo se não houver trocas
- **Merge Sort:** Divide a lista recursivamente em sublistas menores, ordena-as individualmente e as combina em ordem.
- **Quick Sort Aleatório:** Escolhe um elemento pivô (aleatório neste caso), particiona a lista em relação a ele, e ordena recursivamente as sublistas resultantes.
- **Shell Sort:** Usa inserção direta em sublistas (ou gaps) que vão sendo reduzidos até 1, melhorando a eficiência do Insertion Sort para listas maiores.
- **Heap Sort:** Usa uma estrutura de heap (max-heap ou min-heap) para selecionar o maior ou menor elemento e reorganizar a lista iterativamente.
- **Radix Sort utilizando Counting Sort:** Ordena os números com base em cada dígito, começando do menos significativo. Usa o Counting Sort para organizar os números por posição de cada dígito.

Serão feitos diversos testes sobre o tempo de execução, número de movimentações e número de comparações dos algoritmos de ordenação.
Além disso, vale ressaltar que para resultados sobre tempo de execução, a unidade de medida utilizada é milisegundo, e para gráficos, utilizaremos uma escala logarítmica para facilitar a visualização do mesmo.

3 GRÁFICOS E TABELAS OBTIDOS

3.1 Tempo de execução por número de chaves

	Vetor ordenado crescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	0.0006 ms	0.0030 ms	0.0452 ms
SelectionSort	0.0084 ms	0.7578 ms	66.2234 ms
BubbleSort Melhorado	0.0002 ms	0.0020 ms	0.0162 ms
MergeSort	0.0072 ms	0.0832 ms	0.9266 ms
QuickSort Aleatório	0.0058 ms	0.0720 ms	0.8108 ms
ShellSort	0.0020 ms	0.0220 ms	0.2730 ms
HeapSort	0.0060 ms	0.1124 ms	1.0566 ms
RadixSort com CountingSort	0.0122 ms	0.1160 ms	1.0470 ms

Tabela 1: Vetor Crescente e campoDaEstrutura = 1

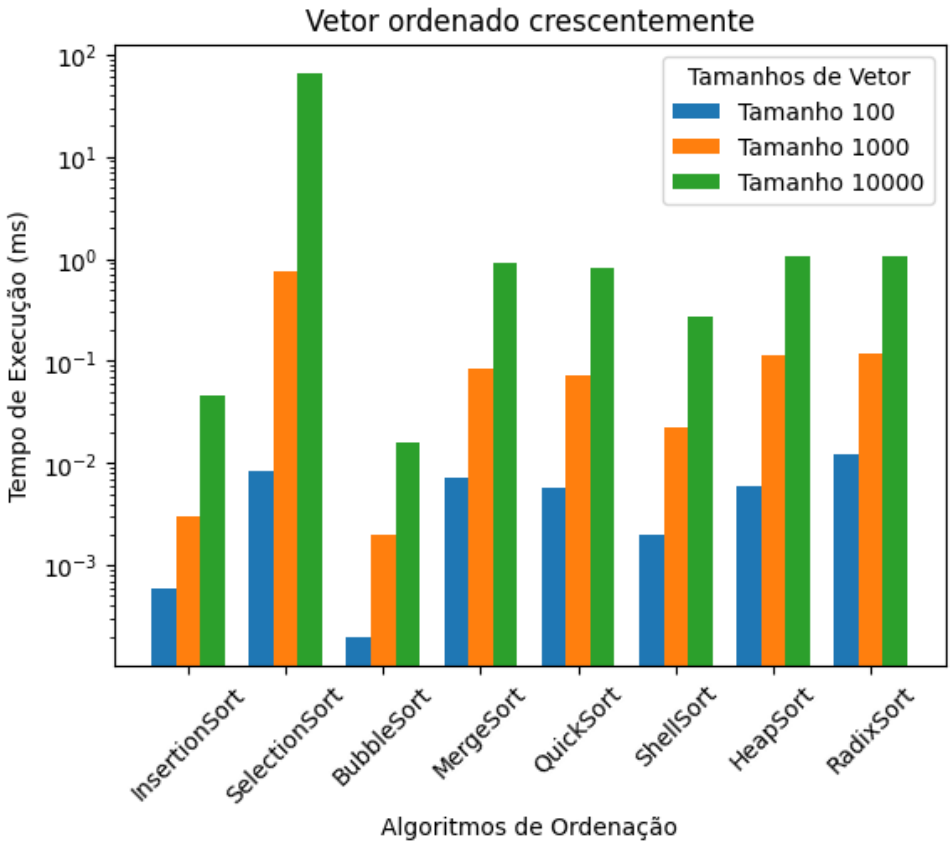

Gráfico 1: Vetor Crescente, campoDaEstrutura = 1, escala logarítmica

	Vetor ordenado aleatoriamente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	0.0072 ms	0.5316 ms	54.1192 ms
SelectionSort	0.0098 ms	0.8258 ms	66.2260 ms
BubbleSort Melhorado	0.0234 ms	1.5456 ms	136.3010 ms
MergeSort	0.0080 ms	0.1046 ms	1.5570 ms
QuickSort Aleatório	0.0078 ms	0.0972 ms	1.2366 ms
ShellSort	0.0038 ms	0.0768 ms	1.5896 ms
HeapSort	0.0062 ms	0.1074 ms	1.4530 ms
RadixSort com CountingSort	0.0122 ms	0.1172 ms	1.1558 ms

Tabela 2: Vetor Aleatório e campoDaEstrutura = 1

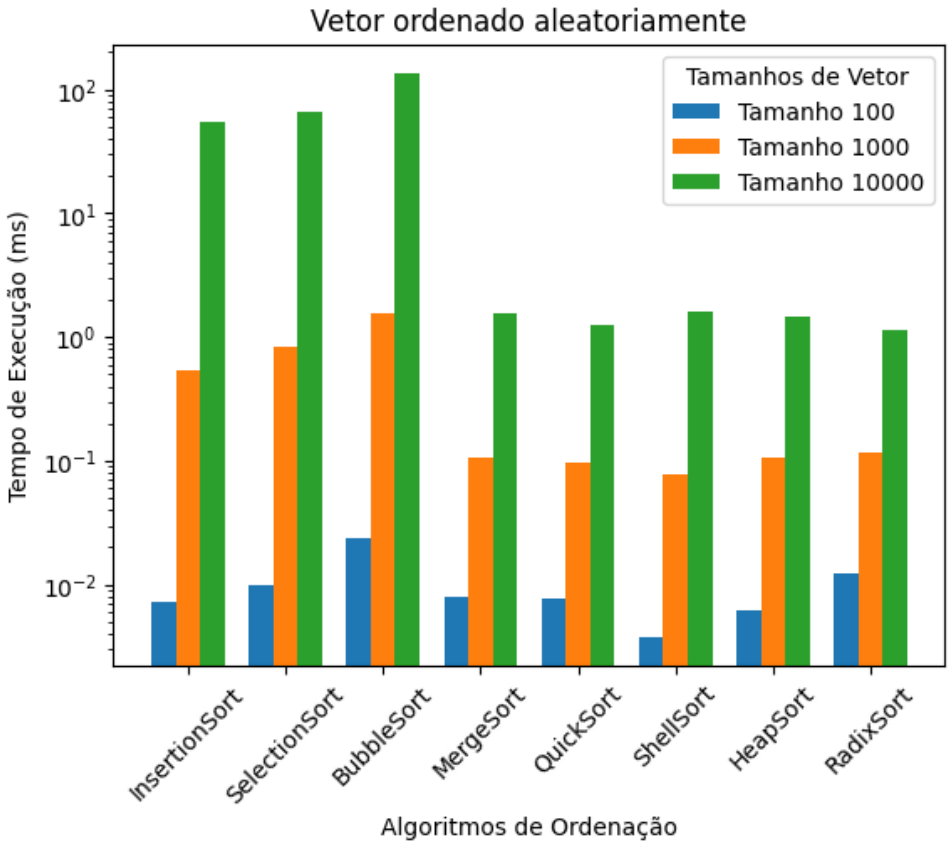

Gráfico 2: Vetor Aleatório, campoDaEstrutura = 1, escala logarítmica

	Vetor ordenado decrescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	0.0112 ms	1.3806 ms	107.6886 ms
SelectionSort	0.0094 ms	0.8420 ms	72.0094 ms
BubbleSort Melhorado	0.0208 ms	1.7444 ms	176.3340 ms
MergeSort	0.0070 ms	0.0892 ms	0.8794 ms
QuickSort Aleatório	0.0062 ms	0.0750 ms	0.8098 ms
ShellSort	0.0030 ms	0.0364 ms	0.4524 ms
HeapSort	0.0040 ms	0.0932 ms	1.0390 ms
RadixSort com CountingSort	0.0122 ms	0.1152 ms	1.0012 ms

Tabela 3: Vetor Derescente e campoDaEstrutura = 1

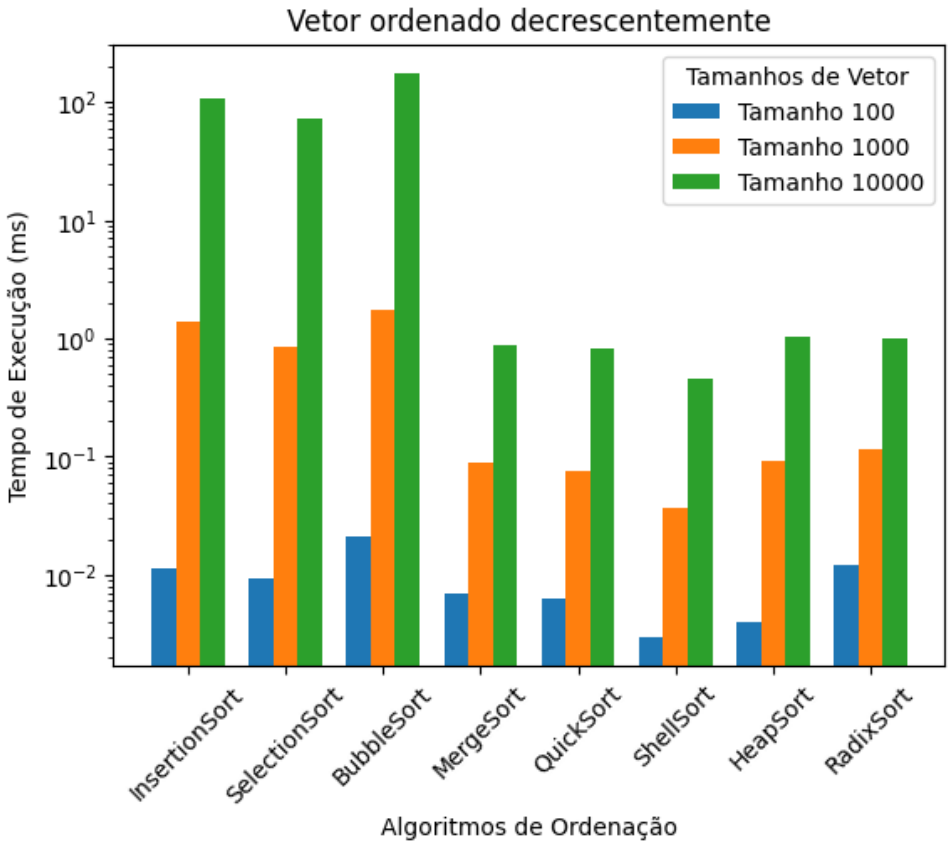

Gráfico 3: Vetor Decrescente, campoDaEstrutura = 1, escala logarítmica

3.2 Tempo de execução por tamanho do campoDaEstrutura

	Vetor ordenado crescentemente	
Algoritmos\\\\CampoDaEstrutura	1	1000
InsertionSort	0.0030 ms	0.0048 ms
SelectionSort	0.7578 ms	1.0866 ms
BubbleSort Melhorado	0.0020 ms	0.0038 ms
MergeSort	0.0832 ms	0.0902 ms
QuickSort Aleatório	0.0720 ms	1.3672 ms
ShellSort	0.0220 ms	0.0254 ms
HeapSort	0.1124 ms	0.1118 ms
RadixSort com CountingSort	0.1160 ms	0.3936 ms

Tabela 4: Vetor Crescente e tamanho do vetor = 1000

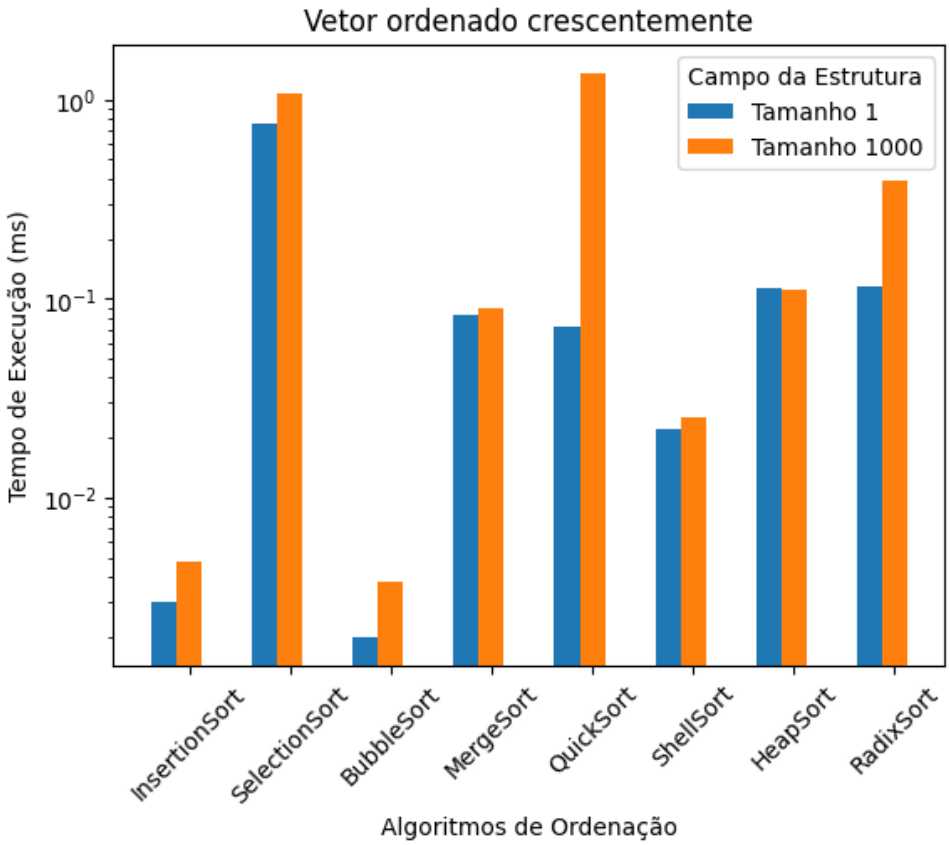

Gráfico 4: Vetor Crescente, tamanho do vetor = 1000 e escala logarítimica

	Vetor ordenado aleatoriamente	
Algoritmos\\\\CampoDaEstrutura	1	1000
InsertionSort	0.5316 ms	0.6098 ms
SelectionSort	0.8258 ms	1.4646 ms
BubbleSort Melhorado	1.5456 ms	1.7124 ms
MergeSort	0.1046 ms	0.1050 ms
QuickSort Aleatório	0.0972 ms	1.7482 ms
ShellSort	0.0768 ms	0.0852 ms
HeapSort	0.1074 ms	0.1160 ms
RadixSort com CountingSort	0.1172 ms	0.8750 ms

Tabela 5: Vetor Aleatorio e tamanho do vetor = 1000

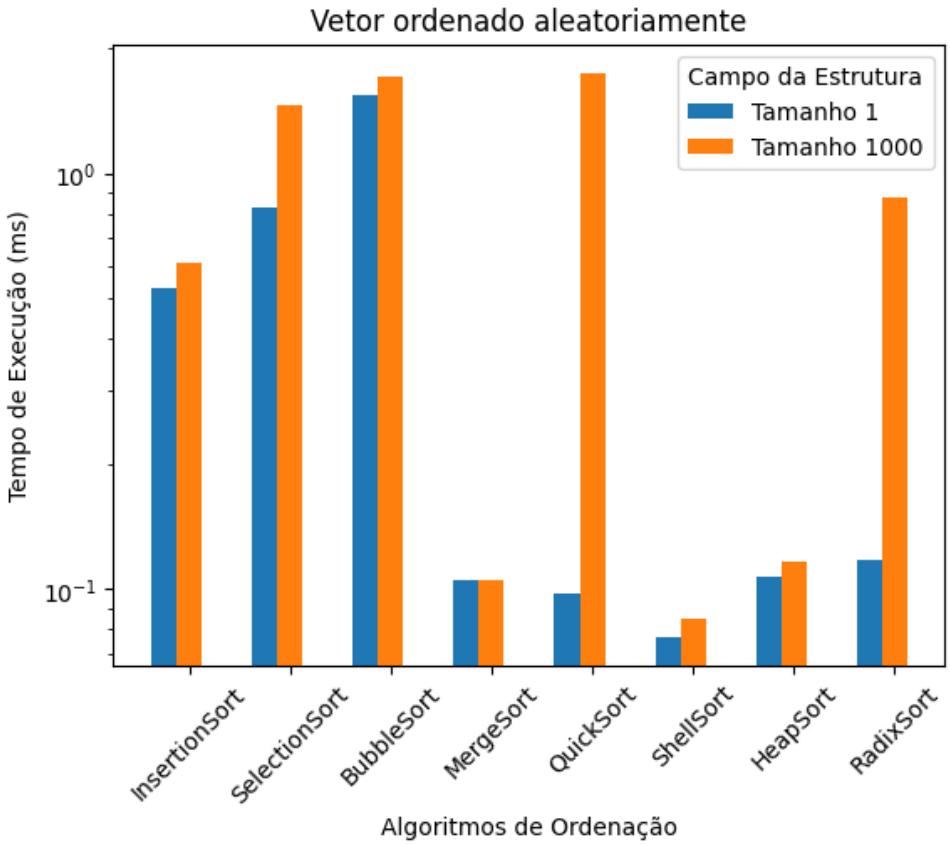

Gráfico 5: Vetor Aleatório, tamanho do vetor = 1000 e escala logarítimica

	Vetor ordenado decrescentemente	
Algoritmos\\\\CampoDaEstrutura	1	1000
InsertionSort	1.3806 ms	1.1970 ms
SelectionSort	0.8420 ms	1.0856 ms
BubbleSort Melhorado	1.7444 ms	1.8434 ms
MergeSort	0.0892 ms	0.0910 ms
QuickSort Aleatório	0.0750 ms	1.4554 ms
ShellSort	0.0364 ms	0.0416 ms
HeapSort	0.0932 ms	0.1048 ms
RadixSort com CountingSort	0.1152 ms	0.2506 ms

Tabela 6: Vetor Decrescente e tamanho do vetor = 1000

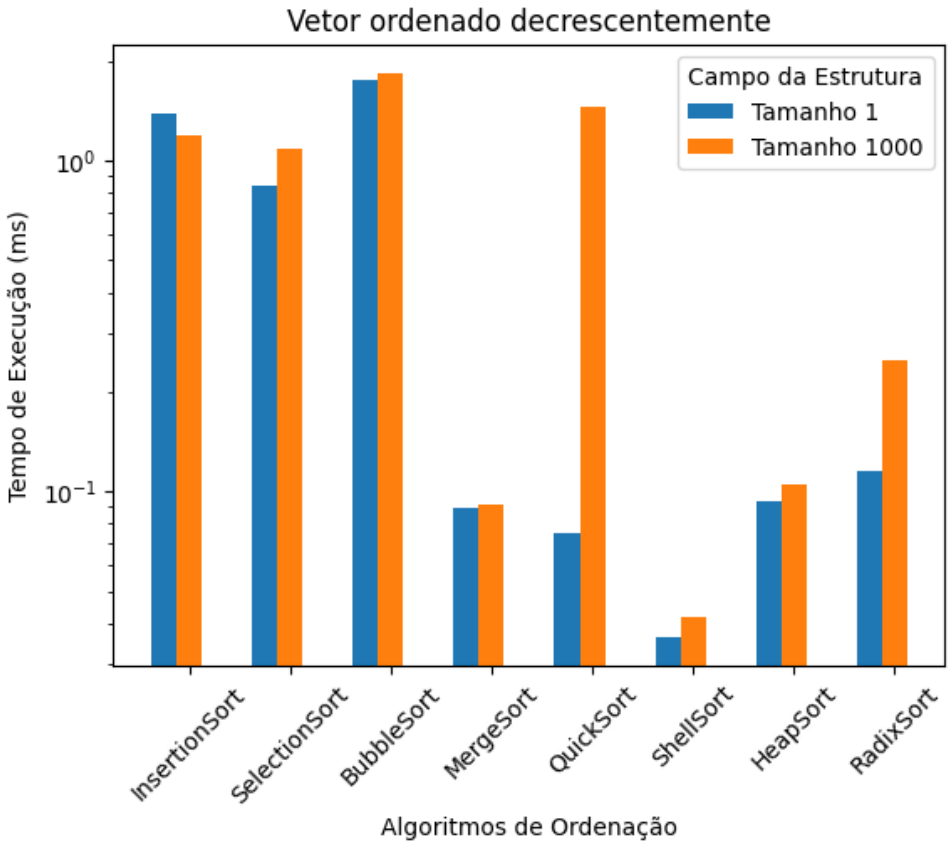

Gráfico 6: Vetor Decrescente, tamanho do vetor = 1000 e escala logarítimica

3.3 Número de comparações por número de chaves

	Vetor ordenado crescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	108	1099	10981
SelectionSort	4950	499500	49995000
BubbleSort Melhorado	99	999	9999
MergeSort	672	9976	133616
QuickSort Aleatório	742	12895	175177
ShellSort	442	6457	85243
HeapSort	1926	29465	396777
RadixSort com CountingSort	99	999	9999

Tabela 7: Vetor Crescente e campoDaEstrutura = 1

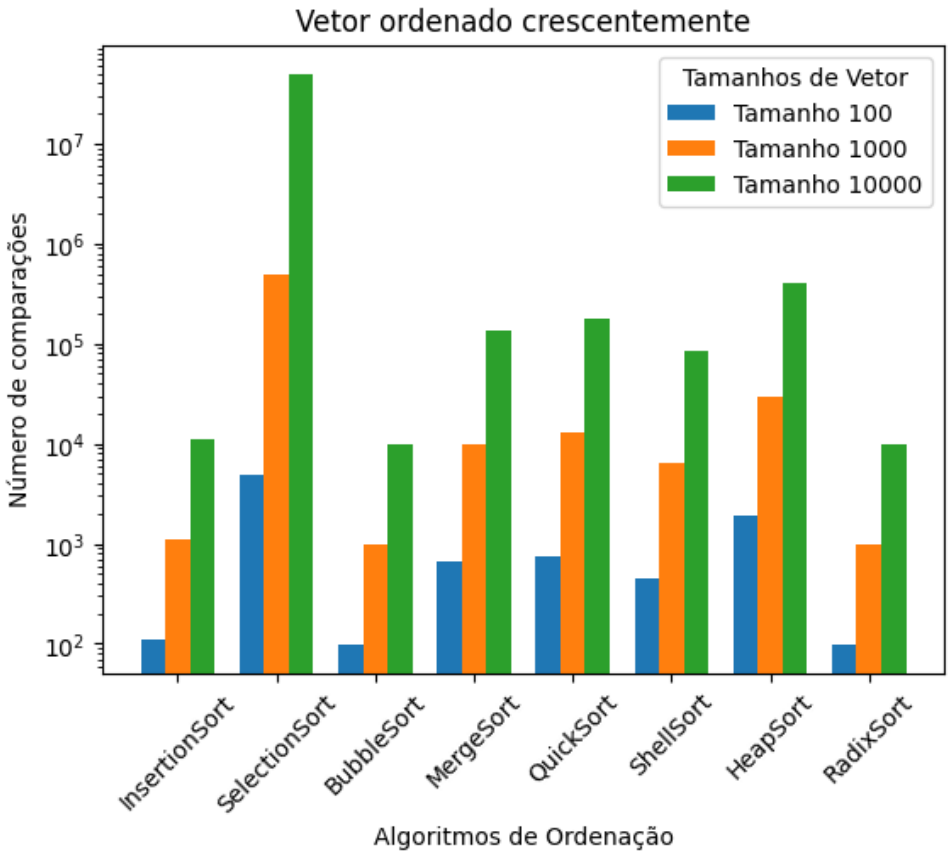

Gráfico 7: Vetor Crescente, campoDaEstrutura = 1 e escala logarítimica

	Vetor ordenado aleatoriamente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	2259	249042	25010530
SelectionSort	4950	499500	49995000
BubbleSort Melhorado	4845	497547	49984269
MergeSort	672	9976	133616
QuickSort Aleatório	802	12685	171688
ShellSort	865	15495	242615
HeapSort	1859	28185	383055
RadixSort com CountingSort	99	999	9999

Tabela 8 Vetor Aleatorio e campoDaEstrutura = 1

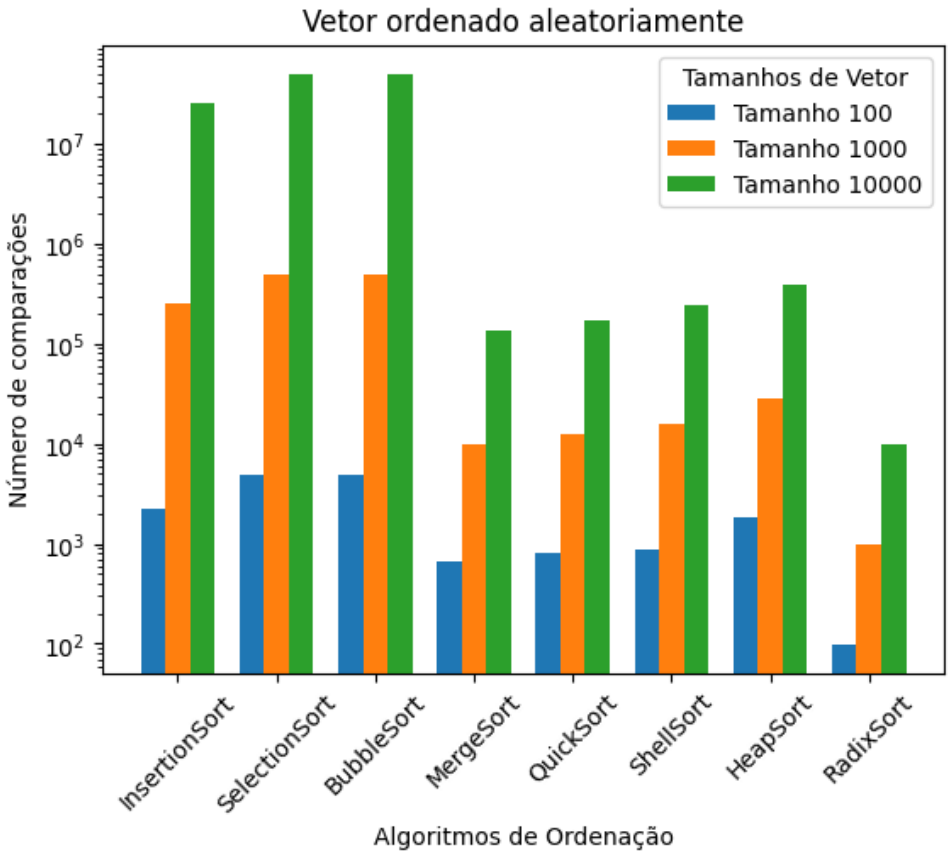

Gráfico 8: Vetor Aleatório, campoDaEstrutura = 1 e escala logarítimica

	Vetor ordenado decrescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	4950	499500	49996033
SelectionSort	4950	499500	49995000
BubbleSort Melhorado	4950	499500	49995000
MergeSort	672	9976	133616
QuickSort Aleatório	803	12345	172746
ShellSort	672	10377	138947
HeapSort	1693	26937	370078
RadixSort com CountingSort	99	999	9999

Tabela 9: Vetor Decrescente e campoDaEstrutura = 1

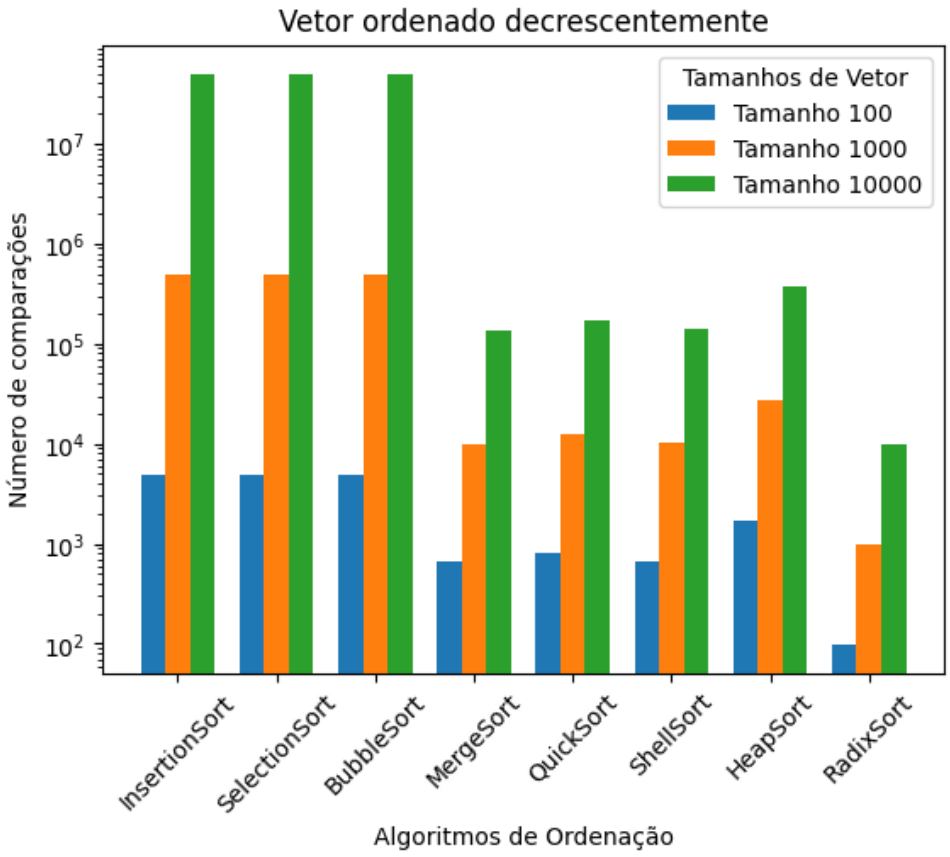

Gráfico 9: Vetor Decrescente, campoDaEstrutura = 1 e escala logarítimica

3.4 Número de movimentações por número de chaves

	Vetor ordenado crescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	207	2098	20980
SelectionSort	297	2997	29997
BubbleSort Melhorado	0	0	0
MergeSort	1344	19952	267232
QuickSort Aleatório	1428	20983	276583
ShellSort	884	12914	170486
HeapSort	2214	32130	424272
RadixSort com CountingSort	2394	23094	230094

Tabela 10: Vetor Crescente e campoDaEstrutura = 1

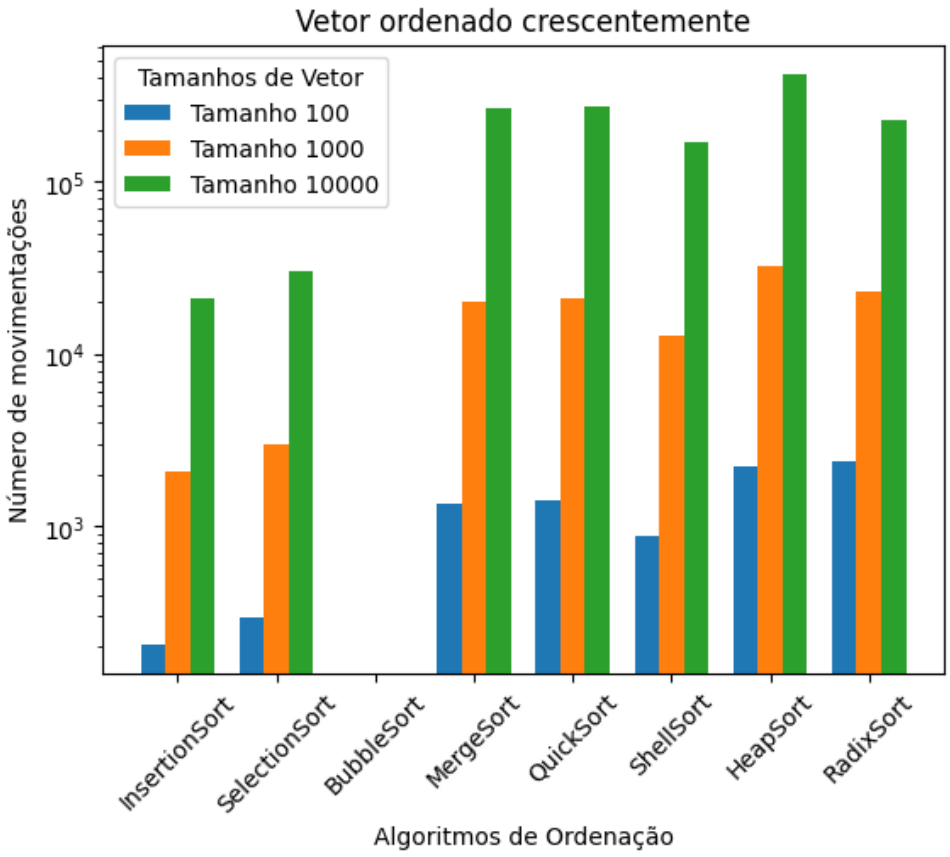

Gráfico 10: Vetor Crescente, campoDaEstrutura = 1 e escala logarítimica

	Vetor ordenado aleatoriamente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	2358	250041	25020529
SelectionSort	297	2997	29997
BubbleSort Melhorado	6525	744900	75012816
MergeSort	1344	19952	267232
QuickSort Aleatório	1432	20448	281377
ShellSort	1307	21952	327858
HeapSort	2091	30186	402636
RadixSort com CountingSort	2302	22104	220101

Tabela 11: Vetor Aleatorio e campoDaEstrutura = 1

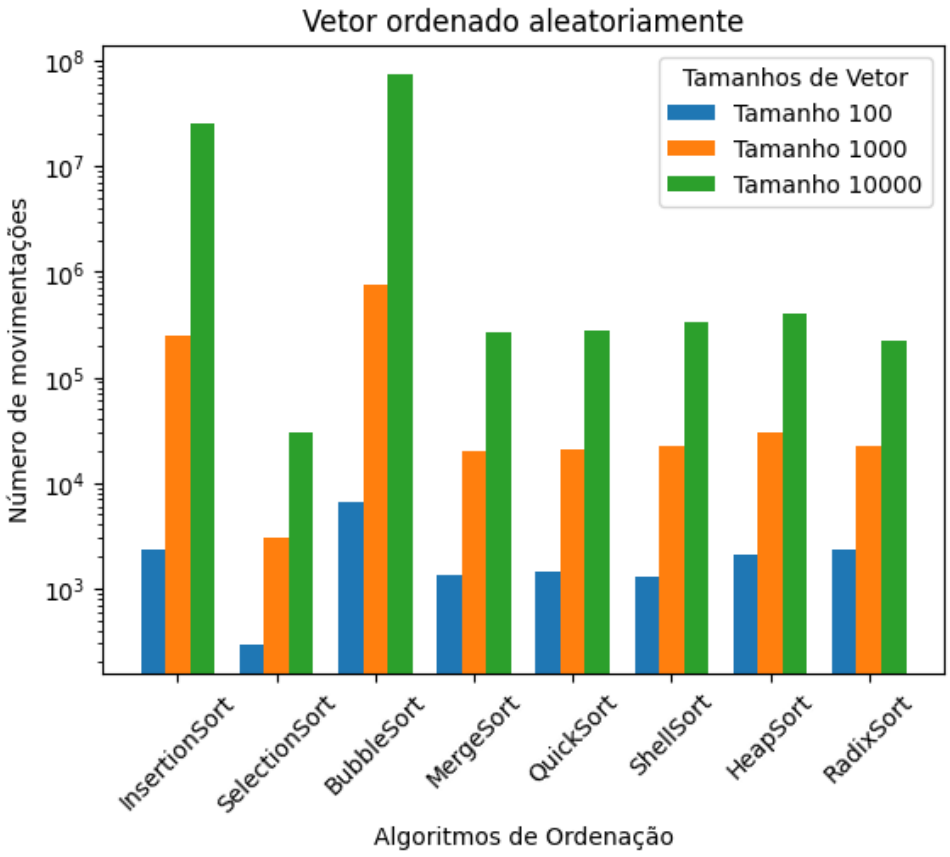

Gráfico 11: Vetor Aleatorio, campoDaEstrutura = 1 e escala logarítimica

	Vetor ordenado decrescentemente		
Algoritmos\\\\Tamanho Vetor	100	1000	10000
InsertionSort	5049	500499	50006002
SelectionSort	297	2997	29997
BubbleSort Melhorado	14850	1498500	149985000
MergeSort	1344	19952	267232
QuickSort Aleatório	1458	20132	277272
ShellSort	1114	16834	224190
HeapSort	1794	27936	380928
RadixSort com CountingSort	2295	22095	220095

Tabela 12: Vetor Decrescente e campoDaEstrutura = 1

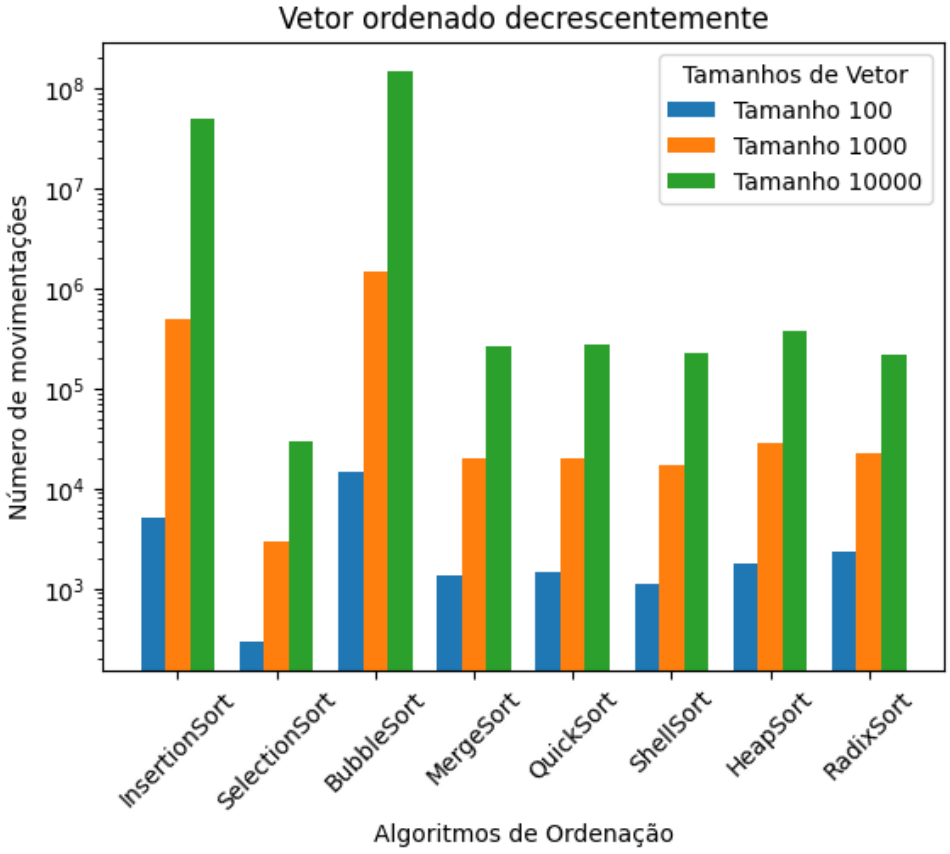

Gráfico 12: Vetor Decrescente, campoDaEstrutura = 1 e escala logarítimica

4 DISCUSSÃO DOS RESULTADOS

Para facilitar a discussão dos resultados, iremos analisar os algoritmos um a um, e verificar como ele se comportou nos testes executados. Para depois, fazer uma conclusão geral.

4.1 InsertionSort

Nos resultados observados, o InsertionSort apresentou excelente desempenho em vetores já ordenados ou quase ordenados, devido ao número reduzido de comparações e movimentações. Nessas condições, o algoritmo atinge sua melhor eficiência, com complexidade $\mathcal{O}(n)$. Entretanto, quando aplicado a vetores em ordem decrescente ou aleatória, o número de movimentações e comparações aumenta significativamente, levando a uma complexidade $\mathcal{O}(n^2)$ Isso torna o Insertion Sort inadequado para grandes conjuntos de dados desordenados. Por outro lado, é ideal em situações onde os dados já estão parcialmente ordenados ou quando o número de elementos é pequeno, destacando-se pela simplicidade de implementação. Além disso, não apresentou muita diferença quando alteramos o tamanho do campoDaEstrutura, isso é visto ao observar que para vetores ordenados decrescentemente, o struct Registro1000 apresentou valores menores do que Registro1, o que é um comportamento incomum. Vale destacar também que o InsertionSort é um dos poucos algoritmos estáveis e in loco, sendo assim, caso precise de um algoritmo que cumpra esses dois requisitos, InsertionSort pode ser uma opção.

4.2 SelectionSort

Nos testes, o SelectionSort demonstrou desempenho consistente, independentemente da disposição inicial dos dados, pois o número de comparações não varia em função da ordem do vetor. Esse comportamento reflete sua natureza determinística, onde a complexidade é sempre $\mathcal{O}(n^2)$. Porém, também é importante mencionar que em número de movimentações, o SelectionSort é sempre linear, $\mathcal{O}(n)$, o que o torna melhor do que os demais algoritmos nesse quesito. Embora, em geral, seja inferior em eficiência em comparação com os outros algoritmos, o SelectionSort possui a vantagem de realizar um número fixo e previsível de operações. No entanto, sua lentidão o torna impróprio para listas extensas. Além disso, apresentou uma diferença um pouco maior do tempo de execução quando alteramos o tamanho do campoDaEstrutura em vetores aleatórios, e uma diferença um pouco menor para vetores crescentes e decrescentes.

4.3 BubbleSort Melhorado

O Bubble Sort Melhorado se diferencia do Bubble Sort convencional por incluir uma otimização que interrompe o processamento se o vetor já estiver ordenado, fazendo assim com que ele atinja o melhor tempo para vetores ordenados crescentemente e surpreenda com 0 movimentações nesses casos. Ainda assim, mesmo com essa melhoria, os resultados indicam que ele permanece menos eficiente quando comparado a algoritmos mais avançados. Sua complexidade $\mathcal{O}(n^2)$ no caso geral o limita a pequenas listas ou a situações em que a ordenação inicial já está próxima. Apesar disso, sua implementação simples faz dele uma escolha comum em contextos didáticos para introdução de conceitos básicos de ordenação. Além disso, também mostrou um aumento pequeno quando alteramos o valor do campoDaEstrutura, com uma diferença um pouco maior em vetores ordenados de forma crescente, mas mesmo assim, teve o melhor tempo nesse caso.

4.4 MergeSort

O MergeSort demonstrou desempenho consistente e robusto, com complexidade $\mathcal{O}(n \log n)$ em todos os casos. Sua abordagem de dividir para conquistar, que divide recursivamente a lista em sublistas menores e as combina ordenadamente, o torna extremamente eficiente para grandes conjuntos de dados. No entanto, devido à necessidade de espaço adicional para as sublistas, ele pode ser menos adequado em sistemas com restrições de memória. Mesmo assim, o Merge Sort é amplamente utilizado quando a estabilidade e o desempenho são cruciais. Além de apresentar uma mudança de tempo quase insignificativa quando alteramos o valor do campoDaEstrutura, se destacando nesse quesito.

4.5 QuickSort Aleatório

Em média, o QuickSort Aleatório foi um dos algoritmos mais rápidos nos testes, especialmente em vetores aleatórios, devido à sua abordagem eficiente de divisão recursiva baseada na escolha de um pivô. Com o uso de uma escolha aleatória para o pivô, o algoritmo exclui seu pior caso de $\mathcal{O}(n^2)$, e vira $\mathcal{O}(n \log n)$ em todos os casos. Contudo, em casos dos vetores crescentes e decrescentes, o QuickSort não foi o melhor algoritmo, ficando atrás do ShellSort, mas se levarmos em conta as médias para vetores crescentes, aleatórios e decrescentes, o QuickSort apresentou o melhor resultado em geral. Por outro

lado, é visto uma desvantagem gigante do QuickSort quando alteramos o valor do campoDaEstrutura, sendo o algoritmo que mais apresentou disparidade de tempo nos testes executados. Desse modo, o QuickSort acaba pecando nesse quesito, onde nos casos que o campoDaEstrutura = 1000, ficou com tempo parecido dos algoritmos quadráticos, o que acaba atrapalhando seu funcionamento. Mas em geral, o QuickSort Aleatório apresentou um dos melhores tempos médios, com uma complexidade de movimentações, comparações e tempo de execução sempre $\mathcal{O}(n \log n)$, além de ser in loco, economizando memória e se destacando dentre os algoritmos apresentados. Assim, o Quick Sort é ideal para aplicações práticas onde o desempenho médio é mais relevante do que o pior caso.

4.6 ShellSort

Nos testes, o ShellSort se destacou por ter tempos de execução significativamente menores que os dos algoritmos quadráticos (Insertion, Selection e Bubble Sort) em dados desordenados. Além disso, em vetores crescentes e decrescentes, apresentou um tempo menor do que os algoritmos de complexidade $\mathcal{O}(n \log n)$ e $\mathcal{O}(n)$ e mesmo nos vetores aleatórios, ainda apresentou o melhor tempo para vetores de tamanho 1 e 1000. Sua eficiência se deve à redução do número de comparações em sublistas menores antes da ordenação final. Embora a complexidade dependa do intervalo usado, tornando-a difícil de prever com precisão, o ShellSort se mostrou uma ótima escolha para a ordenação de vetores, porém perdendo um pouco de eficiência com o aumento do tamanho do vetor e com vetores ordenados aleatoriamente. Além disso, o algoritmo é in loco, sendo ideal para casos onde o uso de memória é limitado, mas vale ressaltar que o algoritmo não é estável. Dessa forma, para vetores não tão grandes e de preferência ordenados decrescentemente, o ShellSort se mostra a opção ideal, com tempos surpreendentes.

4.7 HeapSort

Nos testes, o HeapSort também se destacou pela sua complexidade $\mathcal{O}(n \log n)$, mas difere do Merge Sort ao ser implementado in loco, sem a necessidade de memória extra significativa. Nos testes, ele apresentou tempos de execução competitivos e é particularmente útil quando se deseja um algoritmo eficiente e que não exija memória adicional. Contudo, ao comparar com os algoritmos que não são $\mathcal{O}(n^2)$, ele apresentou o maior número de comparacções, movimentações e tempo de execução nos vetores de tamanho 10000, mostrando uma clara queda de eficiência ao aumentarmos o tamanho do vetor, com exceção dos vetores ordenados aleatoriamente, onde conseguiu um tempo melhor que o MergeSort e ShellSort. Além disso, sua falta de estabilidade pode ser uma desvantagem em aplicações específicas. Dessa forma, dentre os algoritmos apresentados, o HeapSort não possui vantagem em muitos testes e perde sua força com o aumento do vetor, mas tem a vantagem de ser in loco.

4.8 RadixSort com CountingSort

Por fim, o Radix Sort com Counting Sort foi único em sua abordagem não comparativa, utilizando os dígitos individuais dos números para ordenar os elementos, mas vale lembrar que suas comparações não ficaram zeradas, pois para o CountingSort é necessário achar o maior elemento do vetor, sendo necessário comparações para isso. Contudo, o algoritmo se apresentou extremamente eficiente em diversos casos, superando os algoritmos baseados em comparações em algumas situações, sendo o segundo melhor tempo

em vetores aleatórios e crescentes, desconsiderando os algoritmos quadráticos. Contudo, sua aplicabilidade é limitada a tipos de dados que podem ser representados como valores discretos, como inteiros. Além de que não pode haver números negativos, e é necessário achar o maior elemento antes de sua aplicação, custando tempo, comparações e movimentações a mais. Além disso, a necessidade de espaço adicional para contagens pode ser uma restrição em sistemas com pouca memória. Também é importante citar que quando alteramos o campoDaEstrutura, o RadixSort apresentou uma grande mudança de tempo, se destacando negativamente nesse quesito. Mas em geral, o RadixSort com CountingSort apresentou boas médias de tempo, com poucas movimentações e comparações, em algoritmos que temos números inteiros, sabemos o maior elemento e só possuímos números negativos, o RadixSort com CountingSort certamente seria uma das melhores opções a serem utilizadas, sua estabilidade também é um destaque positivo para esse algoritmo.

4.9 Conclusão

Os resultados mostram que a escolha do algoritmo de ordenação ideal depende das características do conjunto de dados e dos requisitos do sistema. Algoritmos quadráticos como InsertionSort, SelectionSort e BubbleSort Melhorado são simples e podem ser eficientes em listas pequenas ou parcialmente ordenadas, mas tornam-se inadequados para grandes conjuntos de dados. Entre os algoritmos mais eficientes, o ShellSort apresentou bom desempenho geral, especialmente em listas de tamanhos moderados, enquanto o MergeSort destacou-se pela robustez e estabilidade, apesar do consumo extra de memória. O HeapSort mostrou-se competitivo em alguns casos, mas com desempenho inferior para vetores grandes e aleatórios. O QuickSort Aleatório apresentou o melhor desempenho médio em diversos cenários, mas sua eficiência foi impactada pelo tamanho do campoDaEstrutura. Por fim, o RadixSort com CountingSort, com sua abordagem não comparativa, mostrou-se extremamente rápido em cenários específicos, mas é limitado a dados discretos e requer memória adicional. Assim, a escolha do algoritmo de ordenação deve considerar o tamanho, a disposição dos dados e as restrições de memória do sistema.