

INSTITUTO FEDERAL DE MATO GROSSO DO SUL

TSI

Gabriel Ferreira Albuquerque Rangel

George Lucas Goulart De Oliveira

Josef Ferreira Melcher

**ANÁLISE DE DESEMPENHO E VISUALIZAÇÃO COMPARATIVA DE
ALGORITMOS DE ORDENAÇÃO**

AQUIDAUANA

2025

Gabriel Ferreira Albuquerque Rangel

George Lucas Goulart De Oliveira

Josef Ferreira Melcher

**ANÁLISE DE DESEMPENHO E VISUALIZAÇÃO COMPARATIVA DE
ALGORITMOS DE ORDENAÇÃO**

Trabalho apresentado a Instituto Federal de
Mato Grosso do Sul como primeira nota do
semestre de Tecnologia em Sistemas para
Internet

Orientador: Prof. Leandro Magalhães de
Oliveira

AQUIDAUANA

2025

RESUMO

O presente trabalho realiza uma análise teórica e empírica comparativa de seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort (com busca binária), Merge Sort (in-place), Quick Sort (pivô aleatório) e Heap Sort. O objetivo é conectar a teoria (complexidade assintótica, estabilidade, uso de memória) com resultados práticos obtidos por uma ferramenta desenvolvida em JavaScript/Node.js que mede tempo (ms), número de comparações e número de trocas/movimentações. Apresentam-se definições, metodologia experimental, tabelas de resultados e discussão sobre limitações e implicações práticas.

Palavras-chave: algoritmos de ordenação; análise empírica; complexidade; medição; estruturas de dados.

SUMÁRIO

1 INTRODUÇÃO.....	5
2 FUNDAMENTAÇÃO TEÓRICA.....	5
2.1 Bubble Sort.....	5
2.1.2 Uso da Flag - Bubble Sort Otimizado.....	6
2.1.3 Complexidade de Bubble Sort.....	6
2.2 Selection Sort.....	6
2.2.1 Double Selection Sort - Selection Sort Otimizado.....	7
2.2.2 Complexidade em Selection Sort.....	7
2.3 Insertion Sort.....	8
2.3.1 Binary Insertion Sort - Insertion Sort Otimizado.....	8
2.3.2 Complexidade em Insertion Sort.....	9
2.4 Merge Sort.....	9
2.4.1 Merge Sort In-Place.....	10
2.4.2 Complexidade do Merge Sort.....	10
2.5 Quick Sort.....	11
2.5.1 Complexidade do Quick Sort.....	11
2.6 Heap Sort.....	12
2.6.1 Complexidade do Heap Sort.....	12
3 ANÁLISE COMPARATIVA DE COMPLEXIDADE.....	13
3.1 Comparação de Complexidade de Tempo.....	13
3.1.1 Algoritmos Quadráticos.....	13
3.1.2 Algoritmos Log-Lineares.....	13
4. METODOLOGIA E PROTOCOLO EXPERIMENTAL.....	14
5. RESULTADOS EXPERIMENTAIS.....	14
5.1 Vetor Aleatório.....	14
5.2 Vetor Ordenado.....	15
5.3 Vetor Inversamente Ordenado.....	16
5.4 Vetor Quase Ordenado.....	17
6. DISCUSSÃO.....	18
7. CONCLUSÃO.....	18

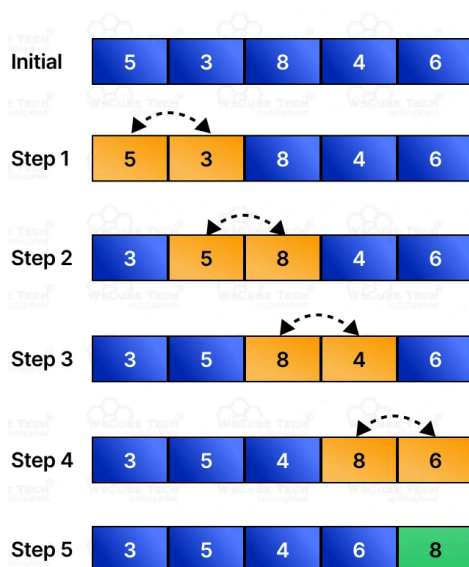
1 INTRODUÇÃO

A análise de algoritmos é fundamental para a Ciência da Computação. Entre os algoritmos mais estudados, os de ordenação desempenham papel central por serem componentes de inúmeras aplicações. Este trabalho compara seis algoritmos clássicos, relacionando teoria e prática por meio de experimentos executados com o software desenvolvido pela equipe. O objetivo é demonstrar as diferenças de desempenho em cenários diversos: vetores aleatórios, ordenados, inversamente ordenados e quase ordenados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Bubble Sort

O Bubble Sort é um dos algoritmos de ordenação mais simples e é frequentemente utilizado para fins didáticos. Ele compara duas bolhas vizinhas. Se a da esquerda for maior que a da direita, ele troca de lugar. Vai fazendo isso da esquerda até a direita, empurrando a “maior bolha” para o final da fila. Depois, repete o processo várias vezes, até que todas as bolhas estejam ordenadas.



2.1.2 Uso da Flag - Bubble Sort Otimizado

```
for (let n = 0; n < array.length; n++) {  
  let troca = false  
  for (let i = 0; i < array.length - 1; i++) {  
    if (array[i] > array[i + 1]) {  
      let aux = array[i]  
      array[i] = array[i + 1]  
      array[i + 1] = aux  
      troca = true  
    }  
  }  
  if(!troca) break  
}
```

Nesse algoritmo, também é possível aplicar algumas melhorias. Uma delas é o uso de uma flag, que é uma variável que funciona como um interruptor, servindo para indicar um estado, uma condição ou para sinalizar eventos.

Nesse caso, conforme mostrado na imagem, ela é utilizada para detectar se houve alguma troca durante uma passagem do laço e, caso não tenha ocorrido, o algoritmo é encerrado mais cedo, evitando continuar comparando elementos que já estão ordenados.

2.1.3 Complexidade de Bubble Sort

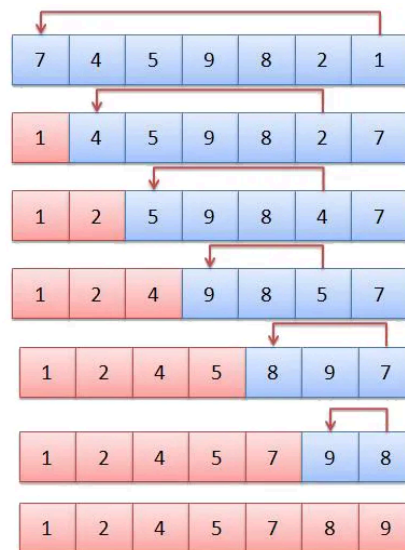
No **melhor caso**, quando o vetor já está ordenado, o Bubble Sort pode ter complexidade $O(n)$, desde que use a flag de parada para evitar passadas desnecessárias.

No **caso médio** e no **pior caso**, a complexidade é $O(n^2)$, pois o algoritmo precisa comparar e trocar muitos elementos. O caso médio ocorre quando o vetor está em ordem aleatória. O pior caso ocorre quando o vetor está em ordem inversa, exigindo a troca de cada elemento até a chegar na posição correta, resultando no maior número possível de trocas e comparações.

2.2 Selection Sort

A ordenação por seleção é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição, depois o de segundo

menor valor para a segunda posição, e assim é feito sucessivamente com os $n - 1$ elementos restantes, até os últimos dois elementos.



2.2.1 Double Selection Sort - Selection Sort Otimizado

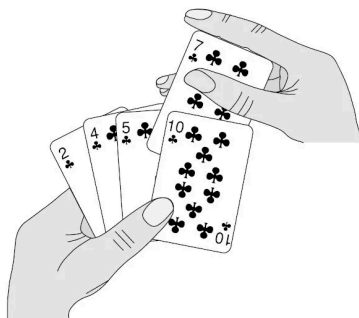
Para implementar uma melhoria nesse algoritmo, podemos utilizar o **Double Selection Sort**, essa variação otimizada faz com que em cada passagem pelo vetor, você procura tanto o menor quanto o maior elemento. Assim, em cada iteração, dois elementos vão para o lugar correto (um no início, outro no fim), e o algoritmo precisa de cerca da metade das passagens do Selection Sort comum.

2.2.2 Complexidade em Selection Sort

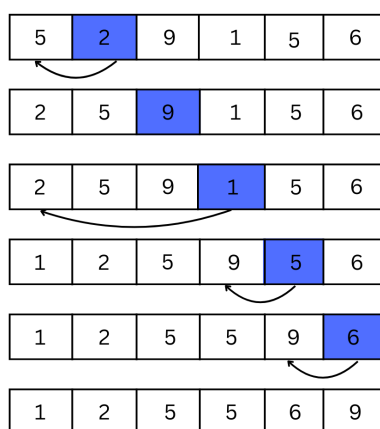
Em questão de complexidade do Selection Sort, todos os casos levam o $O(n^2)$. Independente da disposição dos dados, pois sempre percorre todo o vetor para encontrar o menor elemento em cada iteração. Contudo, por mais que a sua otimização possa reduzir o número de trocas quase pela metade, a complexidade continua sendo $O(n^2)$, porque ainda é preciso comparar todos os elementos do subarray não ordenado.

2.3 Insertion Sort

O algoritmo Insertion Sort tem como objetivo ordenar um conjunto de números de forma simples e intuitiva, funcionando de maneira semelhante a como organizamos cartas em uma mão de baralho.



Ele percorre o vetor elemento por elemento, e, para cada novo item, o insere na posição correta entre os anteriores, garantindo que a parte à esquerda esteja sempre ordenada.



Embora o Insertion Sort não seja o mais eficiente para grandes conjuntos de dados, ele é muito utilizado em contextos didáticos e também em situações onde o vetor está quase ordenado, pois nesse caso ele tem desempenho muito bom.

2.3.1 Binary Insertion Sort - Insertion Sort Otimizado

No Insertion Sort normal, para colocar um elemento no lugar certo, você vai comparando um a um com os elementos anteriores. Com inserção binária, o

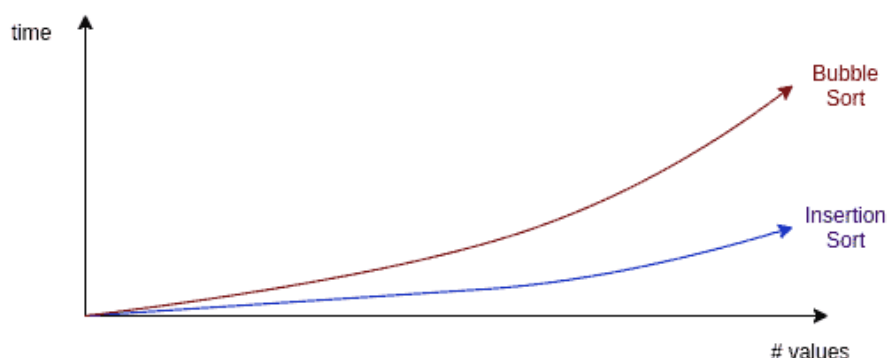
algoritmo usa busca binária no trecho já ordenado, ou seja, ele olha o elemento do meio e decide se o novo valor deve ficar à esquerda ou à direita, repetindo esse processo até encontrar o ponto exato de inserção.

Isso reduz o número de comparações de $O(n)$ para $O(\log n)$ por inserção. Quanto maior o vetor, mais vantagem o Binary tem, porque corta pela metade cada vez. O movimento de troca dos elementos ainda é necessário, então o tempo total não muda tanto, mas as comparações diminuem.

2.3.2 Complexidade em Insertion Sort

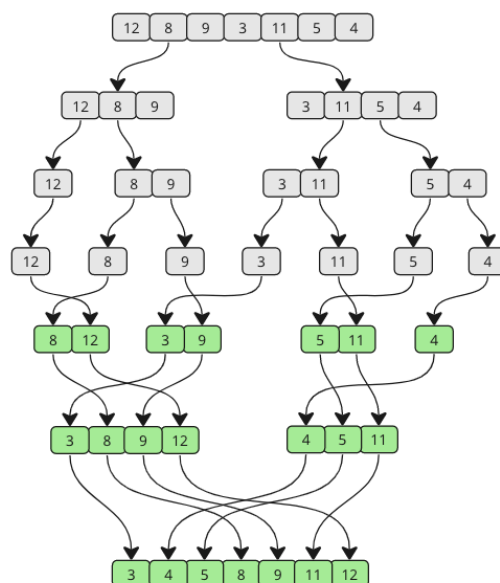
Sua complexidade no melhor caso é $O(n)$ (quando o vetor já está ordenado), enquanto o pior e médio caso são $O(n^2)$ (quando os elementos estão em ordem inversa ou aleatória).

Apesar disso, ele possui uma vantagem sobre o Bubble Sort, pois realiza menos trocas, tornando-o mais rápido em muitos casos.



2.4 Merge Sort

O Merge Sort é um algoritmo de ordenação eficiente que utiliza o conceito de dividir para conquistar. Ele funciona dividindo repetidamente o vetor em duas metades até que cada subvetor tenha apenas um elemento (ou esteja vazio). Em seguida, ele combina essas sublistas de forma ordenada, reconstruindo o vetor original já ordenado.



Esse algoritmo é muito utilizado quando é necessário ordenar grandes conjuntos de dados, pois possui complexidade garantida de forma consistente, diferentemente do Insertion Sort, que pode ser ineficiente em grandes vetores.

2.4.1 Merge Sort In-Place

Uma forma de melhoria para o Merge Sort é o chamado Merge Sort In-Place, que busca reduzir o consumo de memória extra do algoritmo tradicional. No Merge Sort convencional, a cada divisão do vetor, são criados novos arrays temporários para armazenar as metades e realizar a mesclagem. Isso aumenta o uso de memória, principalmente em grandes conjuntos de dados.

O Merge Sort In-Place realiza a mesclagem diretamente no vetor original, evitando a criação de arrays auxiliares e, assim, diminuindo o consumo de memória. Apesar disso, ele é mais complexo de implementar e pode ter um custo adicional de tempo devido aos deslocamentos dentro do mesmo vetor.

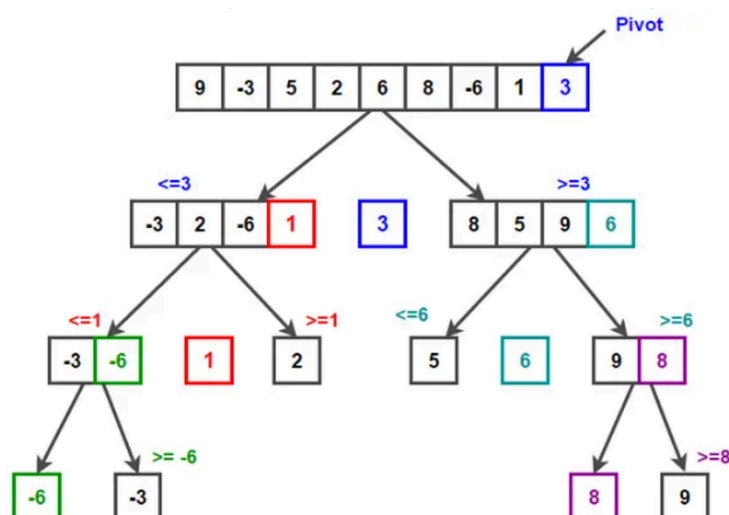
2.4.2 Complexidade do Merge Sort

O Merge Sort é muito eficiente, mesmo em grandes conjuntos de dados, graças à sua abordagem de dividir para conquistar. Melhor, médio e pior caso: O tempo de execução é sempre $O(n \log n)$, independentemente da ordem inicial dos elementos.

Possui a vantagem de ser estável (mantém a ordem relativa de elementos iguais) e realiza menos comparações do que algoritmos simples como Bubble ou Insertion Sort. A desvantagem é que ele requer memória extra proporcional ao tamanho do vetor, devido à criação de subvetores durante a mesclagem.

2.5 Quick Sort

O Quick Sort é um algoritmo de ordenação eficiente que utiliza a estratégia "dividir para conquistar". Ele seleciona um elemento como pivô e particiona o array de forma que os elementos menores que o pivô fiquem à esquerda e os maiores à direita. Em seguida, aplica recursivamente o mesmo processo para as partições esquerda e direita.



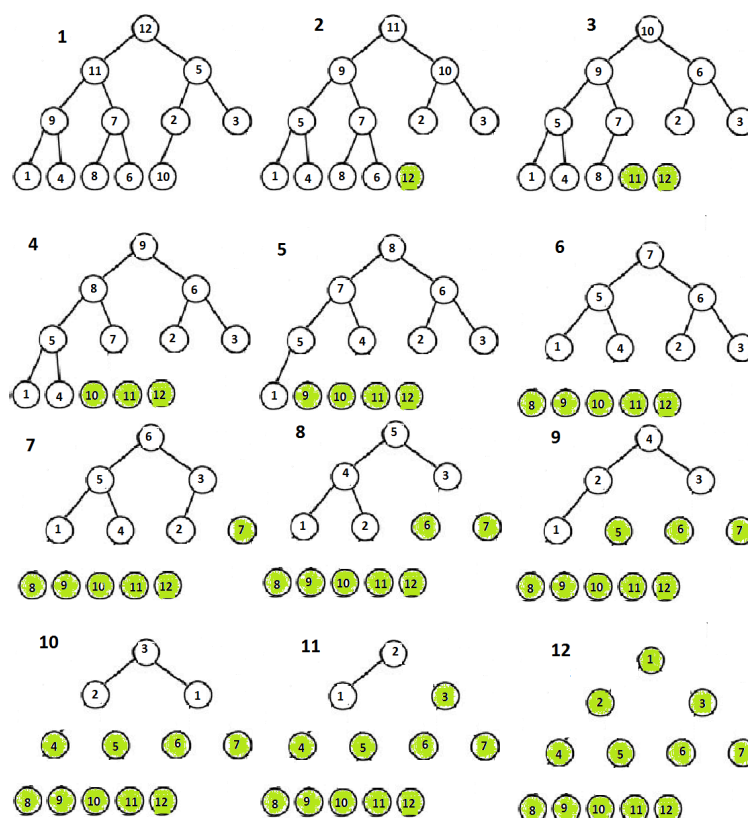
2.5.1 Complexidade do Quick Sort

No caso médio, o Quick Sort apresenta complexidade $O(n \log n)$, pois divide o vetor em partes menores e as ordena de forma eficiente.

No pior caso, a complexidade pode chegar a $O(n^2)$, quando as divisões ficam muito desbalanceadas. No código da equipe, foi usado um pivô aleatório, o que reduz bastante a chance desse pior caso acontecer.

2.6 Heap Sort

O Heap Sort é um algoritmo de ordenação baseado na estrutura de dados Heap (binária). Ele constrói um max-heap a partir do array e extrai repetidamente o maior elemento, colocando-o no final do array. O processo resulta em um array ordenado.



O algoritmo é particularmente útil quando se necessita de garantia de tempo de execução, pois evita o pior caso do Quick Sort.

2.6.1 Complexidade do Heap Sort

O algoritmo possui complexidade $O(n \log n)$ e uso de memória constante. Seu desempenho é estável em termos assintóticos, ou seja, não depende da ordem inicial dos dados, e não necessita de memória extra para execução.

3 ANÁLISE COMPARATIVA DE COMPLEXIDADE

3.1 Comparação de Complexidade de Tempo

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

3.1.1 Algoritmos Quadráticos

Bubble Sort, Selection Sort e Insertion Sort apresentam um tempo de execução que cresce exponencialmente com o aumento do tamanho do vetor. Isso os torna ineficientes para grandes conjuntos de dados. No entanto, Bubble Sort e Insertion Sort mostram uma vantagem significativa no melhor caso, que ocorre quando o vetor já está ordenado.

O Selection Sort, por sua vez, é o menos adaptativo, mantendo a complexidade independentemente da ordenação inicial dos dados, pois sempre precisa percorrer o restante do vetor para encontrar o menor elemento.

3.1.2 Algoritmos Log-Lineares

Merge Sort, Quick Sort e Heap Sort são consideravelmente mais eficientes para grandes volumes de dados. Sua abordagem de "dividir para conquistar" permite um desempenho escalável. O Merge Sort e o Heap Sort se destacam por sua consistência, mantendo a complexidade em todos os cenários. O Quick Sort, embora geralmente o mais rápido na prática no caso médio, possui um ponto fraco: seu pior caso, que o degrada para um desempenho quadrático ($O(n^2)$), similar aos algoritmos mais simples.

4. METODOLOGIA E PROTOCOLO EXPERIMENTAL

O software foi desenvolvido em **JavaScript (Node.js)**, utilizando a biblioteca **prompt-sync** para interação pelo terminal. Foram implementados os algoritmos de ordenação conforme o código disponibilizado pela equipe.

O tempo de execução é medido com `performance.now()` (módulo `perf_hooks`). Métricas coletadas: tempo (ms), número de comparações entre elementos e número de trocas (ou movimentações, no caso do merge in-place).

Configurações Utilizada nos testes:

- **Máquina:** AMD Ryzen 5 5600G with Radeon Graphics (3.90 GHz) e 32gb RAM
- **Versão do Node.js:** v22.19.0
- **Repetições:** média de 5 execuções por configuração
- **Tamanhos testados:** $n = 1.000, 10.000, 50.000$ e 100.000 .

As comparações representam cada verificação lógica entre dois elementos (por exemplo, `if (a > b)`).

As trocas indicam operações em que dois elementos são efetivamente permutados, enquanto as movimentações referem-se às atribuições sucessivas usadas durante o processo de mesclagem.

5. RESULTADOS EXPERIMENTAIS

Para a análise de desempenho dos algoritmos de ordenação, o código foi executado 5 vezes para cada configuração de tamanho de vetor ($n = 1.000, 10.000, 50.000$ e 100.000 elementos). Os resultados foram obtidos considerando a média dos tempos de execução, garantindo maior precisão e reduzindo o impacto de variações pontuais.

A seguir são apresentados os resultados e gráficos referentes a cada tipo de vetor testado.

5.1 Vetor Aleatório

$n = 1.000$

Análise de Desempenho:					
(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'2.56'	'1.96'	970029	257646
1	'Selection Sort'	'1.48'	'2.09'	250000	1231
2	'Insertion Sort'	'0.82'	'0.90'	9566	257763
3	'Merge Sort'	'0.85'	'0.58'	9337	257646
4	'Quick Sort'	'1.08'	'0.69'	11306	7645
5	'Heap Sort'	'0.66'	'0.61'	16825	9058

n= 10.000

Análise de Desempenho:						
(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)	Caso
0	'Bubble Sort'	'521.57'	'28.25'	99440055	24880506	'Médio'
1	'Selection Sort'	'50.20'	'8.29'	25000000	12381	'Médio'
2	'Insertion Sort'	'33.87'	'2.33'	128967	24921721	'Médio'
3	'Merge Sort'	'34.06'	'2.64'	126550	24880506	'Indiferente'
4	'Quick Sort'	'3.43'	'3.13'	179728	131037	'Médio'
5	'Heap Sort'	'2.68'	'1.39'	235304	124191	'Indiferente'

n= 50.000

Análise de Desempenho:					
(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'16478.78'	'1656.11'	2478400431	625458866
1	'Selection Sort'	'849.26'	'85.55'	625000000	62099
2	'Insertion Sort'	'957.63'	'81.26'	761570	626657964
3	'Merge Sort'	'615.94'	'6.33'	747479	625458866
4	'Quick Sort'	'10.01'	'4.11'	1885578	1667307
5	'Heap Sort'	'13.64'	'1.74'	1409586	737196

n= 100.000

Análise de Desempenho:						
(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)	Caso
0	'Bubble Sort'	'80741.66'	'3266.49'	9959800401	2500259618	'Médio'
1	'Selection Sort'	'9782.22'	'2227.60'	2500000000	124066	'Médio'
2	'Insertion Sort'	'9326.00'	'2504.15'	1622860	2505163507	'Médio'
3	'Merge Sort'	'9309.81'	'2481.90'	1595126	2500259618	'Indiferente'
4	'Quick Sort'	'23.06'	'4.87'	6265659	5835750	'Médio'
5	'Heap Sort'	'21.72'	'4.40'	3018436	1573824	'Indiferente'

Nos vetores aleatórios, os algoritmos quadráticos (Bubble Sort, Selection Sort e Insertion Sort) apresentaram tempos muito altos, especialmente o Bubble Sort, que foi o mais lento. O Insertion Sort teve desempenho um pouco melhor em vetores menores, mas ainda inferior aos algoritmos mais eficientes.

Os algoritmos de complexidade $O(n \log n)$ — Merge Sort, Quick Sort e Heap Sort — tiveram ótimo desempenho, com tempos muito menores. O Heap Sort foi o mais rápido, seguido de perto pelo Quick Sort e pelo Merge Sort.

5.2 Vetor Ordenado

n = 1.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'0.06'	'0.03'	999	0
1	'Selection Sort'	'1.07'	'1.26'	250000	0
2	'Insertion Sort'	'0.27'	'0.25'	8977	0
3	'Merge Sort'	'0.25'	'0.19'	999	0
4	'Quick Sort'	'1.10'	'0.56'	11010	6497
5	'Heap Sort'	'0.63'	'0.63'	17583	9708

n = 10.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'0.25'	'0.07'	9999	0
1	'Selection Sort'	'37.61'	'6.43'	25000000	0
2	'Insertion Sort'	'1.10'	'0.66'	123617	0
3	'Merge Sort'	'0.66'	'0.38'	9999	0
4	'Quick Sort'	'1.86'	'2.14'	149396	91695
5	'Heap Sort'	'2.21'	'1.27'	244460	131956

n= 50.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'0.35'	'0.45'	49999	0
1	'Selection Sort'	'802.88'	'117.97'	625000000	0
2	'Insertion Sort'	'4.76'	'0.61'	734465	0
3	'Merge Sort'	'1.29'	'0.71'	49999	0
4	'Quick Sort'	'6.42'	'5.71'	942731	513140
5	'Heap Sort'	'8.25'	'1.80'	1455438	773304

n= 100.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'0.43'	'0.52'	99999	0
1	'Selection Sort'	'11039.90'	'2929.65'	2500000000	0
2	'Insertion Sort'	'10.26'	'1.04'	1568929	0
3	'Merge Sort'	'2.34'	'0.96'	99999	0
4	'Quick Sort'	'10.53'	'5.13'	2007657	1153221
5	'Heap Sort'	'22.27'	'2.23'	3112517	1650854

Com os vetores já ordenados, o Bubble Sort e o Insertion Sort foram extremamente rápidos, pois praticamente não precisaram fazer trocas. O Merge Sort também manteve bom desempenho e o Heap Sort continuou eficiente, embora um pouco mais lento que o Quick. Enquanto o Selection Sort continuou lento mesmo com os dados organizados.

5.3 Vetor Inversamente Ordenado

n = 1.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'2.70'	'1.82'	999000	499500
1	'Selection Sort'	'1.20'	'1.56'	250000	500
2	'Insertion Sort'	'0.99'	'0.91'	9966	499500
3	'Merge Sort'	'1.08'	'0.93'	5931	499500
4	'Quick Sort'	'0.96'	'0.51'	10885	6412
5	'Heap Sort'	'0.62'	'0.63'	15965	8316

n = 10.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'585.82'	'65.80'	99990000	49995000
1	'Selection Sort'	'34.80'	'4.72'	25000000	5000
2	'Insertion Sort'	'83.93'	'19.27'	133602	49995000
3	'Merge Sort'	'50.01'	'2.36'	74607	49995000
4	'Quick Sort'	'1.85'	'2.26'	148881	90375
5	'Heap Sort'	'2.35'	'1.54'	226682	116696

n = 50.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'20200.10'	'1548.28'	2499950000	1249975000
1	'Selection Sort'	'897.21'	'115.44'	625000000	25000
2	'Insertion Sort'	'1880.79'	'141.07'	784448	1249975000
3	'Merge Sort'	'1246.90'	'15.42'	432511	1249975000
4	'Quick Sort'	'6.36'	'4.39'	941852	536458
5	'Heap Sort'	'12.25'	'1.66'	1366047	698892

n = 100.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'126656.49'	'7859.00'	9999900000	4999950000
1	'Selection Sort'	'11056.43'	'2847.51'	2500000000	50000
2	'Insertion Sort'	'27273.26'	'2982.19'	1668911	4999950000
3	'Merge Sort'	'27209.78'	'3379.75'	915023	4999950000
4	'Quick Sort'	'10.47'	'6.26'	2033457	1091036
5	'Heap Sort'	'17.37'	'4.54'	2926640	1497434

Nos vetores em ordem inversa, os algoritmos quadráticos sofreram grandes aumentos de tempo, principalmente o Bubble Sort, que teve o pior desempenho, o Insertion Sort também foi muito afetado.

Entre os algoritmos $O(n \log n)$, o Quick Sort manteve boa eficiência, com tempos próximos aos do Heap e do Merge Sort, todos bem mais rápidos que os quadráticos.

5.4 Vetor Quase Ordenado

n = 1.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'2.19'	'1.76'	893106	58834
1	'Selection Sort'	'1.40'	'1.94'	250000	104
2	'Insertion Sort'	'0.65'	'0.79'	9551	58834
3	'Merge Sort'	'0.54'	'0.76'	7843	58834
4	'Quick Sort'	'0.98'	'0.43'	10639	6566
5	'Heap Sort'	'0.72'	'0.72'	17417	9596

n = 10.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'219.23'	'8.54'	95690430	5675858
1	'Selection Sort'	'36.83'	'7.14'	25000000	1086
2	'Insertion Sort'	'9.85'	'1.27'	129018	5675858
3	'Merge Sort'	'6.80'	'1.24'	112328	5675858
4	'Quick Sort'	'1.75'	'1.74'	154410	89064
5	'Heap Sort'	'2.26'	'1.31'	242581	130094

n = 50.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'12188.54'	'2379.11'	2483100337	143724464
1	'Selection Sort'	'861.78'	'129.40'	625000000	5346
2	'Insertion Sort'	'217.12'	'22.49'	761270	143724464
3	'Merge Sort'	'145.59'	'1.89'	679253	143724464
4	'Quick Sort'	'6.25'	'4.08'	950136	526143
5	'Heap Sort'	'10.95'	'2.70'	1447024	767738

n = 100.000

Análise de Desempenho:

(index)	Algoritmo	Tempo (ms) média	Tempo (ms) std	Comparações (média)	Trocas/Movim. (média)
0	'Bubble Sort'	'69561.31'	'2208.74'	9925700742	575490436
1	'Selection Sort'	'11107.07'	'2950.37'	2500000000	10708
2	'Insertion Sort'	'801.73'	'31.47'	1622235	575490436
3	'Merge Sort'	'575.39'	'7.89'	1456169	575490436
4	'Quick Sort'	'10.48'	'4.65'	2024972	1155838
5	'Heap Sort'	'18.47'	'3.49'	3095184	1636084

Nos vetores quase ordenados, o Insertion Sort teve o melhor desempenho comparado com os outros algoritmos quadráticos, já que precisou de poucas comparações. O Quick Sort e o Heap Sort também mantiveram tempos muito bons, com o Merge Sort logo atrás. O Bubble Sort e o Selection Sort, apesar de melhorarem um pouco, continuaram sendo os menos eficientes.

6. DISCUSSÃO

Os resultados obtidos confirmam o comportamento teórico esperado dos algoritmos de ordenação.

Os algoritmos quadráticos — Bubble Sort, Selection Sort e Insertion Sort — possuem complexidade $O(n^2)$, o que os torna ineficientes em vetores grandes e desordenados. Por isso, seus tempos crescem rapidamente conforme aumenta o valor de n .

Em contrapartida, os algoritmos Merge Sort, Quick Sort e Heap Sort, de complexidade $O(n \log n)$, apresentaram desempenho muito superior, sendo adequados para grandes volumes de dados.

O Heap Sort destacou-se no caso aleatório por manter um tempo consistente independentemente da distribuição dos valores, enquanto o Quick Sort foi o mais rápido em vetores ordenados devido à menor quantidade de comparações e trocas necessárias.

O Insertion Sort, embora quadrático, mostrou ótimo desempenho nos vetores quase ordenados, pois nesse tipo de dado ele realiza poucas movimentações, aproximando-se de um comportamento quase linear.

7. CONCLUSÃO

Conclui-se que o desempenho dos algoritmos varia conforme o tipo de vetor e o grau de ordenação inicial dos dados. Os algoritmos $O(n^2)$ são aceitáveis apenas para vetores pequenos ou quase ordenados, enquanto os algoritmos $O(n \log n)$ são os mais indicados para grandes quantidades de dados.

De modo geral, o Heap Sort e o Quick Sort apresentaram os melhores resultados, mostrando-se mais estáveis e eficientes nos diferentes cenários testados.