UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
CAMPUS NATAL CENTRAL

Tobias dos Santos Neto
Wisla Alves Argolo

ANÁLISE EMPÍRICA DOS ALGORITMOS DE ORDENAÇÃO

Natal - RN
2023

Tobias dos Santos Neto
Wisla Alves Argolo

ANÁLISE EMPÍRICA DOS ALGORITMOS DE ORDENAÇÃO

Relatório técnico apresentado como avaliação da disciplina Estruturas de Dados Básicas I ministrada pelo professor Dr. Selan Rodrigues dos Santos para o curso de Bacharelado em Tecnologia da Informação do Instituto Metropóle Digital da Universidade Federal do Rio Grande do Norte - Campus Natal Central.

Natal - RN
2023

Sumário

Lista de Figuras

Lista de Tabelas

1 INTRODUÇÃO

Um algoritmo consiste em um conjunto de instruções organizadas sistematicamente para resolver um problema computacional (SZWARCFITER; MARKENZON, 2015). Esses conjuntos de instruções organizadas desempenham um papel essencial em nossa vida cotidiana, muitas vezes sem que percebamos. A eficiência relativa ao tempo de processamento no manuseio de dados pode ser significativamente aumentada quando os dados são organizados seguindo um determinado critério de ordenação (DROZDEK, 2013). Nesse contexto, um exemplo notável é encontrado no funcionamento do Spotify e Netflix, que utilizam algoritmos de ordenação para recomendar músicas e filmes aos usuários com base em suas preferências. Esses algoritmos analisam uma vasta quantidade de dados e classificam o conteúdo de forma eficiente, proporcionando sugestões personalizadas.

Compreender a importância e a utilidade desses algoritmos é fundamental para aproveitar plenamente os benefícios que eles proporcionam. Essas sequências de passos lógicos nos auxiliam a resolver problemas de forma consistente e otimizada, contribuindo para uma experiência personalizada e agradável em nosso dia à dia.

Em casos que a análise matemática contribui pouco para entender o desempenho esperado de um determinado algoritmo, a análise empírica é a ferramenta responsável pela obtenção desse entendimento (SEDGEWICK, 1998). Considerando tal importância, esse relatório propõe a análise empírica em termos de tempo de execução dos 7 algorítmos de ordenação (*Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Shell Sort*, *Merge Sort*, *Quick Sort* e *Radix Sort*) mediante testes em diferentes condições, para entender melhor o comportamento desses algoritmos e determinar, dada uma situação específica, o algorítmo mais indicado em termos de eficiência.

2 PROCEDIMENTO EXPERIMENTAL

2.1 Materiais

Esta subseção apresenta e descreve os softwares e hardwares necessários para a realização deste trabalho.

2.1.1 Computador e Sistema Operacional

Este trabalho foi conduzido com as especificações do computador e sistema operacional presentes na Tabela 1 e 2, respectivamente.

Tabela 1: Especificações do Computador

Processador	**Memória**	**Armazenamento**
AMD Ryzen 7 5700X @ 3.4-4.6GHz	2x16GB DDR4 @ 3600MHz	SSD Netac 3000 500GB NVME

Fonte: Elaborado pelo autor (2023).

Tabela 2: Especificações do Sistema Operacional

Sistema Operacional	**Versão**	**Arquitetura**
Windows 11 Home	22H2	x64

Fonte: Elaborado pelo autor (2023).

Além disso, foi utilizado o terminal WSL (*Windows Subsystem for Linux*) executando a versão 20.04 da distribuição Linux Ubuntu.

2.1.2 Ferramentas de programação

Os 7 algoritmos de ordenação analisados foram implementados na linguagem C++17 a partir do editor de código-fonte gratuito e de código aberto VS Code (*Visual Studio Code*).

Além disso, tais códigos foram compilados no WSL utilizando a versão 3.22.1 do CMake (*Cross-Plataform Make*) a partir dos seguintes comandos:

```
cmake -S source -B build -D CMAKE_BUILD_TYPE=Release
cmake --build build # Aciona o processo de compilação dentro da pasta build
```

Em que *source* corresponde pasta na qual o cmake será encontrado.

E para realizar a execução:

```
cd build # Entra na pasta build
./sortsuite
```

As medições de tempo foram realizadas usando a biblioteca *chrono*, e por meio da biblioteca *fstream* os dados foram escritos em arquivos de extensão *.txt*.

2.1.3 Gráficos

Os gráficos obtidos neste trabalho foram gerados através do aplicativo de visualização Gnuplot versão 5.4 *patchlevel* 6 para Windows.

2.2 Algoritmos

Nesta subseção, os pseudocódigos dos algoritmos de ordenação serão apresentados.

2.2.1 Insertion Sort

Este algoritmo divide o arranjo em duas partes: uma parte já ordenada, começando com o primeiro elemento, e uma parte não ordenada, inicialmente vazia. Em cada iteração, um elemento da parte não ordenada é selecionado e inserido na posição correta na parte ordenada, deslocando os elementos maiores para a direita. Esse processo é repetido até que todos os elementos estejam na posição correta e o arranjo esteja totalmente ordenada.

Algoritmo 1: Insertion Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores
Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1 Função insertion(A)
2     para i ← 1 até n − 1 faça
3         chave ← A[i]
4         j ← i − 1
5         enquanto j ≥ 0 e A[j] > chave faça
6             A[j + 1] ← A[j]
7             j ← j − 1
8         A[j + 1] ← chave
```

2.2.2 Selection Sort

O *Selection Sort* divide o arranjo em duas partes: uma parte ordenada à esquerda e uma parte não ordenada à direita. O algoritmo percorre repetidamente a parte não ordenada em busca do menor elemento e o coloca na posição correta na parte ordenada.

Algoritmo 2: Selection Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1 Função selection(A)
2     para i ← 0 até n − 2 faça
3         menor ← i
4         para j ← i + 1 até n − 1 faça
5             se A[j] < A[menor] então
6                 menor ← j
7         A[i] ↔ A[menor]
```

2.2.3 Bubble Sort

Esta solução refere-se a versão clássica do *Bubble Sort* em que o algoritmo percorre repetidamente o arranjo a ser ordenado, comparando pares de elementos adjacentes e os trocando de posição se estiverem na ordem errada, de modo que o maior elemento do arranjo é gradualmente "empurrado"para o final a cada iteração até que o arranjo esteja completamente ordenado.

Algoritmo 3: Bubble Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1 Função bubble(A)
2     para i ← 0 até n − 1 faça
3         para j ← 0 até n − 2 faça
4             se A[j] > A[j + 1] então
5                 A[j] ↔ A[j + 1]
```

2.2.4 Shell Sort

O *Shell Sort* é um algoritmo de ordenação baseado no *Insertion Sort*. Contudo, em vez de comparar e mover elementos adjacentes, o *Shell Sort* compara elementos que estão separados por um intervalo maior (*gap*) e os ordena utilizando o *Insertion Sort*. O intervalo é reduzido gradualmente até que seja igual a 1, quando o algotitmo é finalizado com uma última passagem do *Insertion Sort* no arranjo completo. Essa abordagem permite mover elementos distantes mais rapidamente para suas posições corretas, resultando em um menor número de comparações e trocas.

Neste projeto implementamos o *Shell Sort* original, no qual a sequência de intervalos

é calculada através da fórmula

$$gap = \left\lfloor \frac{n}{2^k} \right\rfloor$$

Em que n é o tamanho do arranjo e k é o número de iterações.

Algoritmo 4: Shell Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1  Função shell(A)
2      gap ← n/2
3      enquanto gap > 0 faça
4          para i ← gap até n − 1 faça
5              temp ← A[i]
6              j ← i
7              enquanto j ≥ gap e A[j − gap] > temp faça
8                  A[j] ← A[j − gap]
9                  j ← j − gap
10             A[j] ← temp
11         gap ← gap/2
```

2.2.5 Merge Sort

A estratégia do *Merge Sort* é dividir o arranjo original em partes menores de tamanho aproximadamente igual, recursivamente ordenar cada parte separadamente e, em seguida, mesclar (*merge*) as partes ordenadas a partir da comparação dos elementos para obter o arranjo final ordenado. O algoritmo aproveita a facilidade de ordenar listas pequenas e, gradualmente, combina as partes em ordem crescente até obter o arranjo completamente ordenado.

Para isso, o *Merge Sort* utiliza uma função auxiliar, a *merge*, que realiza a mesclagem.

Algoritmo 5: Merge Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1  Função merge_sort(A):
2      n ← tam(A)
3      se n ≥ 2 então
4          meio ← n/2
5          listaEsquerda[meio]
6          listaDireita[n − meio]
7          para i ← 0 até meio − 1 faça
8              listaEsquerda[i] ← A[i]
9          para j ← meio até n − 1 faça
10             listaDireita[j − meio] ← A[j]
11         merge_sort(listaEsquerda)
12         merge_sort(listaDireita)
13         merge(listaEsquerda, listaDireita, A)
14 Função merge(listaEsquerda, listaDireita, A):
15     tam_e ← tam(listaEsquerda)
16     tam_d ← tam(listaDireita)
17     i ← j ← k ← 0
18     enquanto i < tam_e e j < tam_d faça
19         se listaEsquerda[i] < listaDireita[j] então
20             A[k] ← listaEsquerda[i]
21             i ← i + 1
22         senão
23             A[k] ← listaDireita[j]
24             j ← j + 1
25         k ← k + 1
26     enquanto i < tam_e faça
27         A[k] ← listaEsquerda[i]
28         i ← i + 1
29         k ← k + 1
30     enquanto j < tam_d faça
31         A[k] ← listaDireita[j]
32         j ← j + 1
33         k ← k + 1
```

2.2.6 Quick Sort

Este algoritmo escolhe um elemento chamado de "pivô"e particiona o arranjo ao redor desse pivô, de forma que os elementos menores que o pivô fiquem à sua esquerda e os elementos maiores fiquem à sua direita. Em seguida, o algoritmo é aplicado recursivamente aos dois subarranjos resultantes (à esquerda e à direita do pivô) até que o arranjo original esteja completamente ordenado.

A escolha do pivô pode afetar o desempenho do *Quick Sort*, pois pode resultar em subarranjos desbalanceadis. Neste projeto, empregamos a estratégia da mediana de três em que são selecionados três elementos do arranjo (o primeiro, o do meio e o último) e o pivô é escolhido como o valor mediano destes. Essa abordagem melhora o desempenho do algoritmo, evitando casos de tempo quadrático em dados ordenados de forma não-crescente e não-decrescente.

Algoritmo 6: Quick Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores e indíces para o $inicio$ e fim do intervalo considerado

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente no intervalo $[inicio,\ fim]$

```
1  Função quick(A, inicio, fim):
2      se inicio < fim então
3          pivoIndice ← partition(A, inicio, fim)
4          quick(A, inicio, pivoIndice − 1)
5          quick(A, pivoIndice + 1, fim)
6  Função partition(A, inicio, fim):
7      meio ← (inicio + fim)/2
8      se A[meio] < A[inicio] então
9          A[meio] ↔ A[inicio]
10     se A[fim] < A[inicio] então
11         A[fim] ↔ A[inicio]
12     se A[fim] < A[meio] então
13         A[fim] ↔ A[meio]
14     A[fim] ↔ A[meio]
15     pivo ← A[fim]
16     pivoIndice ← inicio
17     para i ← inicio até fim − 1 faça
18         se A[i] < pivo então
19             A[i] ↔ A[pivoIndice]
20             pivoIndice ← pivoIndice + 1
21     A[pivoIndice] ↔ A[fim]
22     retorna pivoIndice
```

2.2.7 Radix Sort

Diferentemente dos algoritmos de ordenação apresentados anteriormente que comparam os elementos diretamente, a estratégia do *Radix Sort* é baseada na análise dos dígitos individuais dos números de um arranjo qualquer. Ele começa pelo dígito menos significativo (o dígito mais à direita) e vai até o dígito mais significativo (o mais à esquerda).

Neste trabalho, utilizamos o *Radix Sort* baseado no dígito menos significativo versão *buckets* em que os elementos do arranjo são divididos em diferentes "baldes"(*buckets*) de

acordo com o valor do dígito atual analisado. Posteriormente, os elementos são coletados de volta da ordem dos baldes e colocados novamente no arranjo original. Esse processo é repetido para cada dígito, do menos significativo ao mais significativo, até que todos os dígitos tenham sido considerados.

Algoritmo 7: Radix Sort

Entrada: Um arranjo qualquer $A[0 \ldots n-1]$ de n valores

Saída: O arranjo $A[0 \ldots n-1]$ ordenado de forma não-decrescente

```
1  Função radix(A)
2      max ← maxElemento(A)
3      max_digitos ← numDigitos(max)
4      para i ← 0 até max_digitos faça
5          baldes[10][ ]
6          para j ← 0 até n − 1 faça
7              indice ← (int)(A[j]/10^i)%10
8              adicionarElementoBalde(A[j], buckets[indice])
9              destino ← 0
10             para cada balde em baldes faça
11                 para cada elemento em balde faça
12                     A[destino] ← elemento
13                     destino ← destino + 1
14 Função maxElemento(A)
15     max ← A[0] para i ← 1 até n − 1 faça
16         se A[i] > max então
17             max ← A[i]
18     retorna max
19 Função numDigitos(max)
20     contador ← 0
21     enquanto max > 0 faça
22         max/10
23         contador ← contador + 1
24     retorna contador
25 Função adicionarElementoBalde(elemento, balde)
26     tam_balde ← tam(balde)
27     balde[tam_balde] ← elemento
```

2.3 Obtenção dos dados

Neste trabalho, realizamos uma simulação de 7 algoritmos de ordenação sobre um arranjo de inteiros. Tais algoritmos foram analisados empiricamente quanto ao tempo de execução.

Para tanto, foram considerados os seguintes cenários: i) arranjos com elementos em ordem não-decrescente, ii) arranjos com elementos em ordem não-crescente, iii) arranjos com elementos 100% aleatórios, iv) arranjos com 75% de seus elementos em sua posição definitiva, v) arranjos com 50% de seus elementos em sua posição definitiva, e vi) arranjos com 25% de seus elementos em sua posição definitiva.

Adicionalmente, em cada cenário descrito, os algoritmos foram testados para 25 tamanhos distintos de entrada n_i, variando de 10^2 a 10^5, com incremento progressivo em etapas de 4162 elementos. Para aumentar a precisão das medidas e evitar flutuações, realizamos 5 execuções para cada instância de tamanho n_i.

Em seguida, para obter o o tempo médio de execução do algoritmo para uma instância do problema com tamanho n_i, foi calculada a média aritmética dessas execuções. Para isso, empregamos a média progressiva utilizando a equação a seguir:

$$M_0 = 0, \qquad \text{(valor inicial da média)}$$

$$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k}, \qquad \text{(atualização progressiva da média)}$$

Essa fórmula foi aplicada para $k = 1, 2, \ldots 5$ execuções, em que x_k representa o tempo registrado para a k-ésima execução e M_5 é equivalente à média aritmética final da sequência de 5 tempos medidos.

2.4 Tratamento dos dados

Os dados recolhidos permitiram a criação de uma tabela em formato *.txt* para cada cenário, correlacionando o tamanho da entrada com o tempo de execução de cada algoritmo. Em seguida, utilizando estas tabelas e visando uma representação gráfica clara das diferenças nos tempos de execução entre algoritmos, foram gerados 3 tipos de gráficos: i) gráficos, em escala logarítmica no eixo y, que analisam a velocidade dos 7 algoritmos de ordenação, ii) gráficos, em escala logarítmica no eixo y, que analisam a velocidade dos algoritmos com complexidade do caso médio $O(n^2)$ (*Bubble Sort*, *Insertion Sort* e *Selection Sort*) juntamente com o *Radix Sort*, e ii) gráficos, em escala logarítmica no eixo y, que analisam a velocidade dos algoritmos com a complexidade do caso médio $O(n \log n)$ (*Quick Sort*, *Merge Sort*) juntamente com o *Radix Sort* e *Shell Sort*.

Por fim, foi criado um gráfico em escala logarítmica no eixo y para cada um dos 7 algoritmos de ordenação, representando o ajuste de curva entre a função correspondente

ao pior caso do algoritmo e os dados medidos do tempo de execução do mesmo. Esse ajuste, também conhecido como *fitting*, permite a comparação direta entre a função do pior caso teoricamente prevista com os tempos de execução realmente observados.

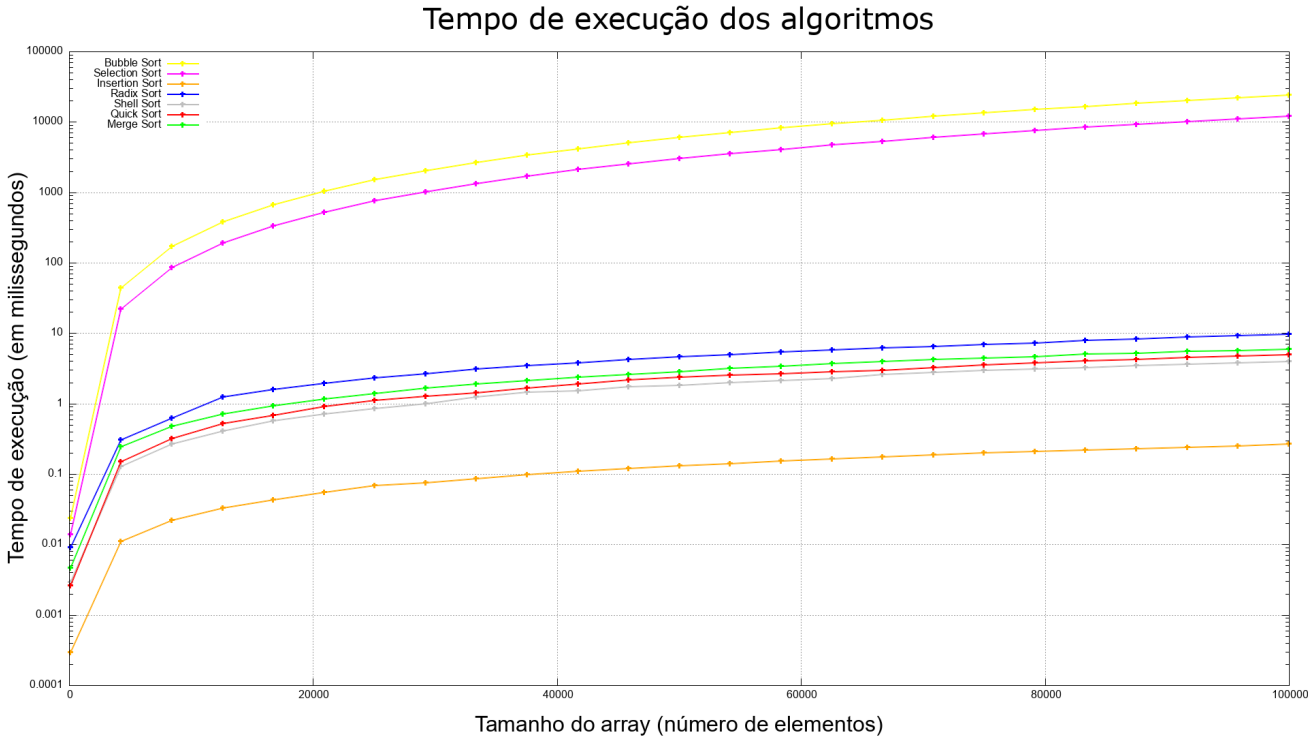
3 RESULTADOS

Nesta seção, são apresentados os gráficos e tabelas gerados a partir dos dados de medição de tempo para cada um dos cenários e grupos de algoritmos.

3.1 Cenário em que o arranjo está em ordem não-decrescente

3.1.1 Os 7 algoritmos

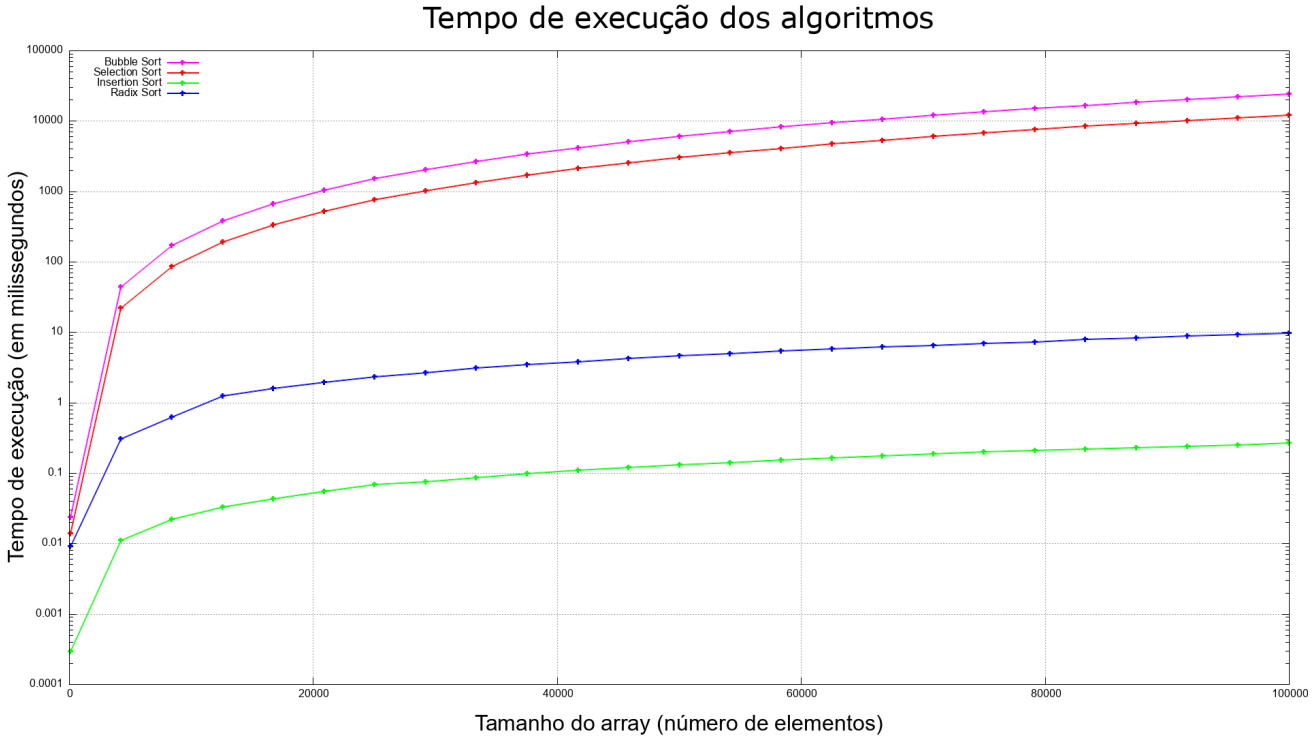
Figura 1: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está em ordem não-decrescente



Fonte: Elaborado pelo autor (2023).

3.1.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 2: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala linear no eixo y, quando o arranjo está em ordem não-decrescente



Fonte: Elaborado pelo autor (2023).

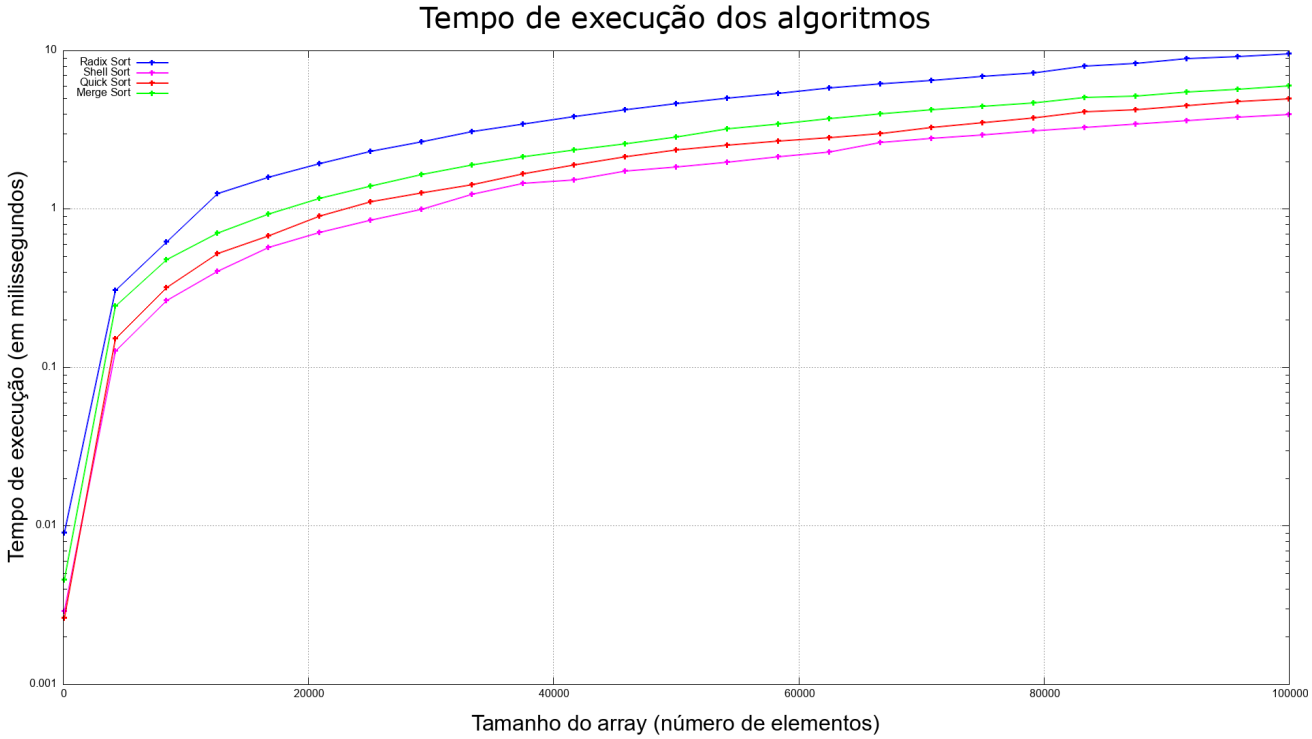
Tabela 3: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está em ordem não-decrescente

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.023634	0.013748	0.000290	0.008970
4262	43.396427	21.817689	0.011034	0.306812
8424	169.514234	85.090781	0.021796	0.620786
12586	377.425838	189.078579	0.032724	1.244912
16748	670.005493	334.547971	0.043285	1.588864
20910	1044.118152	523.196955	0.054613	1.930689
25072	1501.444479	755.313522	0.067959	2.315609
29234	2045.598026	1023.684132	0.075533	2.657084
33396	2670.336614	1336.302829	0.086297	3.063274
37558	3366.228809	1685.231278	0.097053	3.441991
41720	4174.666182	2100.984773	0.108760	3.817912
45882	5061.608286	2523.333304	0.119197	4.231907
50044	5995.499669	2995.971522	0.129418	4.625304
54206	7078.373744	3514.133815	0.140920	4.976962
58368	8158.666166	4085.463936	0.152534	5.356829
62530	9364.752992	4678.278329	0.161798	5.779317
66692	10640.344375	5317.471728	0.174844	6.128675
70854	12004.542651	6024.058499	0.185324	6.441836
75016	13467.193178	6740.604601	0.198039	6.835511
79178	14980.129987	7503.257125	0.207890	7.199069
83340	16607.603281	8423.820961	0.217061	7.963659
87502	18267.387288	9178.499788	0.228645	8.298057
91664	20090.376253	10049.983246	0.239427	8.907967
95826	21900.333452	10972.681861	0.250571	9.147672
99988	23845.592893	11953.282626	0.264831	9.548011

Fonte: Elaborado pelo autor (2023).

3.1.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 3: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala linear no eixo y, quando o arranjo está em ordem não-decrescente



Fonte: Elaborado pelo autor (2023).

Tabela 4: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está em ordem não-decrescente

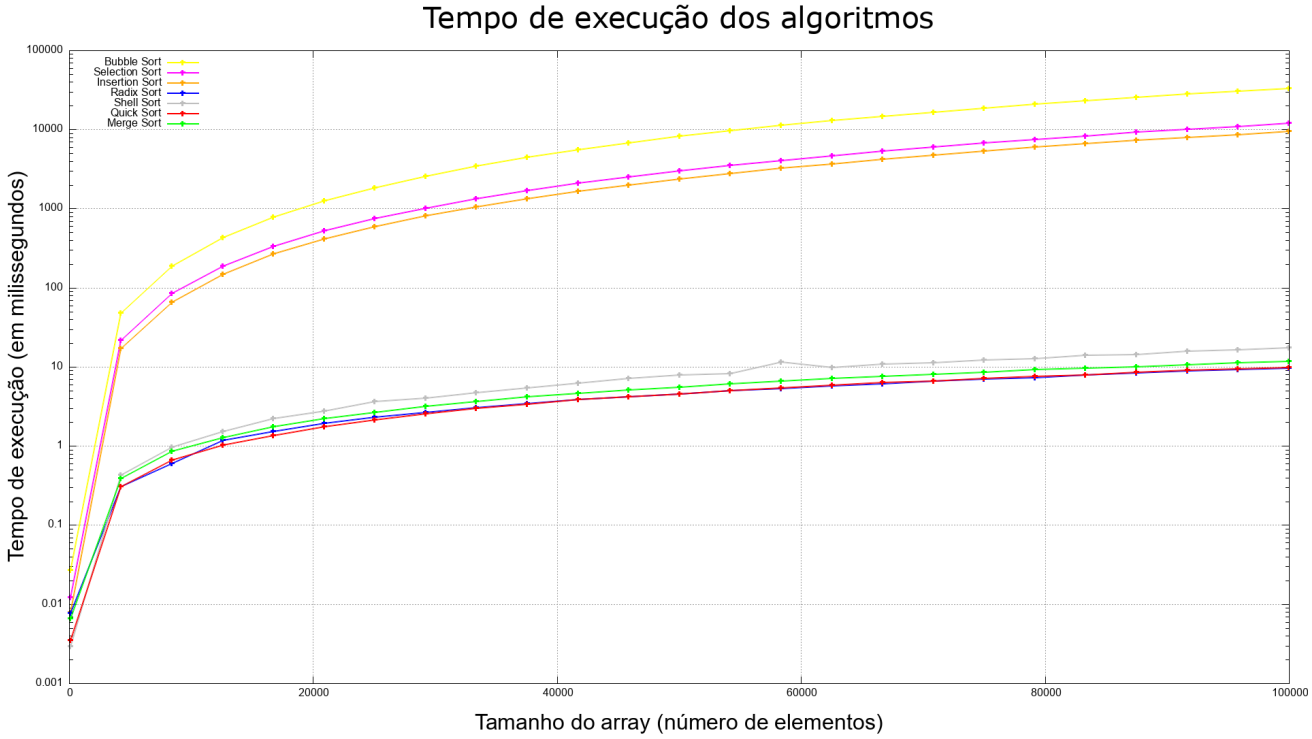
Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.002892	0.002602	0.004570	0.008970
4262	0.126950	0.150990	0.242841	0.306812
8424	0.263767	0.319342	0.478658	0.620786
12586	0.404849	0.524467	0.705377	1.244912
16748	0.569251	0.678791	0.931716	1.588864
20910	0.710539	0.899560	1.166467	1.930689
25072	0.853325	1.111084	1.391056	2.315609
29234	0.990473	1.260628	1.656058	2.657084
33396	1.234182	1.425254	1.891176	3.063274
37558	1.444095	1.669682	2.135717	3.441991
41720	1.527708	1.891758	2.348300	3.817912
45882	1.725336	2.141580	2.576447	4.231907
50044	1.830597	2.353407	2.856773	4.625304
54206	1.975040	2.514196	3.196443	4.976962
58368	2.126635	2.668731	3.414600	5.356829
62530	2.285542	2.826104	3.714408	5.779317
66692	2.614244	2.988243	3.961579	6.128675
70854	2.779090	3.259364	4.213154	6.441836
75016	2.934088	3.507160	4.451466	6.835511
79178	3.105352	3.756253	4.657514	7.199069
83340	3.263328	4.081513	5.046911	7.963659
87502	3.434786	4.244879	5.175748	8.298057
91664	3.595253	4.504592	5.475527	8.907967
95826	3.792513	4.756527	5.704584	9.147672
99988	3.934927	4.934402	5.963245	9.548011

Fonte: Elaborado pelo autor (2023).

3.2 Cenário em que o arranjo está 25% ordenado

3.2.1 Os 7 algoritmos

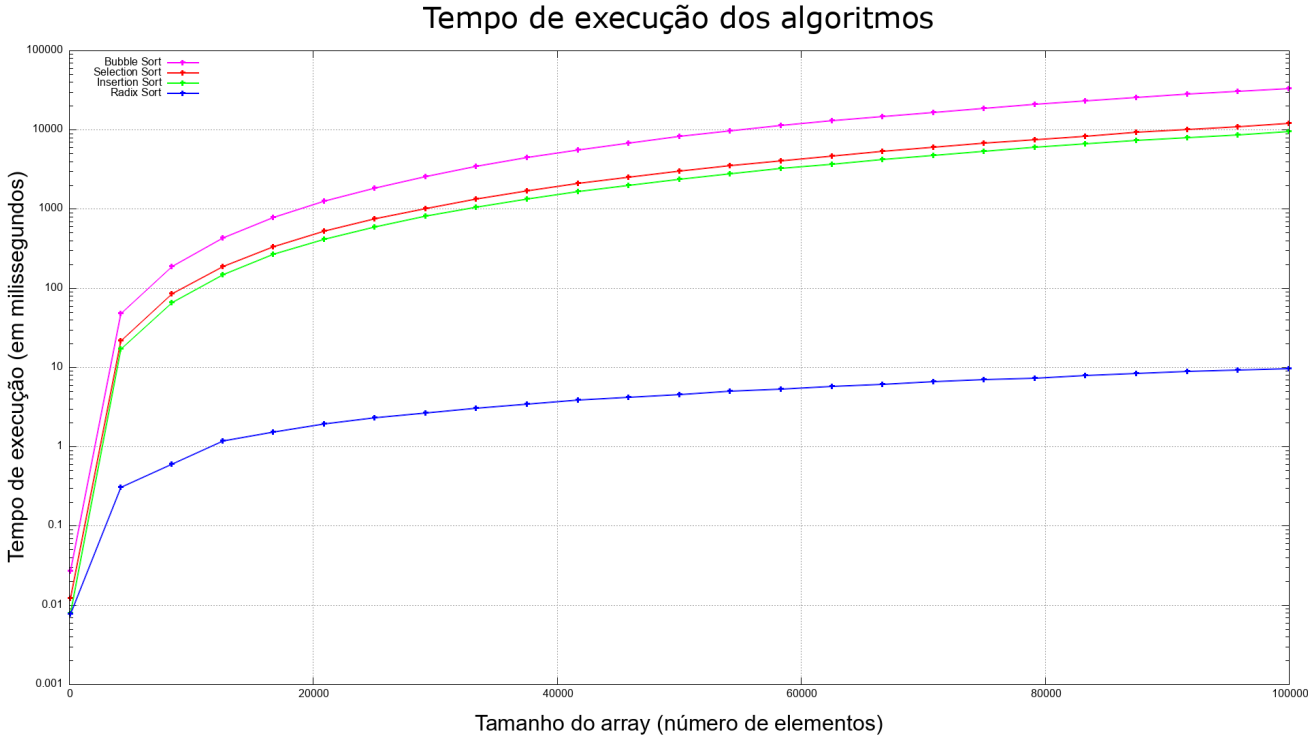
Figura 4: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está 25% ordenado



Fonte: Elaborado pelo autor (2023).

3.2.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 5: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 25% ordenado



Fonte: Elaborado pelo autor (2023).

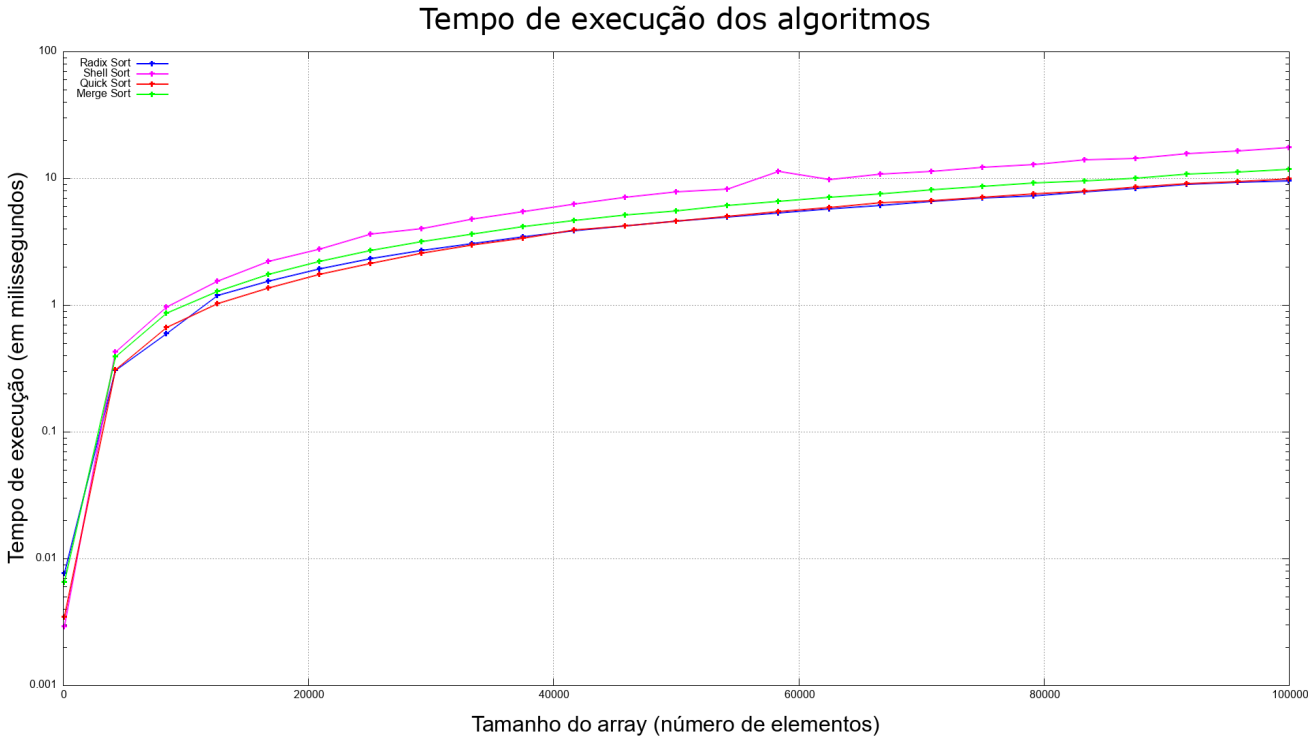
Tabela 5: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está 25% ordenado

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.026636	0.012052	0.007816	0.007632
4262	48.241740	21.598674	17.062072	0.304724
8424	189.079629	84.904863	66.069906	0.595602
12586	428.360432	188.863404	147.611921	1.184607
16748	785.304282	335.100844	270.048151	1.533562
20910	1263.749389	522.545551	411.517793	1.935435
25072	1839.646665	751.313581	590.464885	2.308978
29234	2577.797896	1019.059833	805.252197	2.676885
33396	3430.274673	1332.974317	1059.839079	3.058300
37558	4432.303849	1686.100779	1323.471734	3.438898
41720	5568.001099	2082.627796	1646.346768	3.843758
45882	6799.556794	2523.583037	1981.965236	4.199014
50044	8174.941707	2995.510430	2350.245188	4.571371
54206	9648.981260	3519.401283	2781.301176	4.965141
58368	11221.257512	4075.956806	3226.320383	5.325065
62530	12891.822986	4686.002664	3684.570469	5.720562
66692	14685.699860	5316.943086	4213.516712	6.089769
70854	16582.194590	6001.935559	4729.215274	6.561209
75016	18645.027983	6726.844518	5324.456954	6.959292
79178	20794.644349	7488.322787	5953.669893	7.280694
83340	23108.575956	8303.437126	6652.439195	7.852796
87502	25458.121647	9207.493867	7257.325205	8.348944
91664	28043.796018	10056.109485	7972.599117	8.921670
95826	30542.227235	10968.274053	8660.855141	9.229023
99988	33257.671426	11952.890318	9468.190276	9.562580

Fonte: Elaborado pelo autor (2023).

3.2.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 6: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 25% ordenado



Fonte: Elaborado pelo autor (2023).

Tabela 6: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está 25% ordenado

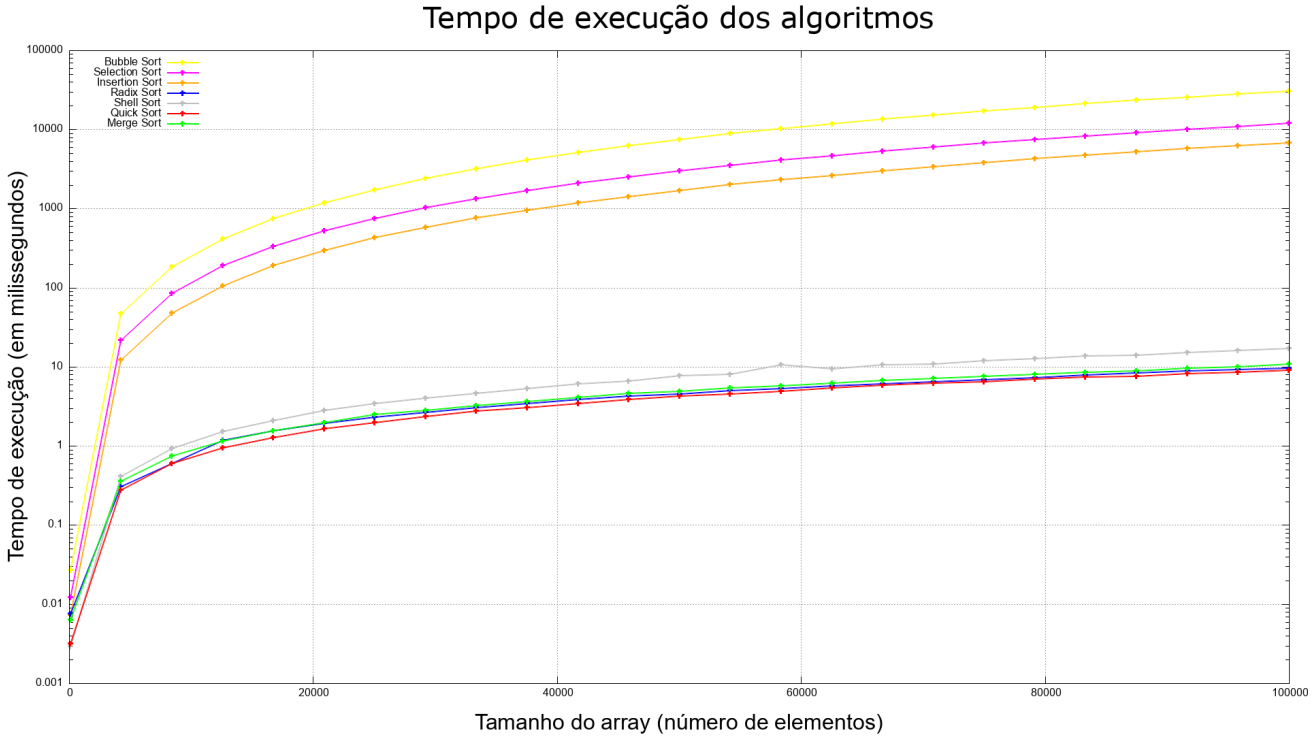
Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.002898	0.003466	0.006522	0.007632
4262	0.426402	0.308126	0.390615	0.304724
8424	0.967028	0.667759	0.864463	0.595602
12586	1.535868	1.022069	1.280835	1.184607
16748	2.203853	1.358558	1.742775	1.533562
20910	2.769258	1.748997	2.210656	1.935435
25072	3.620245	2.115721	2.677449	2.308978
29234	4.015542	2.568347	3.145563	2.676885
33396	4.731301	2.980369	3.628767	3.058300
37558	5.465195	3.379466	4.156146	3.438898
41720	6.240797	3.891756	4.632358	3.843758
45882	7.092968	4.200022	5.127930	4.199014
50044	7.808209	4.566831	5.543380	4.571371
54206	8.196754	4.997929	6.120227	4.965141
58368	11.381759	5.464877	6.557501	5.325065
62530	9.764913	5.865056	7.111062	5.720562
66692	10.767584	6.366634	7.531251	6.089769
70854	11.336107	6.652813	8.120213	6.561209
75016	12.203576	7.079098	8.615580	6.959292
79178	12.780549	7.520011	9.165449	7.280694
83340	13.895036	7.865808	9.541787	7.852796
87502	14.265099	8.514869	10.056066	8.348944
91664	15.637622	9.076825	10.711814	8.921670
95826	16.349277	9.342361	11.195329	9.229023
99988	17.555164	9.876132	11.715350	9.562580

Fonte: Elaborado pelo autor (2023).

3.3 Cenário em que o arranjo está 50% ordenado

3.3.1 Os 7 algoritmos

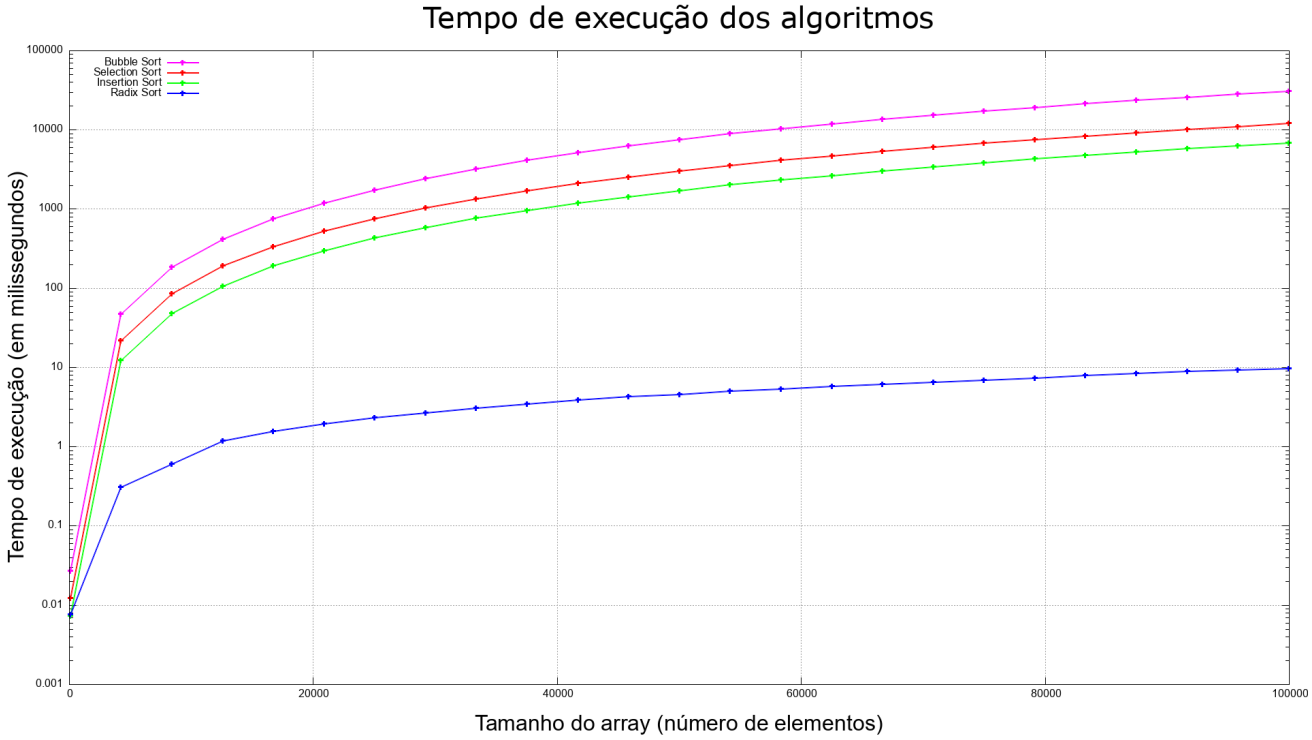
Figura 7: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está 50% ordenado



Fonte: Elaborado pelo autor (2023).

3.3.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 8: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 50% ordenado



Fonte: Elaborado pelo autor (2023).

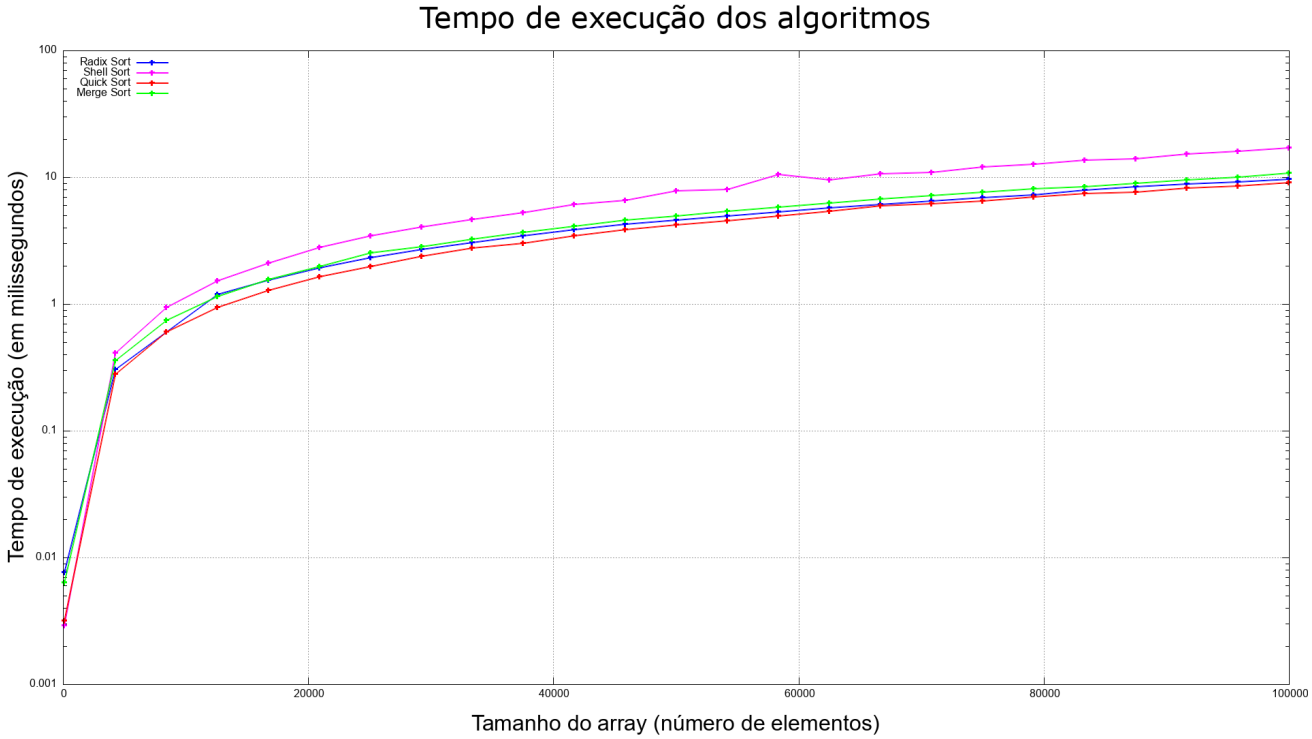
Tabela 7: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está 50% ordenado

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.026812	0.012054	0.007250	0.007572
4262	47.143332	21.690103	12.130547	0.305846
8424	185.261301	85.265933	48.016231	0.600102
12586	416.429519	189.654367	106.102153	1.188505
16748	747.664342	336.145860	191.840875	1.544564
20910	1186.890191	523.587866	293.004157	1.923973
25072	1732.082742	752.909740	430.934234	2.326910
29234	2411.344520	1023.728126	578.979176	2.676043
33396	3185.802640	1335.370947	759.670344	3.064186
37558	4106.680207	1690.153128	959.374526	3.451284
41720	5109.960227	2086.350055	1178.345543	3.857252
45882	6264.269302	2521.793181	1427.034629	4.234303
50044	7519.172568	2999.927823	1704.383742	4.569197
54206	8874.939102	3537.202861	2007.545064	4.959240
58368	10292.227805	4083.321214	2315.207408	5.345119
62530	11814.219210	4671.665377	2635.558065	5.704204
66692	13429.070151	5318.952413	3003.230488	6.104007
70854	15240.640923	6000.756318	3408.532077	6.446278
75016	17081.859244	6724.039701	3821.170054	6.876876
79178	19041.573992	7507.401864	4255.688618	7.238536
83340	21179.903262	8314.562806	4746.365916	7.893460
87502	23347.602837	9168.102486	5226.473391	8.434457
91664	25646.447288	10059.983470	5724.917041	8.817480
95826	28013.488569	10981.408501	6233.243781	9.173125
99988	30488.481551	11974.786707	6726.301355	9.593274

Fonte: Elaborado pelo autor (2023).

3.3.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 9: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 50% ordenado



Fonte: Elaborado pelo autor (2023).

Tabela 8: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está 50% ordenado

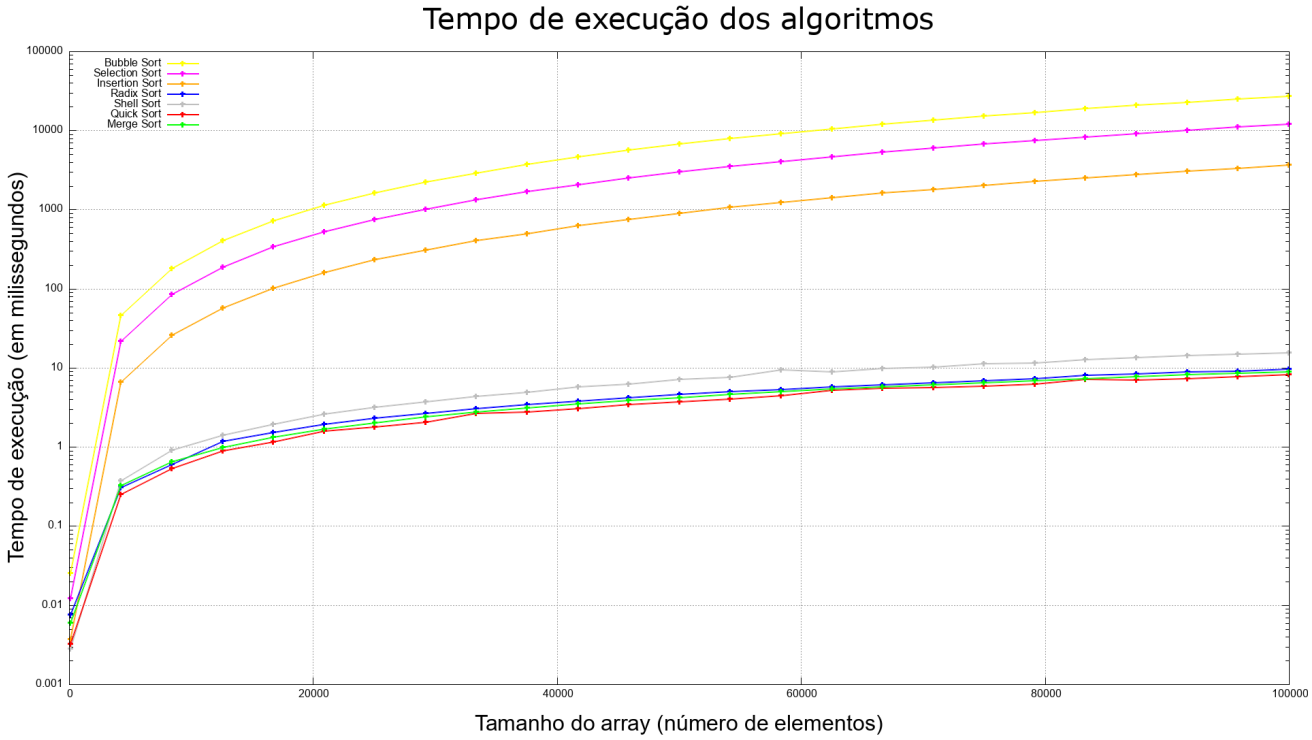
Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.002898	0.003170	0.006292	0.007572
4262	0.411963	0.279137	0.357427	0.305846
8424	0.935030	0.598298	0.740070	0.600102
12586	1.524708	0.939858	1.143541	1.188505
16748	2.097949	1.277436	1.556936	1.544564
20910	2.792798	1.645167	1.973508	1.923973
25072	3.451583	1.977864	2.519765	2.326910
29234	4.055625	2.369150	2.813415	2.676043
33396	4.619208	2.757082	3.232764	3.064186
37558	5.283630	3.025757	3.664125	3.451284
41720	6.094365	3.443942	4.122333	3.857252
45882	6.587403	3.876952	4.592296	4.234303
50044	7.777299	4.226713	4.941184	4.569197
54206	7.973257	4.523695	5.379516	4.959240
58368	10.518234	4.944670	5.787116	5.345119
62530	9.480671	5.377608	6.267725	5.704204
66692	10.683262	5.915198	6.747113	6.104007
70854	10.930830	6.196802	7.188079	6.446278
75016	11.973731	6.489700	7.597160	6.876876
79178	12.607237	7.008145	8.100091	7.238536
83340	13.611038	7.428396	8.460109	7.893460
87502	13.987945	7.646359	8.928448	8.434457
91664	15.282856	8.172242	9.533169	8.817480
95826	16.105612	8.553305	10.002016	9.173125
99988	16.941979	9.071497	10.726817	9.593274

Fonte: Elaborado pelo autor (2023).

3.4 Cenário em que o arranjo está 75% ordenado

3.4.1 Os 7 algoritmos

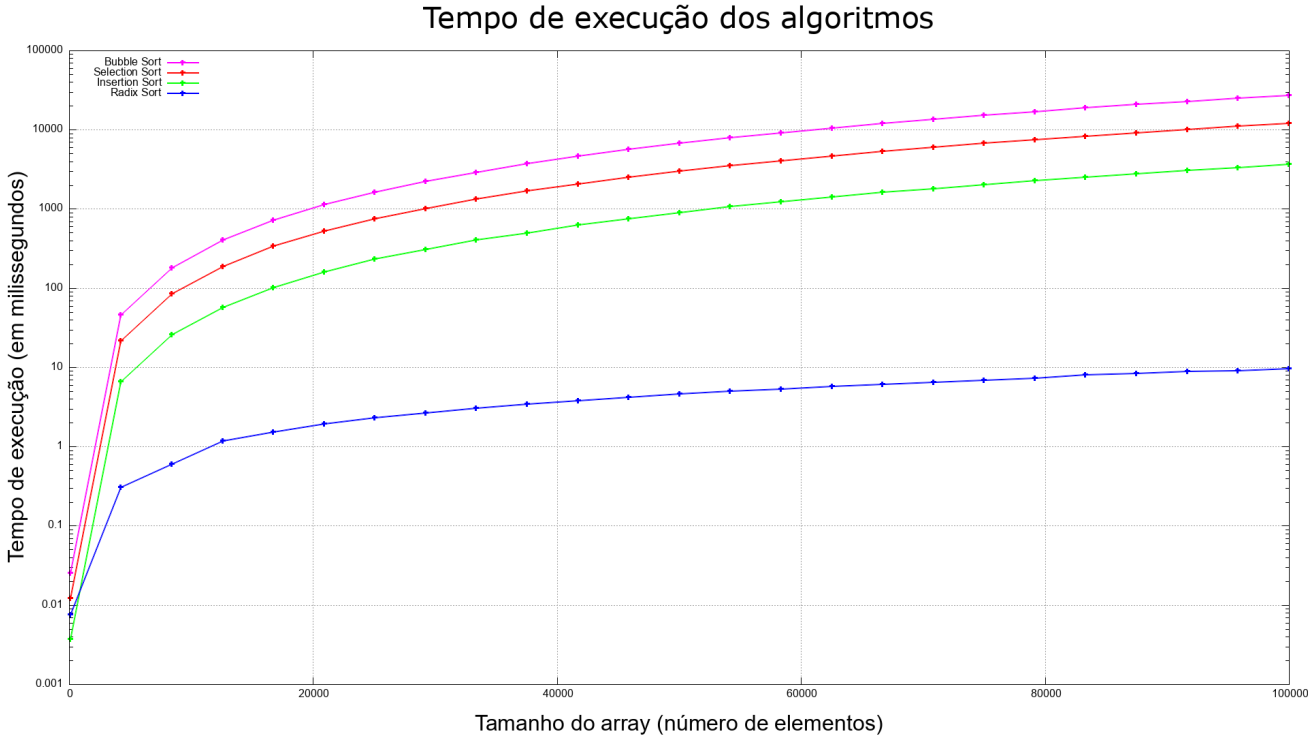
Figura 10: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está 75% ordenado



Fonte: Elaborado pelo autor (2023).

3.4.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 11: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 75% ordenado



Fonte: Elaborado pelo autor (2023).

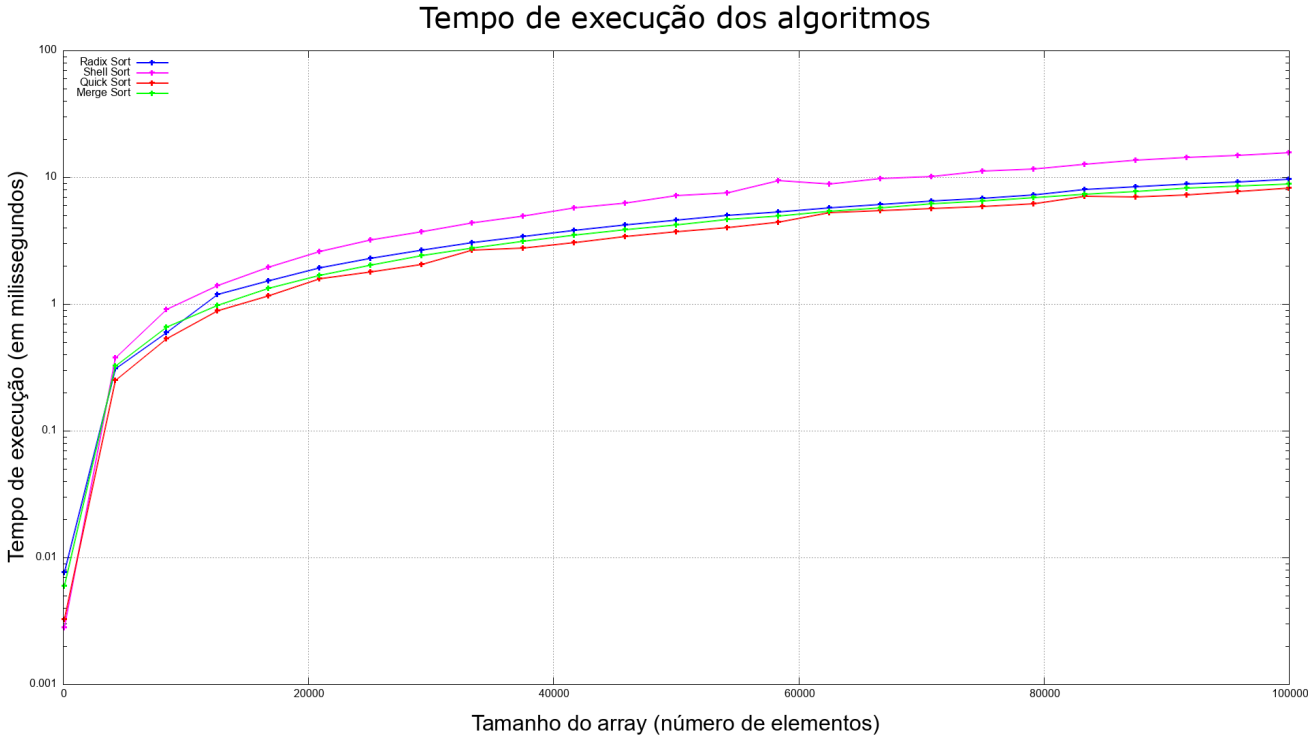
Tabela 9: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está 75% ordenado

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.025264	0.012068	0.003688	0.007608
4262	46.098405	21.709353	6.538326	0.309198
8424	179.530215	84.621105	26.033283	0.596440
12586	404.243571	188.762456	57.419789	1.185857
16748	727.397870	336.885432	101.844098	1.532220
20910	1131.798226	525.764107	160.362326	1.933967
25072	1632.161508	751.374120	231.084920	2.305306
29234	2222.921731	1018.488449	307.397157	2.673754
33396	2912.029470	1331.670952	403.636976	3.058624
37558	3705.077879	1687.618099	497.864768	3.421108
41720	4610.188007	2079.547717	628.925206	3.825294
45882	5608.850620	2518.012710	757.054025	4.210754
50044	6721.003760	3006.398162	903.398927	4.575045
54206	7901.057524	3510.755587	1070.090588	4.989077
58368	9176.153858	4076.079414	1233.198629	5.309701
62530	10544.609034	4673.985752	1413.817033	5.708786
66692	11993.595377	5315.288950	1609.238754	6.120053
70854	13536.364597	6009.743819	1805.688388	6.464208
75016	15199.803208	6727.595886	2027.908181	6.818560
79178	16981.302888	7500.958945	2280.647887	7.213549
83340	18798.376290	8303.070749	2498.517040	7.977535
87502	20765.398455	9163.929725	2778.186270	8.399240
91664	22822.175186	10061.078065	3073.117678	8.810926
95826	24939.942190	10999.434260	3325.469135	9.127530
99988	27168.967093	11987.811666	3654.841585	9.659083

Fonte: Elaborado pelo autor (2023).

3.4.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 12: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está 75% ordenado



Fonte: Elaborado pelo autor (2023).

Tabela 10: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está 75% ordenado

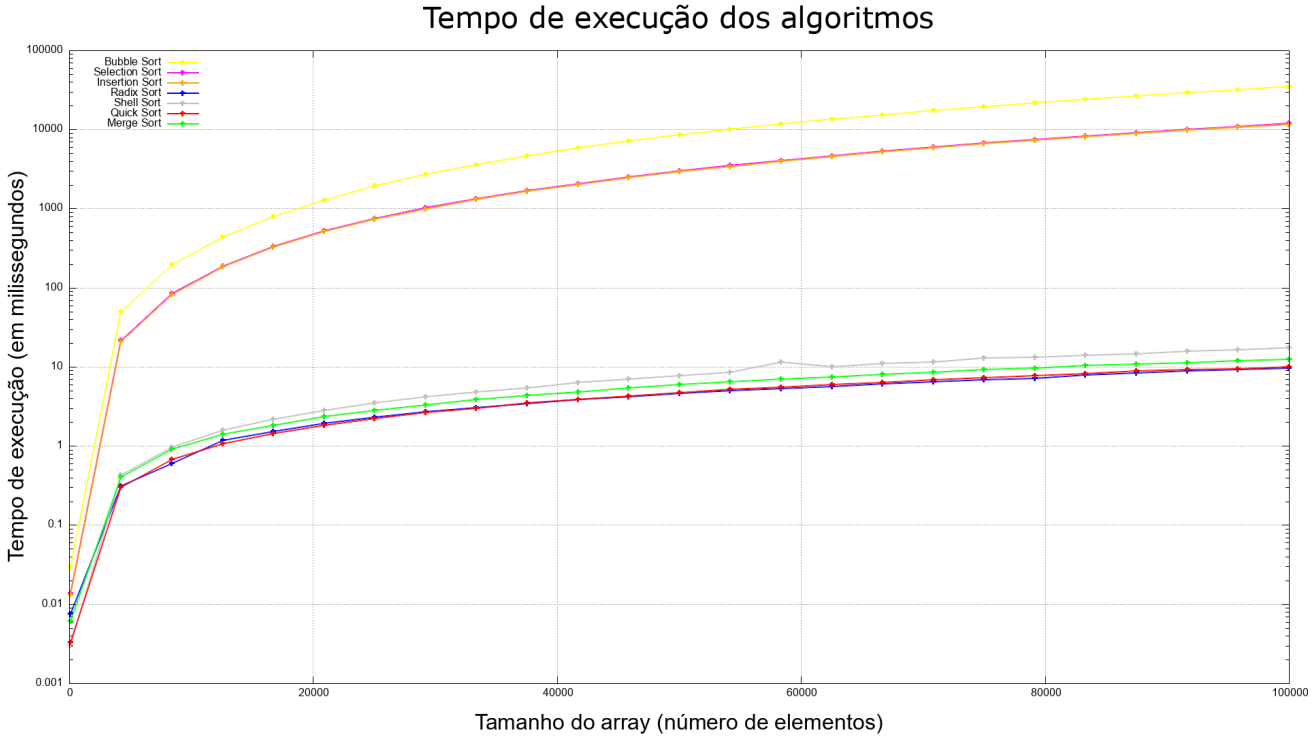
Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.002804	0.003236	0.005962	0.007608
4262	0.374829	0.248953	0.323340	0.309198
8424	0.902178	0.534477	0.654885	0.596440
12586	1.400298	0.888515	0.978831	1.185857
16748	1.944473	1.154693	1.326751	1.532220
20910	2.587879	1.585349	1.688326	1.933967
25072	3.200249	1.786315	2.024208	2.305306
29234	3.736080	2.058861	2.399807	2.673754
33396	4.378911	2.643272	2.751825	3.058624
37558	4.914002	2.763376	3.122151	3.421108
41720	5.737736	3.048838	3.478327	3.825294
45882	6.222845	3.414412	3.847014	4.210754
50044	7.153867	3.714028	4.206057	4.575045
54206	7.525044	3.986730	4.647064	4.989077
58368	9.368129	4.399395	4.953926	5.309701
62530	8.833804	5.233384	5.373946	5.708786
66692	9.772889	5.471243	5.753565	6.120053
70854	10.109279	5.666431	6.145422	6.464208
75016	11.228553	5.840126	6.505544	6.818568
79178	11.541722	6.201778	6.889637	7.213549
83340	12.650026	7.090682	7.356269	7.977535
87502	13.569980	6.950570	7.720199	8.399240
91664	14.356099	7.230875	8.224114	8.810926
95826	14.826956	7.690784	8.500074	9.127530
99988	15.552790	8.165006	8.877615	9.659083

Fonte: Elaborado pelo autor (2023).

3.5 Cenário em que o arranjo está completamente desordenado

3.5.1 Os 7 algoritmos

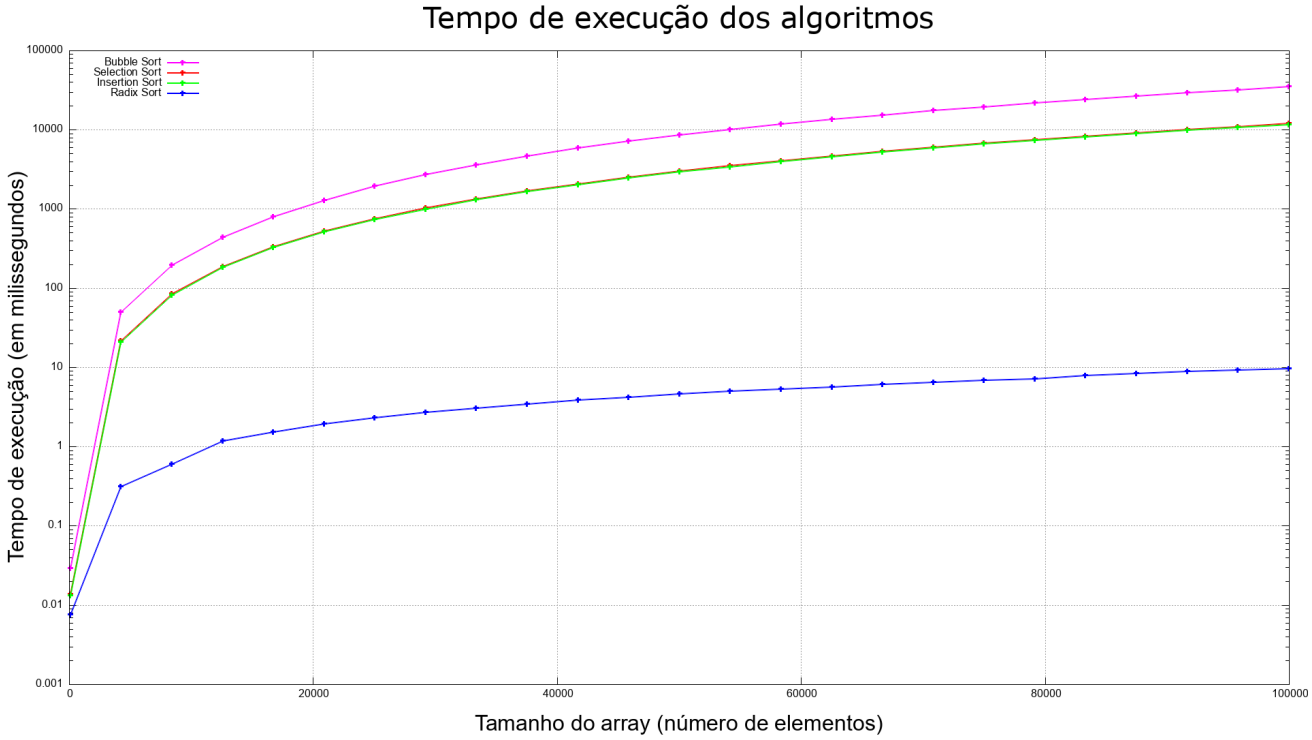
Figura 13: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está completamente desordenado



Fonte: Elaborado pelo autor (2023).

3.5.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 14: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está completamente desordenado



Fonte: Elaborado pelo autor (2023).

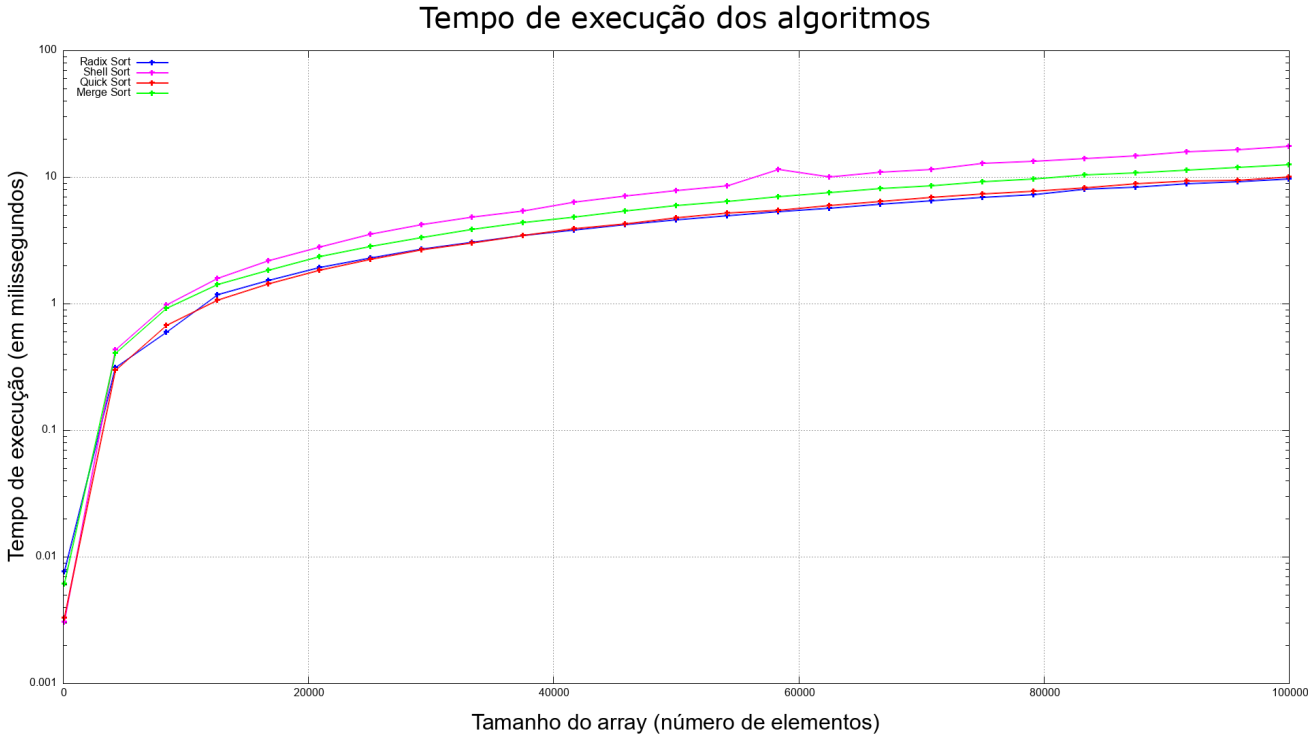
Tabela 11: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está completamente desordenado

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.028860	0.013546	0.013006	0.007592
4262	49.655532	21.697149	20.720292	0.313380
8424	193.729033	84.546972	81.086263	0.597498
12586	438.913226	188.923597	182.467216	1.180489
16748	795.841486	334.673745	325.628126	1.530654
20910	1276.360105	522.315839	510.065670	1.923581
25072	1932.907122	750.718311	729.249280	2.300146
29234	2692.469007	1024.760731	997.655508	2.690973
33396	3606.221668	1335.508772	1297.267036	3.050699
37558	4674.279556	1682.695867	1641.538834	3.435594
41720	5829.129241	2076.509935	2033.069849	3.828823
45882	7109.666190	2509.703377	2444.551390	4.201020
50044	8582.495267	3002.277864	2916.674056	4.586000
54206	10077.635851	3514.297273	3403.088704	4.952674
58368	11735.511018	4073.133500	3980.482686	5.344223
62530	13470.284394	4676.967836	4538.416482	5.682657
66692	15373.605066	5322.525690	5189.247837	6.095505
70854	17358.374117	6003.505545	5840.631731	6.451498
75016	19507.982975	6733.586210	6567.163523	6.902035
79178	21751.611633	7498.780478	7333.570329	7.207881
83340	24078.963228	8313.200754	8095.040812	7.953385
87502	26659.294080	9184.159759	8968.417572	8.340160
91664	29247.968446	10075.324276	9812.865886	8.795438
95826	31949.346519	10984.173223	10713.648782	9.196615
99988	34790.287851	11962.238398	11627.438946	9.600516

Fonte: Elaborado pelo autor (2023).

3.5.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 15: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está completamente desordenado



Fonte: Elaborado pelo autor (2023).

Tabela 12: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está completamente desordenado

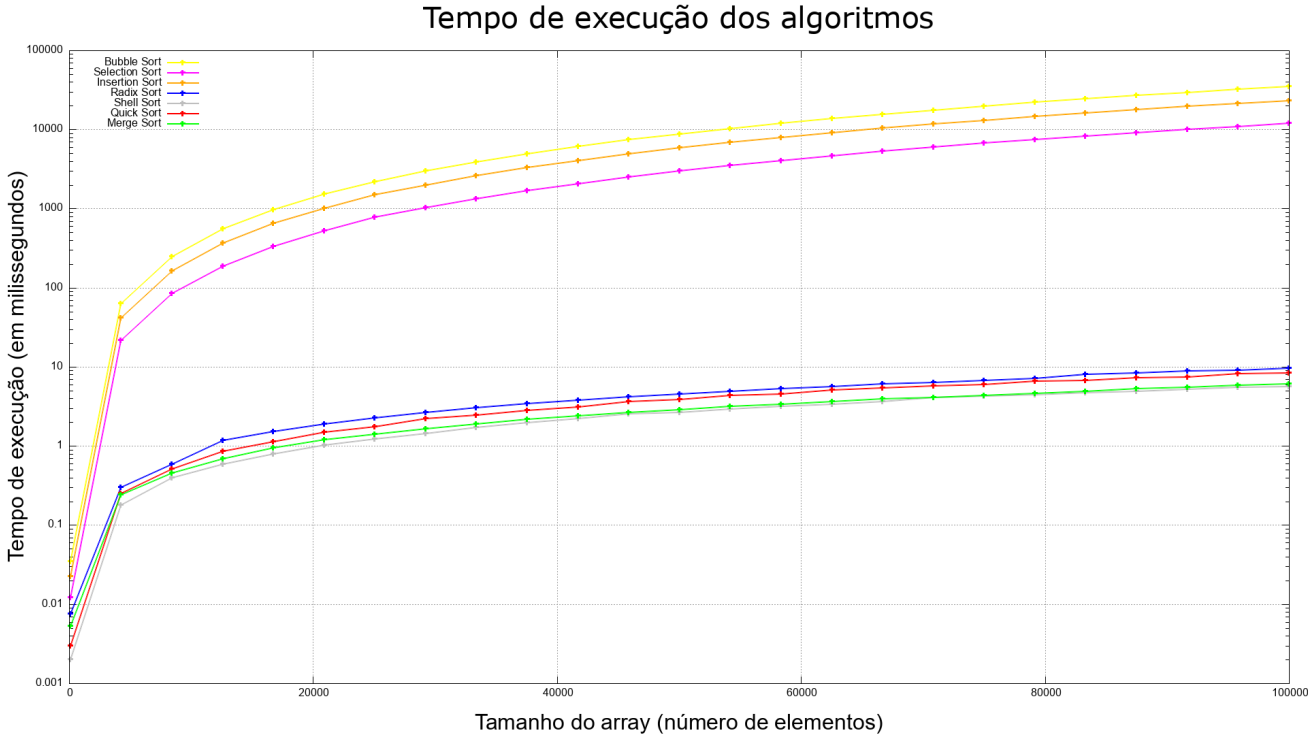
Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.003030	0.003270	0.006098	0.007592
4262	0.432548	0.297870	0.404873	0.313380
8424	0.970719	0.671897	0.915728	0.597498
12586	1.573162	1.069790	1.414022	1.180489
16748	2.191762	1.434213	1.833061	1.530654
20910	2.810740	1.823224	2.350517	1.923581
25072	3.525639	2.230917	2.828221	2.300146
29234	4.213043	2.659641	3.329539	2.690973
33396	4.846073	2.998093	3.844728	3.050699
37558	5.404464	3.467377	4.368449	3.435594
41720	6.327604	3.894075	4.844025	3.828823
45882	7.061240	4.237453	5.384496	4.201020
50044	7.786352	4.735432	5.963605	4.586000
54206	8.486296	5.156863	6.431969	4.952674
58368	11.413396	5.477205	6.978608	5.344223
62530	10.004122	5.977158	7.484243	5.682657
66692	10.951460	6.392802	8.072937	6.095505
70854	11.390438	6.904579	8.558261	6.451498
75016	12.872366	7.318300	9.149847	6.902035
79178	13.319990	7.747940	9.645997	7.207881
83340	13.949914	8.237171	10.332138	7.953385
87502	14.685390	8.836374	10.795192	8.340160
91664	15.785098	9.242989	11.368120	8.795438
95826	16.310763	9.416376	11.947943	9.196615
99988	17.378781	10.040732	12.468583	9.600516

Fonte: Elaborado pelo autor (2023).

3.6 Cenário em que o arranjo está em ordem não-crescente

3.6.1 Os 7 algoritmos

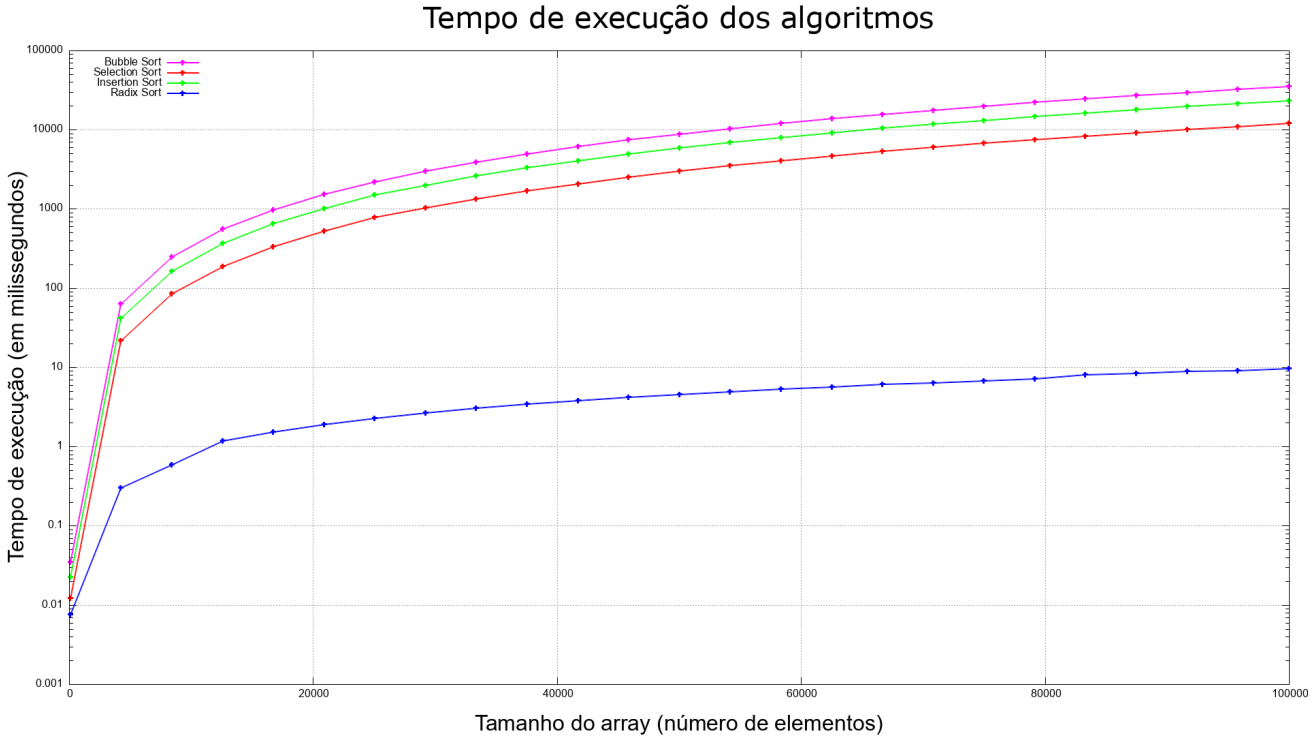
Figura 16: Tempo de execução dos 7 algoritmos de ordenação, em escala logarítmica no eixo y, quando o arranjo está em ordem não-crescente



Fonte: Elaborado pelo autor (2023).

3.6.2 Os algoritmos com caso médio $O(n^2)$ e o *Radix Sort*

Figura 17: Tempo de execução dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está em ordem não-crescente



Fonte: Elaborado pelo autor (2023).

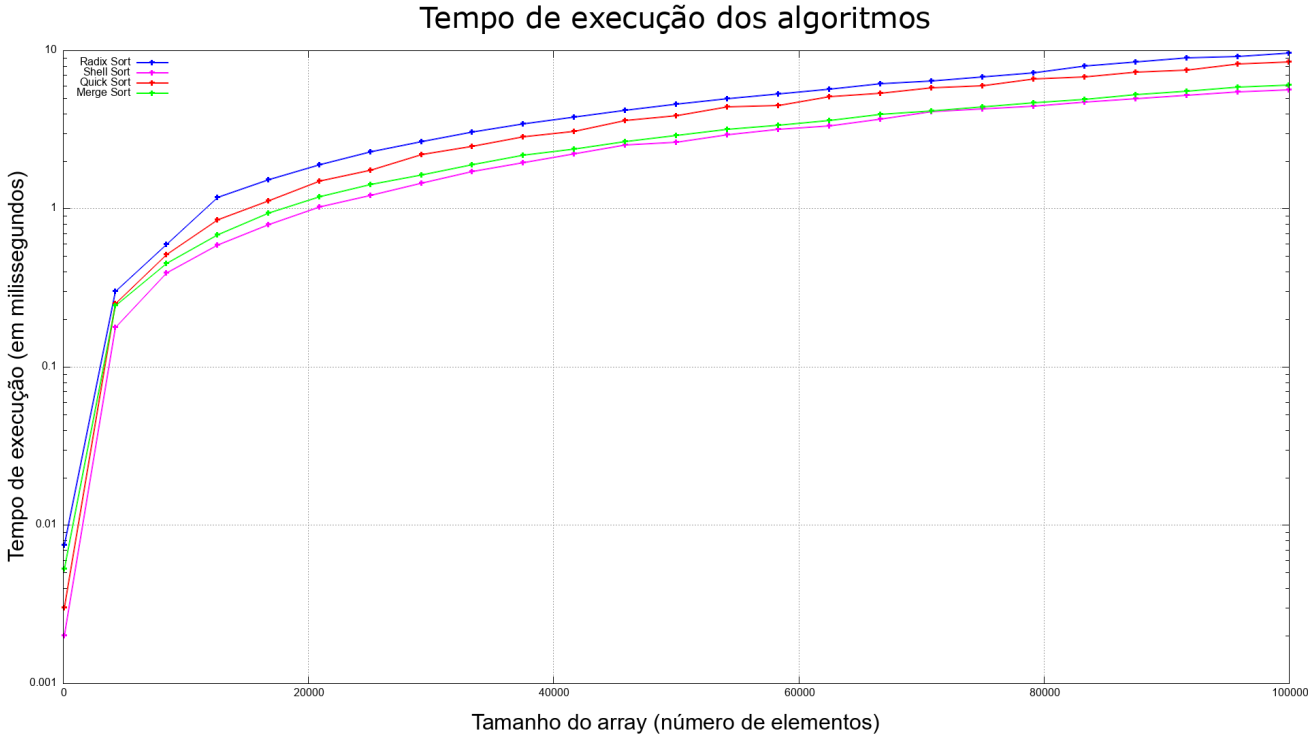
Tabela 13: Tempos de execução (em ms) dos algoritmos com caso médio $O(n^2)$ e o *Radix Sort* quando o arranjo está em ordem não-crescente

Tamanho do array	Bubble Sort	Selection Sort	Insertion Sort	Radix Sort
100	0.034870	0.012054	0.022282	0.007496
4262	63.161589	21.710945	41.515487	0.301420
8424	247.424244	84.874338	164.159055	0.593210
12586	552.974629	189.396105	367.616176	1.175983
16748	980.320368	334.758214	654.640881	1.528412
20910	1535.174216	521.791672	1016.373724	1.902401
25072	2198.287626	780.253165	1497.865225	2.278288
29234	3024.412220	1035.034338	1996.061220	2.665662
33396	3910.258801	1333.446154	2599.210169	3.037809
37558	4945.046984	1684.054163	3290.990534	3.422532
41720	6112.108312	2079.084673	4060.415767	3.797165
45882	7394.994512	2514.482707	4912.018602	4.189120
50044	8802.630727	2988.564480	5843.849097	4.571701
54206	10328.516827	3509.885249	6851.683253	4.946916
58368	11978.736495	4069.224211	7948.324318	5.296583
62530	13744.019242	4669.283471	9122.316862	5.684650
66692	15627.853106	5311.791960	10380.388625	6.150516
70854	17645.863864	5996.144356	11725.354277	6.403455
75016	19814.463077	6712.261713	13119.433491	6.775890
79178	22065.349158	7507.618646	14648.151528	7.189663
83340	24459.466017	8317.910938	16232.887828	7.979521
87502	26957.343807	9169.989162	17889.311059	8.438303
91664	29523.466319	10040.960729	19591.965651	8.942102
95826	32258.808954	10971.173337	21414.372336	9.137668
99988	35122.307495	11956.501884	23312.065538	9.657477

Fonte: Elaborado pelo autor (2023).

3.6.3 Os algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*

Figura 18: Tempo de execução dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort*, em escala logarítmica no eixo y, quando o arranjo está em ordem não-crescente



Fonte: Elaborado pelo autor (2023).

Tabela 14: Tempos de execução (em ms) dos algoritmos com caso médio $O(n \log n)$, *Shell Sort* e o *Radix Sort* quando o arranjo está em ordem não-crescente

Tamanho do array	Shell Sort	Quick Sort	Merge Sort	Radix Sort
100	0.002010	0.002994	0.005296	0.007496
4262	0.178222	0.250323	0.243167	0.301420
8424	0.393335	0.512849	0.452426	0.593210
12586	0.590784	0.852891	0.685511	1.175983
16748	0.792884	1.123239	0.940728	1.528412
20910	1.026113	1.491224	1.191797	1.902401
25072	1.214706	1.756071	1.420212	2.278288
29234	1.446445	2.205486	1.637875	2.665662
33396	1.712850	2.474734	1.887707	3.037809
37558	1.951651	2.840581	2.165788	3.422532
41720	2.220059	3.092392	2.390031	3.797165
45882	2.531829	3.613957	2.640294	4.189120
50044	2.627046	3.874906	2.889043	4.571701
54206	2.914598	4.376093	3.178249	4.946916
58368	3.156845	4.504304	3.362347	5.296583
62530	3.333917	5.109528	3.623387	5.684650
66692	3.669735	5.377044	3.933513	6.150516
70854	4.098899	5.790455	4.136201	6.403455
75016	4.259361	5.992886	4.389365	6.775890
79178	4.422245	6.613976	4.663130	7.189663
83340	4.701525	6.764754	4.891666	7.979521
87502	4.931128	7.264396	5.270522	8.438303
91664	5.179505	7.474345	5.519645	8.942102
95826	5.479075	8.171510	5.882366	9.137668
99988	5.623599	8.422289	6.044146	9.657477

Fonte: Elaborado pelo autor (2023).

4 DISCUSSÃO

Nessa seção apresentaremos uma discussão acerca dos resultados obtidos. De forma geral, o *Radix Sort*, *Quick Sort* e o *Merge Sort* se mostraram os mais rápidos ao passo que o *Bubble Sort*, salvo as situações com pequenas quantidades de elementos, se mostrou o mais lento. Os tópicos a seguir abordarão discussões mais específicas e aprofundadas sobre os resultados da análise empírica feita nos algoritmos de ordenação.

4.1 Algoritmos recomendados

Os desempenhos dos algoritmos variaram consideravelmente entre os diferentes cenários, de modo que é fundamental analisá-los levando em conta cada contexto.

No cenário em que o array está em ordem não-decrescente (Figura 1), o *Insertion Sort* se destaca com o melhor desempenho para todas as dimensões de entrada. Isso ocorre pois essa configuração corresponde ao melhor cenário para o *Insertion Sort* em que sua complexidade é $O(n)$.

Quanto ao cenário em que 25% do arranjo já está ordenado, as curvas do *Quick Sort* e do *Radix Sort* apresentam os menores valores e quase coincidem, como pode ser observado na Figura 4 e Figura 6. Dessa forma, eles são igualmente recomendáveis. Contudo, vale ressaltar que o *Radix Sort* é limitado a trabalhar somente com números inteiros positivos e o *Quick Sort* não é um algoritmo estável, pois é capaz de alterar a ordem relativa dos elementos com chaves iguais durante o processo de ordenação. Assim, tais fatores precisam ser levados em consideração na tomada de decisão.

Com relação ao caso em que conjunto de dados está 50% ordenado, é perceptível, ao analisar as Figuras 7 e 9, que o *Quick Sort*, *Merge Sort* e o *Radix Sort* são os algoritmos com menores tempos de execução e apresentam velocidades semelhantes, com uma discreta diferença de cerca de 1 milissegundo (conforme indicado na Tabela 8), que pode ser justificada pela variabilidade natural dos dados. No entanto, o *Merge Sort* requer memória extra, o que pode se tornar um empecilho ao lidar com grandes volumes de dados. Com base nisso, em termos de desempenho temporal, qualquer um dos três poderia ser selecionado, contudo, assim como no caso anterior, é necessário considerar as restrições já mencionadas desses algoritmos.

De forma análoga ao cenário anterior, no caso em que o arranjo está 75% ordenado (Figura 12), o *Quick Sort*, *Merge Sort* e *Radix Sort* são indicados, desde que suas respectivas limitações sejam levadas em conta.

A dinâmica dos dados no cenário em que o conjunto está totalmente desordenado (Figura 13 e Figura 15) é semelhante à situação em que o arranjo está 25% ordenado e o *Quick Sort* e o *Radix Sort* são sugeridos como a melhor escolha.

No cenário em que o arranjo se encontra em ordem não-crescente (Figura 16 e

Figura 18), *Shell Sort* e o *Merge Sort* são recomendados. No entanto, se a situação envolve restrição de memória, o *Shell Sort* torna-se a opção mais adequada, uma vez que sua complexidade de espaço é inferior em comparação à do *Merge Sort*. Por outro lado, se a estabilidade for um requisito, o *Merge Sort* deve ser o algoritmo de escolha.

4.2 Análise do *Radix Sort*

Diferente da maioria dos algoritmos de ordenação, tais como o *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Shell Sort*, *Merge Sort* e *Quick Sort*, que funcionam mediante comparações diretas entre os elementos a serem ordenados, o *Radix Sort* se destaca como um algoritmo não-comparativo. Nesse sentido, é relevante avaliar o desempenho do *Radix Sort* em contraste com os algoritmos baseados na comparação de chaves.

Na maior parte dos cenários avaliados - quando o arranjo está 25% ordenado, 50% ordenado, 75% ordenado e totalmente desordenado - o *Quick Sort* e o *Merge Sort* apresentam performances semelhantes, ultrapassando os demais algoritmos de ordenação e foram considerados os melhores algoritmos de comparação de chave. Além disso, nestes mesmos cenários, o *Radix Sort* demonstra um desempenho muito próximo desses dois algoritmos, com uma diferença de poucos milissegundos (como pode ser visto na Tabela 6, Tabela 8, Tabela 10 e Tabela 12) e, portanto, é considerado equivalente em termos de tempo de execução.

4.3 *Quick Sort* vs. *Merge Sort*

Como esperado, tanto o *Quick Sort* como o *Merge Sort* obtiveram resultados bem semelhantes. Mesmo assim, por uma margem pequena, o *Quick Sort* se mostrou mais rápido nas situações trabalhadas. No entanto, a vantagem ter sido pequena indica que o tempo de execução do algoritmo não é um fator determinante para diferenciá-los.

Nesse caso, a escolha dentre um dos dois deve ser feita baseando-se no contexto no qual o algoritmo será implementado. Enquanto o *Quick Sort* é mais eficiente em termos de espaço de armazenamento, o *Merge Sort* é melhor no quesito estabilidade.

Portanto, em termos de velocidade, mesmo com uma pequena diferença para um deles, ambos algoritmos são igualmente eficientes.

4.4 Comportamento anômalo

No pior cenário, o *Merge Sort* tem uma complexidade temporal de $\Theta(n \log n)$ (CORMEN et al., 2012), enquanto o *Quick Sort* possui uma complexidade de $O(n^2)$ (DROZDEK, 2013). No entanto, na implementação realizada para este trabalho, usamos a técnica da mediana de três na escolha do pivô, minimizando assim a possibilidade de particionamentos desbalanceados. Assim, o *Quick Sort* raramente alcança seu pior caso, e a

complexidade considerada foi $O(n \log n)$. Por sua vez, o *Radix Sort*, apresenta complexidade no pior caso de $O(n)$ (DROZDEK, 2013). A expectativa inicial era que o *Radix Sort* se sairia melhor em relação aos algoritmos de maior complexidade que se baseiam em comparação, mas, conforme mencionado anteriormente, ele apresentou um desempenho análogo. Uma justificativa possível para isso são as características da arquitetura do computador utilizado no experimento ou o fato de que o *Radix Sort* precisa buscar o maior elemento de cada entrada e determinar a quantidade de dígitos desse elemento. Desse modo, um estudo mais aprofundado é necessário para entender melhor as razões das diferenças observadas.

Além disso, apesar do Shell Sort possuir uma complexidade teórica de $O(n^2)$, ele exibiu um desempenho atípico em nossos testes. Ao invés de se assemelhar aos algoritmos com comportamento quadrático, seus valores de tempo de execução se aproximaram dos algoritmos com complexidade $O(n \log n)$ e do *Radix Sort*.

Analisando os gráficos, nota-se que o comportamento dos algoritmos foram bem padronizados ao longo do intervalo presente no gráfico. Embora houvessem flutuações em alguns gráficos, as variações presentes não foram expressivas o suficiente para gerar um resultado diferente do esperado. Mesmo que o *Shell Sort* tenha tido um comportamento anômalo, os seus gráficos não possuem picos ou vales significantes.

4.5 Uma estimativa matemática

Para estimar quanto tempo seria necessário para ordenar um vetor com 10^12 elementos, foi considerado que estes estarão completamente desordenados. Além disso, para o cálculo, utilizaremos a relação de ordem dos algoritmos.

4.5.1 Algoritmos com complexidade de ordem linear $O(n)$

A fim de estimar o tempo de execução dos algoritmos lineares, considere que o tempo de execução de um algoritmo desse tipo em um vetor com n elementos é proporcional a n. Portanto, segue que:

$$\frac{t_n}{n} = \frac{t_{10^{12}}}{10^{12}} \implies t_{10^{12}} = \frac{t_n \cdot 10^{12}}{n}$$

onde $t_n,\ t_{10^{12}} \in \mathbb{R}_+$ é o tempo de execução para n e 10^{12} elementos respectivamente.

Dessa forma, utilizando a Tabela 11, tomaremos $n = 99988$ e t_n é igual ao tempo que o respectivo algoritmo levou para ordenar o vetor com 99988 elementos. Assim, temos:

Tabela 15: Tempos de execução dos algoritmos linear ordenando 10^{12} elementos

Unidade de Medida	Radix Sort
Milissegundos	$9.60167 \cdot 10^7$
Horas	26.67130

Fonte: Elaborado pelo autor (2023).

4.5.2 Algoritmos com complexidade de ordem quadrática $O(n^2)$

A fim de estimar o tempo de execução dos algoritmos quadratícos, considere que o tempo de execução de um algoritmo desse tipo em um vetor com n elementos é proporcional a n^2. Portanto, segue que:

$$\frac{t_n}{n^2} = \frac{t_{10^{12}}}{(10^{12})^2} \implies t_{10^{12}} = \frac{t_n \cdot (10^{12})^2}{n^2}$$

onde $t_n,\ t_{10^{12}} \in \mathbb{R}_+$ é o tempo de execução para n e 10^{12} elementos respectivamente.

Dessa forma, utilizando a Tabela 11, tomaremos $n = 99988$ e t_n é igual ao tempo que o respectivo algoritmo levou para ordenar o vetor com 99988 elementos. Assim, temos:

Tabela 16: Tempos de execução dos algoritmos quadráticos ordenando 10^{12} elementos

Unidade de Medida	Bubble Sort	Selection Sort	Insertion Sort
Milissegundos	$3.47903 \cdot 10^{18}$	$1.19622 \cdot 10^{18}$	$1.16274 \cdot 10^{18}$
Anos	110319318	37931887	36870243

Fonte: Elaborado pelo autor (2023).

4.5.3 Algoritmos com complexidade de ordem logarítmica $O(n \cdot \log_2 n)$

A fim de estimar o tempo de execução dos algoritmos logarítmicos, considere que o tempo de execução de um algoritmo desse tipo em um vetor com n elementos é proporcional a $n \cdot \log_2 n$. Seja $t_n,\ t_{10^{12}} \in \mathbb{R}_+$ o tempo de execução para n e 10^{12} elementos respectivamente. Portanto, segue que:

$$\frac{t_n}{n \cdot \log_2 n} = \frac{t_{10^{12}}}{10^{12} \cdot \log_2 10^{12}} \implies t_{10^{12}} = \frac{t_n \cdot (10^{12})^2}{n^2}$$

Dessa forma, utilizando a Tabela 11, tomaremos $n = 99988$ e t_n é igual ao tempo que o respectivo algoritmo levou para ordenar o vetor com 99988 elementos. Assim, temos:

Tabela 17: Tempos de execução dos algoritmos logarítmicos ordenando 10^{12} elementos

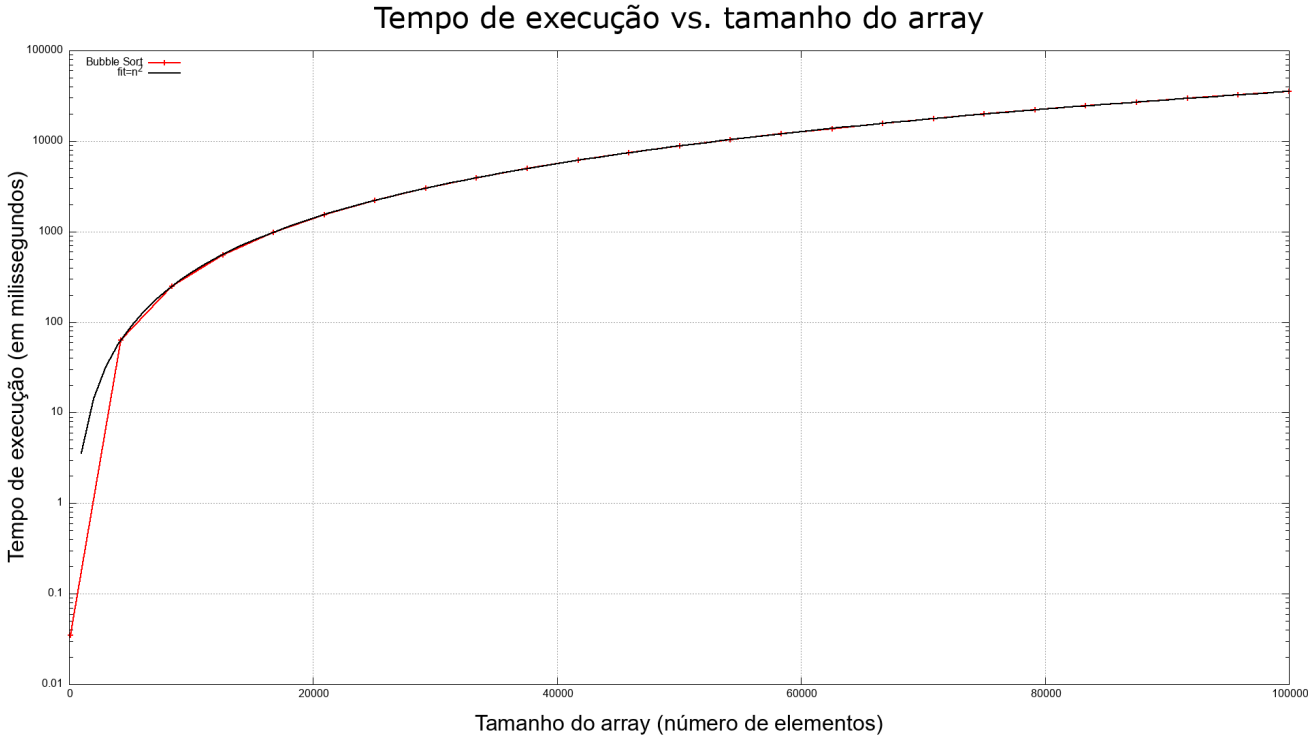
Unidade de Medida	Shell Sort	Quick Sort	Merge Sort
Milissegundos	$4.17091 \cdot 10^8$	$2.40978 \cdot 10^8$	$2.99246 \cdot 10^8$
Horas	115.85861	66.93833	83.12389

Fonte: Elaborado pelo autor (2023).

4.6 Análise matemática

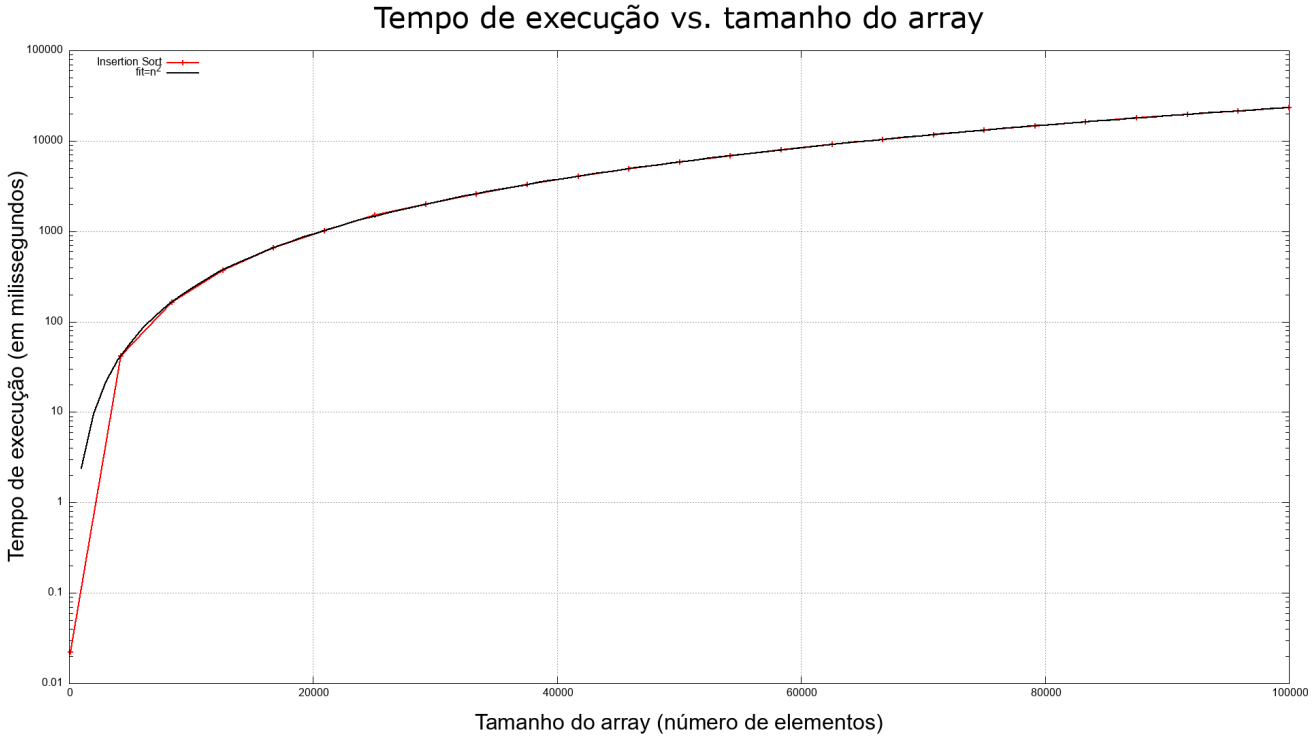
O processo de *fitting* é usado para verificar se os dados coletados durante a análise empírica correspondem à expectativa teórica, isto é, se eles seguem o modelo teórico proposto. Nesse sentido, pensando em responder à pergunta "A análise empírica é compatível com a análise matemática?", submetemos os dados do pior caso dos 7 algoritmos de ordenação ao ajuste de curva.

Figura 19: Aplicação do *fitting* no *Bubble Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

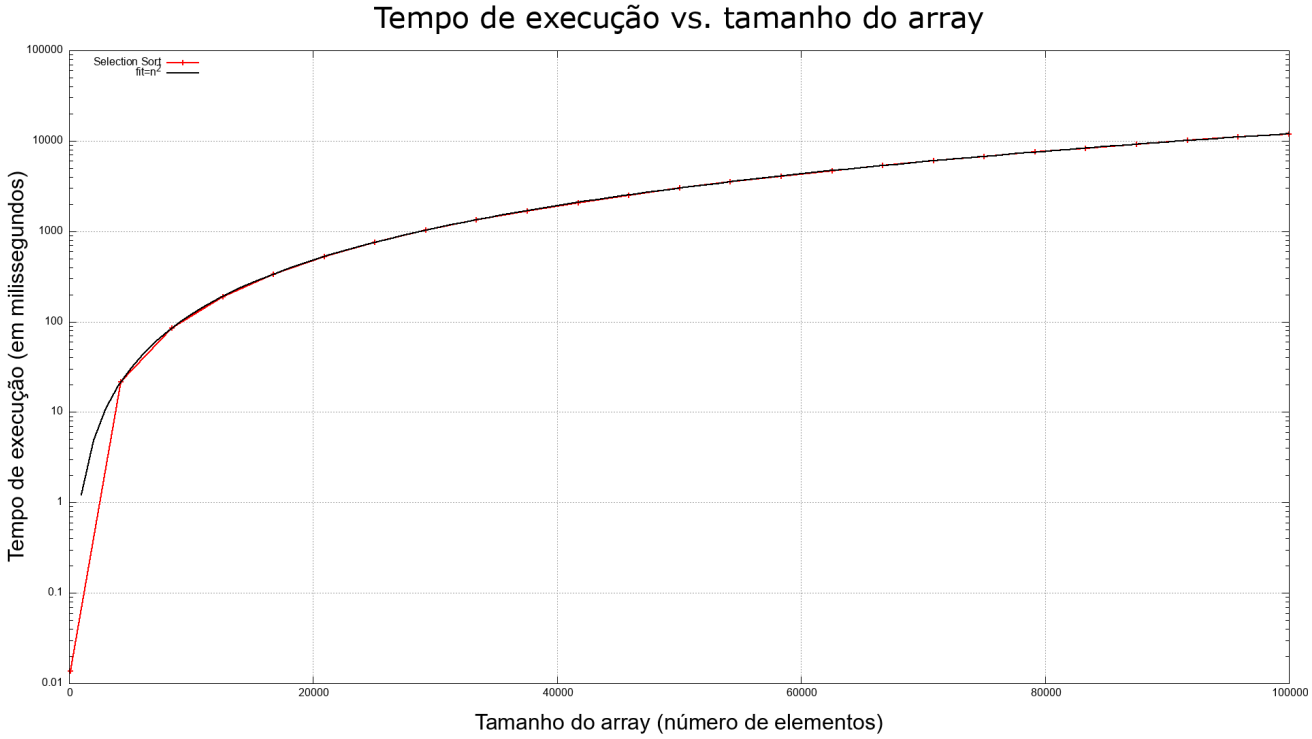
Figura 20: Aplicação do *fitting* no *Insertion Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

Em relação ao *Bubble Sort* e *Insertion Sort* a literatura indica que sua complexidade no pior caso, em que o arranjo está em ordem não-crescente, corresponde a $O(n^2)$ (DROZDEK, 2013; WEISS, 2014). A aplicação do ajuste de curva nesses algoritmos sob essas condições, ilustrado nas Figuras 19 e 20, mostra claramente que a função do tipo $f(n) = n^2$ está sobreposta aos os dados coletados, corroborando com a ideia de que esses dados expressam um comportamento quadrático.

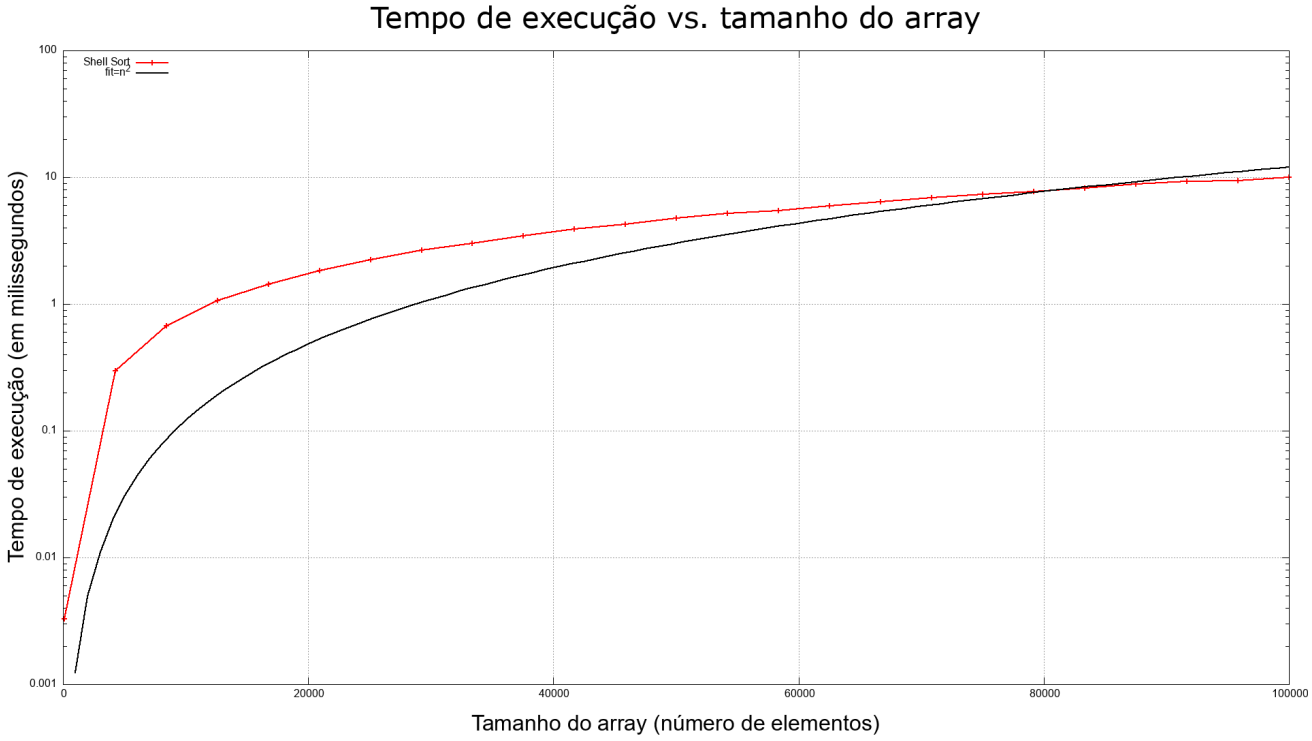
Figura 21: Aplicação do *fitting* no *Selection Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

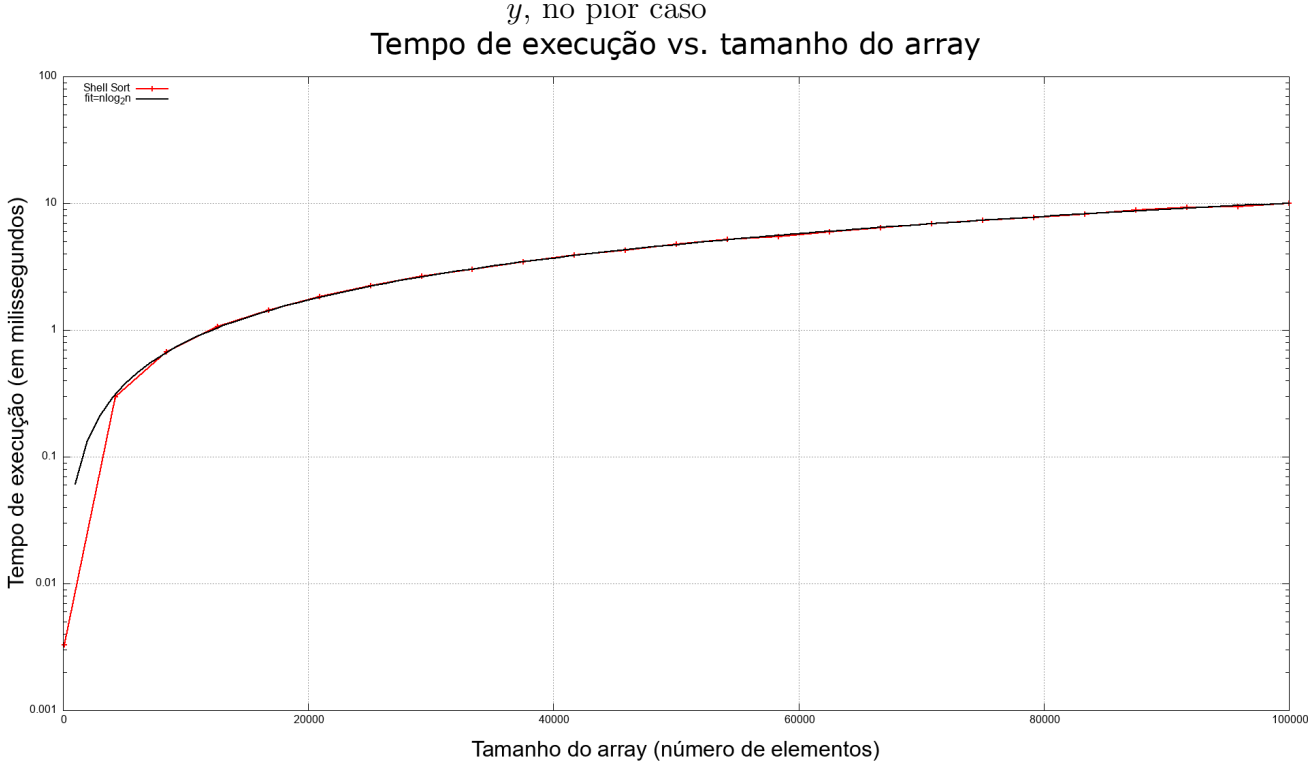
O *Selection Sort* também possui complexidade $O(n^2)$ no pior caso (DROZDEK, 2013). No entanto, devido à dificuldade de determinar o pior caso para este algoritmo, uma vez que os valores se assemelham e alternam na maior parte dos cenários, optamos por realizar o *fitting* quando o arranjo está completamente desordenado. Nessa perspectiva, o resultado obtido (Figura 21) demonstrou que os dados de tempo de execução do *Selection Sort* estão de acordo com as previsões teóricas.

Figura 22: Aplicação do *fitting* no *Shell Sort* com n^2, em escala logarítmica no eixo y, no pior caso



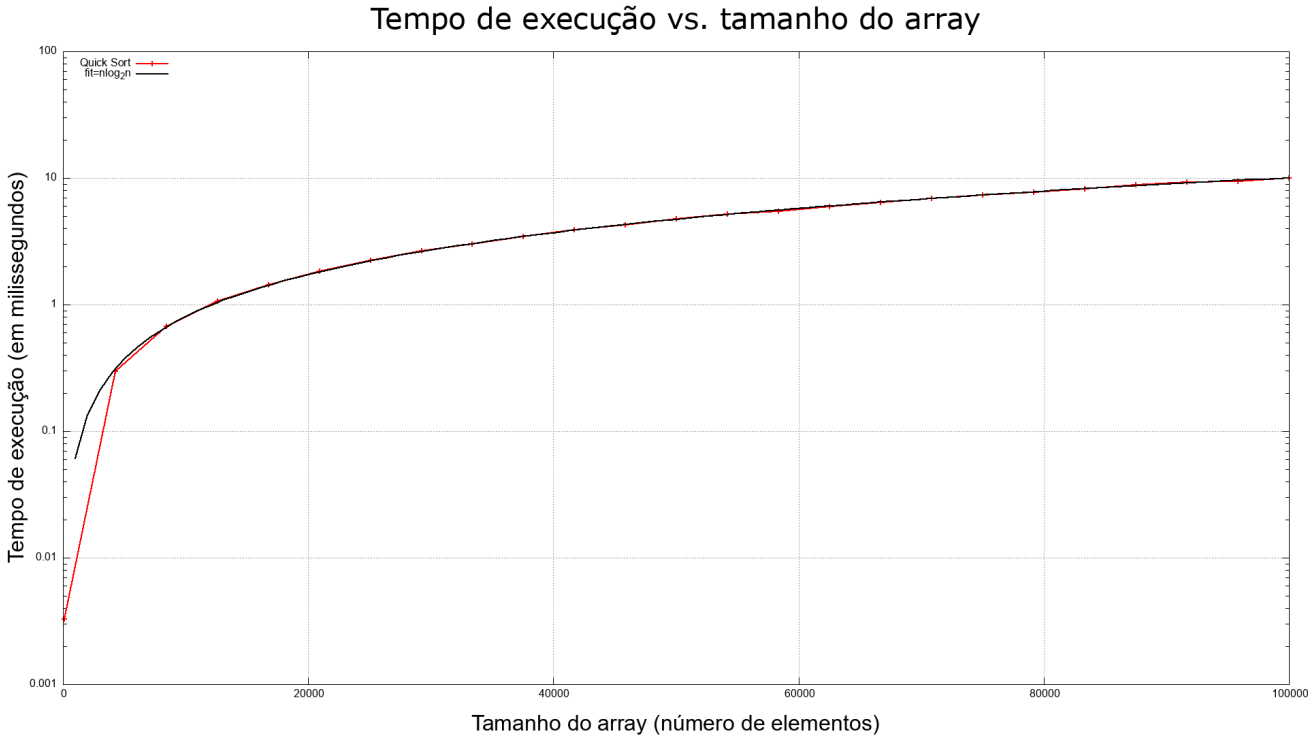
Fonte: Elaborado pelo autor (2023).

Figura 23: Aplicação do *fitting* no *Shell Sort* com $n \log_2 n$, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

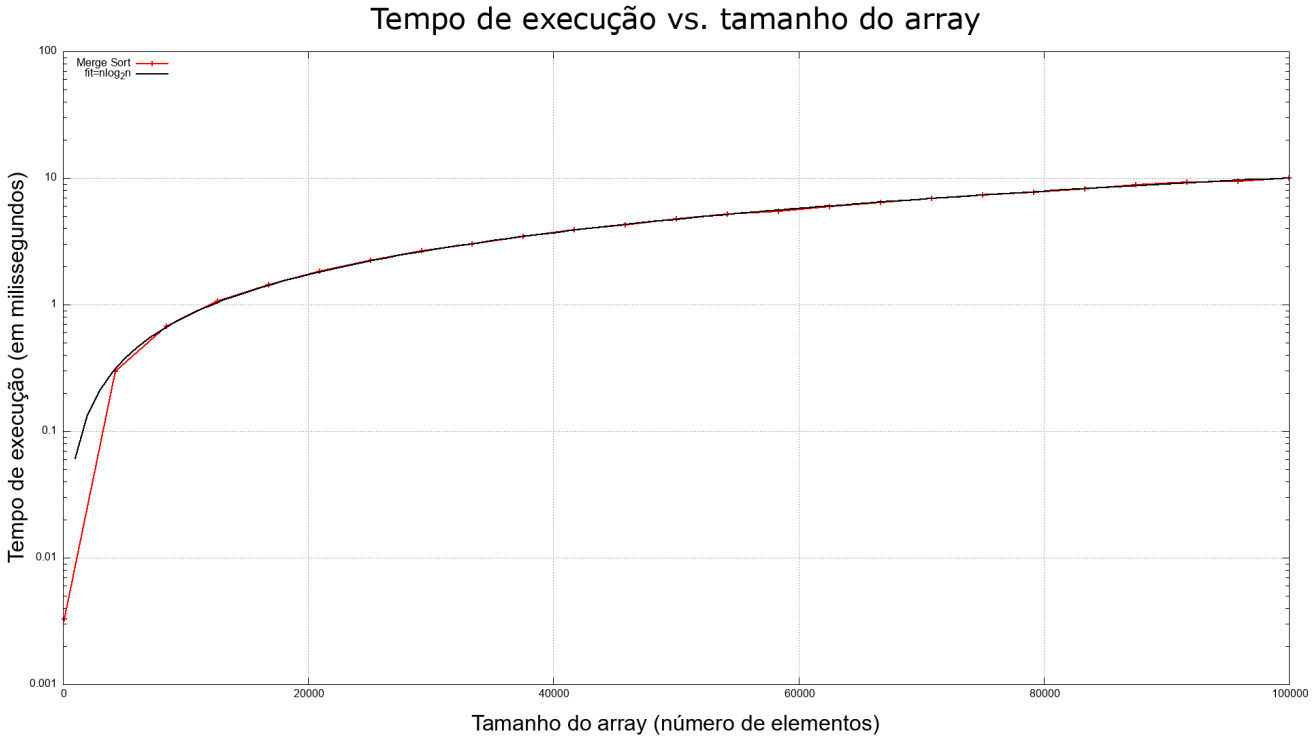
Figura 24: Aplicação do *fitting* no *Quick Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

O *Shell Sort* e *Quick Sort*, assim como os algoritmos anteriores, apresentam uma complexidade de $O(n^2)$ no pior caso (DROZDEK, 2013). Entretanto, para o *Quick Sort*, devido a utilização da mediana de três, considerou-se a complexidade $O(n \log n)$. Além disso, os experimentos indicam que o cenário mais adverso para esses algoritmos ocorre quando o array está completamente desordenado, condição sob a qual o ajuste de curva foi conduzido. Esta avaliação revelou que os dados coletados para o *Quick Sort* estão em sintonia com as previsões teóricas, como ilustrado pela similaridade entre as curvas demonstradas na Figura 24. No entanto, o *Shell Sort* não demonstrou um comportamento quadrático como era esperado, conforme apresentado na Figura 22, e suas medições se alinharam com a função $f(n) = n \log_2 n$ (Figura 23).

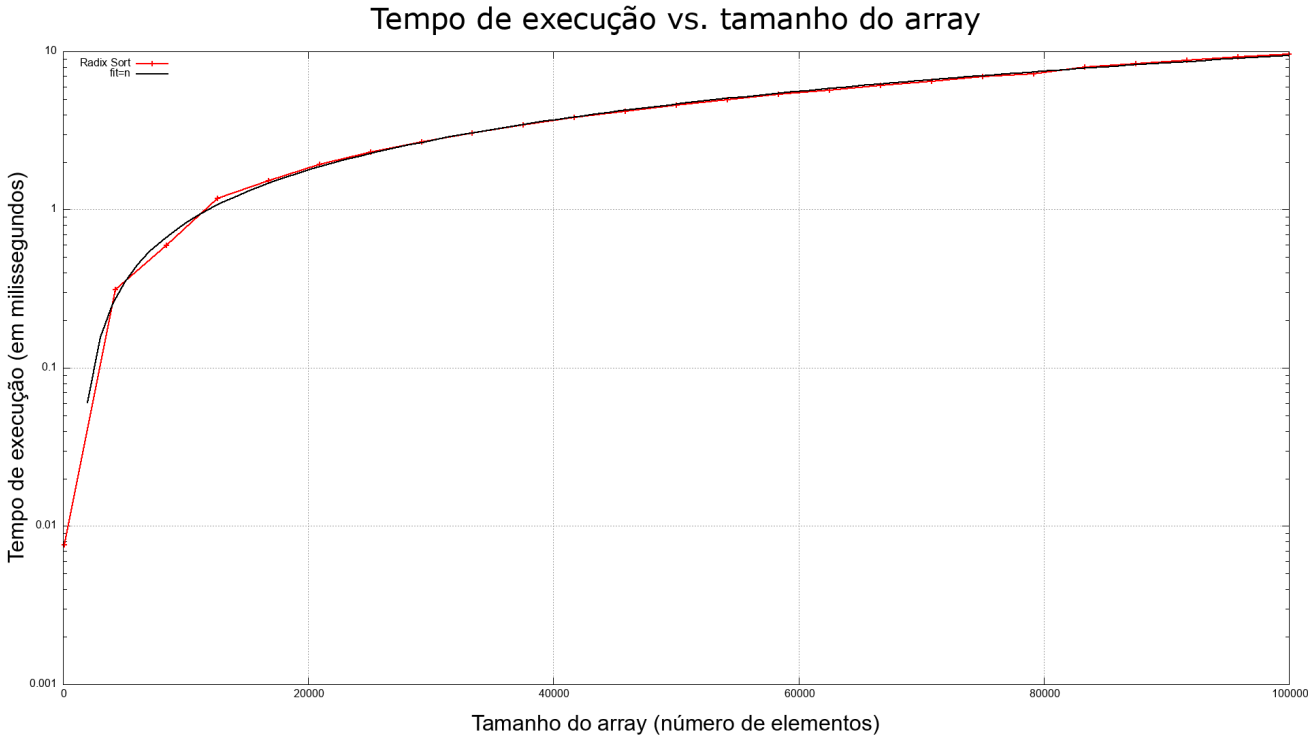
Figura 25: Aplicação do *fitting* no *Merge Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

O *Merge Sort*, por outro lado, apresenta uma complexidade de $\Theta(n \log n)$ e tem seu pior cenário quando o arranjo está completamente desordenado. Com base nisso, no processo de *fitting* (Figura 25), implementado com dados de um conjunto totalmente fora de ordem, pudemos confirmar que os resultados obtidos para este algoritmo estão em consonância com as expectativas teóricas.

Figura 26: Aplicação do *fitting* no *Radix Sort*, em escala logarítmica no eixo y, no pior caso



Fonte: Elaborado pelo autor (2023).

Por ser um algoritmo de ordenação estável e não comparativo, o *Radix Sort* não tem um pior caso da mesma forma que os algoritmos de ordenação comparativos, uma vez que ele não é influenciada pela ordem inicial dos dados. No entanto, se a distribuição dos números for muito desigual (por exemplo, se ocorrer números com um número muito maior de dígitos do que os outros), então o *Radix Sort* pode ser menos eficiente e esta seria a pior configuração para o algoritmo. Dessa forma, dado que as entradas utilizadas em todos os cenários mantêm a mesma quantidade de dígitos e as medições estão consideravelmente próximas, optamos por realizar o ajuste de curva no cenário em que o conjunto de dados está está completamente desordenado. Esse processo é evidenciado na Figura 26, onde as medições para este algoritmo exibem uma tendência linear - conforme demonstrado pelo alinhamento entre a curva da função $f(n) = n$ e os dados coletados - em concordância com o que foi apontado por Drozdek (2013).

5 CONCLUSÃO

A análise empírica é um método bastante utilizado quando é necessário verificar a eficiência de algoritmos para que seja possível compará-los e aplicá-los de modo adequado na resolução de problemas. Com base nisso, os seguintes algoritmos de ordenação foram examinados empiricamente em diferentes configurações de dados quanto ao seu tempo de execução: i) *Bubble Sort*, ii) *Selection Sort*, iii) *Insertion Sort*, iv) *Shell Sort*, v) *Merge Sort*, vi) *Quick Sort* e vii) *Radix Sort*. A maioria dos dados de tempo de execução coletados, com a exceção daqueles referentes ao *Shell Sort*, foram considerados alinhados com o modelo teórico sugerido para os algoritmos, o que permitiu a realização de uma análise baseada em dados alinhados com a literatura existente, levando em conta eventuais restrições.

Nesse contexto, os resultados obtidos indicaram que em grande parte dos cenários - isto é, quando o conjunto de dados está 25%, 50%, 75% ordenado ou totalmente desordenado - os algoritmos *Merge Sort*, *Quick Sort* e *Radix Sort* demonstraram um desempenho comparável e apresentaram-se como os mais eficientes. Assim, é necessário ponderar as limitações e benefícios de cada algoritmo e fazer a escolha apropriada para cada situação específica.

Entretanto, dado o comportamento linear do *Radix Sort*, esse resultado foi surpreendente, de modo que futuras investigações são encorajadas para um melhor entendimento dessa dinâmica.

Além disso, vale ressaltar que, em particular, no caso em que os dados estão em ordem não-decrescente, o Insertion Sort se destacou pelo seu notável desempenho, superando os algoritmos de complexidade $O(n \log n)$.

À vista disso, o desenvolvimento e resultados do trabalho mostraram-se bastante satisfatórios, dado que proporcionaram uma compreensão mais aprofundada sobre o comportamento dos algoritmos de ordenação e atenderam os objetivos propostos.

Referências

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos: Teoria e prática**. [S.l.]: Elsevier, 2012.

DROZDEK, A. **Data Structures and Algorithms in C++**. [S.l.]: Cengage Learning, 2013.

SEDGEWICK, R. **Algorithms in C**. New York: Addison-Wesley Publishing Company, 1998. 657 p.

SZWARCFITER, J. L. S.; MARKENZON, L. **Estruturas de Dados e Seus Algoritmos**. Rio de Janeiro: LTC — Livros Técnicos e Científicos Editora Ltda, 2015. 236 p. ISBN 978-85-216-2994-8.

WEISS, M. A. **Data structures and algorithm analysis in C++**. [S.l.]: Florida International University, 2014.