Trabalho - Ordenação

Autores: Diego Henrique Zanella / Jonathan Domingos Rodrigues

04 de Novembro de 2023

1 Bubble Sort

1.1 Bubble Sort - Tabela de Tempo

Foram realizados 5 testes e os mesmos foram realizados utilizando a plataforma Replit.

Aqui é mostrado o tempo de execução do algoritmo Bubble Sort para diferentes tamanhos de entrada, variando de 50 elementos a 10.000 elementos. Os testes foram realizados cinco vezes e a tabela apresenta os tempos de execução em milissegundos, juntamente com o cálculo da média total.

O desempenho do Bubble Sort é notável em listas pequenas, onde seu tempo de execução é bastante eficiente. No entanto, à medida que o número de elementos aumenta, o algoritmo começa a mostrar um desempenho menos eficiente. Isso ocorre porque o Bubble Sort possui uma complexidade de tempo quadrática, o que significa que seu tempo de execução aumenta significativamente à medida que o número de elementos a serem classificados aumenta.

Table 1: Tempo de Execução - Bubble Sort

Qtd. elem.	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Média
50 elementos	1	1	1	1	1	1
500 elementos	169	164	27	9	18	77,4
1000 elementos	258	74	525	16	30	180,6
5000 elementos	1617	1496	1433	323	211	1016
10000 elementos	2890	1527	1835	460	399	1422,2

1.2 Bubble Sort - Gráfico de Tempo

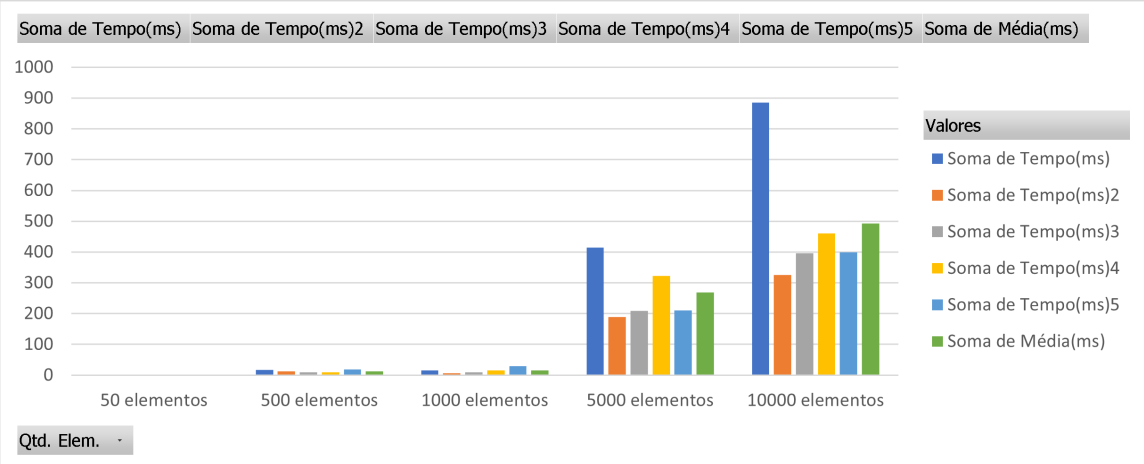

Figure 1: Tempo de Execução do Bubble Sort

1.3 Bubble Sort - Tabela de Trocas

Table 2: Trocas - Bubble Sort

Qtd. elem.	Trocas	Trocas	Trocas	Trocas	Trocas
50 elementos	548	548	548	548	548
500 elementos	64386	64386	64386	64386	64386
1000 elementos	255401	255401	255401	255401	255401
5000 elementos	6208999	6208999	6208999	6208999	6208999
10000 elementos	24877878	24877878	24877878	24877878	24877878

1.4 Bubble Sort - Tabela de Iterações

Table 3: Iterações - Bubble Sort

Qtd. elem.	Iterações	Iterações	Iterações	Iterações	Iterações
50 elementos	548	548	548	548	548
500 elementos	64386	64386	64386	64386	64386
1000 elementos	255401	255401	255401	255401	255401
5000 elementos	6208999	6208999	6208999	6208999	6208999
10000 elementos	24877878	24877878	24877878	24877878	24877878

O motivo pelo qual o número de iterações e trocas é o mesmo em todas as execuções dos testes é porque foi utilizada a mesma semente (seed) para a geração dos vetores de entrada em todos os casos. A semente é um valor inicial que determina a sequência de números pseudoaleatórios gerados. Ao manter a mesma semente, os vetores gerados em cada execução dos testes são idênticos, levando a resultados de iterações e trocas consistentes.

2 Quick Sort

2.1 Quick Sort - Tabela de Tempo

Foram realizados 5 testes e os mesmos foram realizados utilizando a plataforma Replit.

A tabela apresenta o tempo de execução do Quick Sort para diferentes tamanhos de entrada, variando de 50 elementos a 10.000 elementos. Como podemos observar, o Quick Sort demonstrou um desempenho bastante eficiente em comparação com o Bubble Sort.

Diferentemente do Bubble Sort, o Quick Sort possui uma complexidade de tempo médio de O(n log n), o que o torna mais eficiente para listas maiores. Isso se reflete nos tempos de execução registrados na tabela, onde o Quick Sort supera significativamente o Bubble Sort, especialmente à medida que o tamanho da entrada aumenta.

No entanto, é importante observar que o desempenho do Quick Sort pode variar dependendo da natureza dos dados de entrada. Em média, ele é um dos algoritmos de ordenação mais rápidos e é amplamente utilizado na prática devido à sua eficiência em cenários reais.

Table 4: Tempo de Execução - Quick Sort

Qtd. elem.	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Média
50 elementos	1	1	1	1	1	1
500 elementos	11	1	1	1	10	4,8
1000 elementos	1	1	1	66	1	14
5000 elementos	3	1	1	1	2	1,6
10000 elementos	67	1	2	4	1	15

2.2 Quick Sort - Gráfico de Tempo

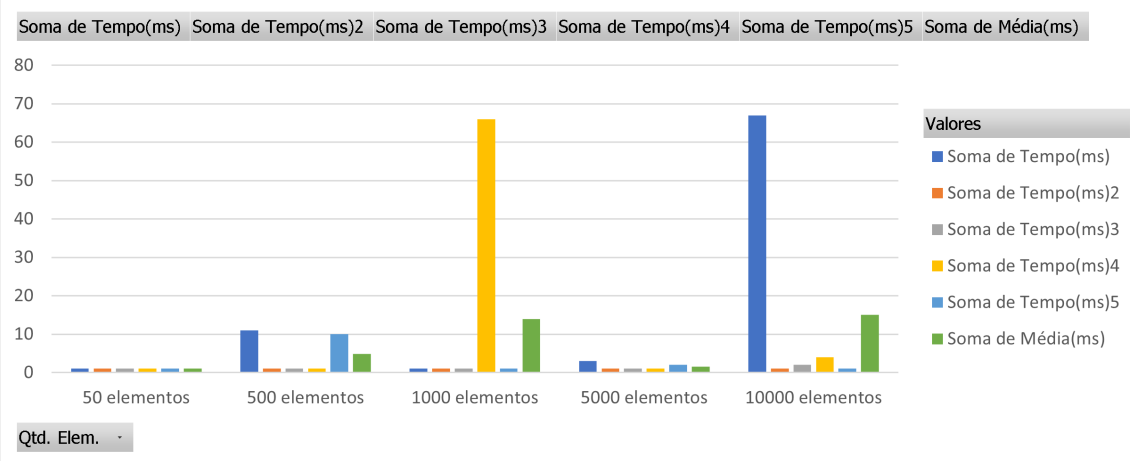

Figure 2: Tempo de Execução do Quick Sort

2.3 Quick Sort - Tabela de Trocas e Iterações

Table 5: Trocas e Iterações - Quick Sort

Qtd. elem.	Quantidade	Quantidade	Quantidade	Quantidade	Quantidade
50 elementos	30	30	30	30	30
500 elementos	304	304	304	304	304
1000 elementos	617	617	617	617	617
5000 elementos	3075	3075	3075	3075	3075
10000 elementos	6138	6138	6138	6138	6138

A tabela exibida acima apresenta a quantidade de trocas e iterações realizadas pelo algoritmo Quick Sort para diferentes tamanhos de entrada. É importante notar que em cada iteração do algoritmo Quick Sort, ocorre uma única troca quando necessário. Essa troca acontece quando a condição "if (i ¡= j)" é atendida, indicando que um elemento na posição "i" deve ser trocado com um elemento na posição "j" para manter a ordenação correta.

Portanto, a cada iteração, uma única troca é realizada, o que é representado pelo incremento da variável "iterações" no código do algoritmo. Isso faz com que o número de trocas seja igual ao número de iterações, como mencionado anteriormente.

Esse comportamento é uma característica fundamental do Quick Sort e contribui para sua eficiência, uma vez que evita trocas desnecessárias e mantém o algoritmo de ordenação rápido, especialmente em conjuntos de dados maiores.

3 Insert Sort

3.1 Insert Sort - Tabela de Tempo

Foram realizados 5 testes e os mesmos foram realizados utilizando a plataforma Replit.

O algoritmo Insertion Sort é um dos métodos mais simples de ordenação, amplamente utilizado devido à sua simplicidade e eficiência em pequenos conjuntos de dados. Ele funciona construindo a lista ordenada um item de cada vez, movendo elementos não classificados para a posição correta, de acordo com sua comparação com os elementos já ordenados. Embora não seja tão rápido quanto algoritmos mais avançados, como o Merge Sort ou o Quick Sort, o Insertion Sort é uma escolha sólida para pequenas quantidades de dados e também é útil para melhorar o desempenho de algoritmos de ordenação maiores.

Embora o Insertion Sort seja eficaz para pequenas quantidades de dados e tenha uma implementação relativamente simples, seu desempenho começa a diminuir à medida que o tamanho da lista a ser ordenada aumenta. Sua complexidade média é de O(n**2), o que significa que o tempo de execução cresce quadraticamente com o número de elementos na lista.

Table 6: Tempo de Execução - Insert Sort

Qtd. elem.	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Média
50 elementos	1	63	0	0	0	12,8
500 elementos	82	10	8	100	89	57,8
1000 elementos	15	93	13	65	13	39,8
5000 elementos	333	215	478	301	330	331,4
10000 elementos	1063	979	1112	963	1096	1042,6

3.2 Insert Sort - Gráfico de Tempo

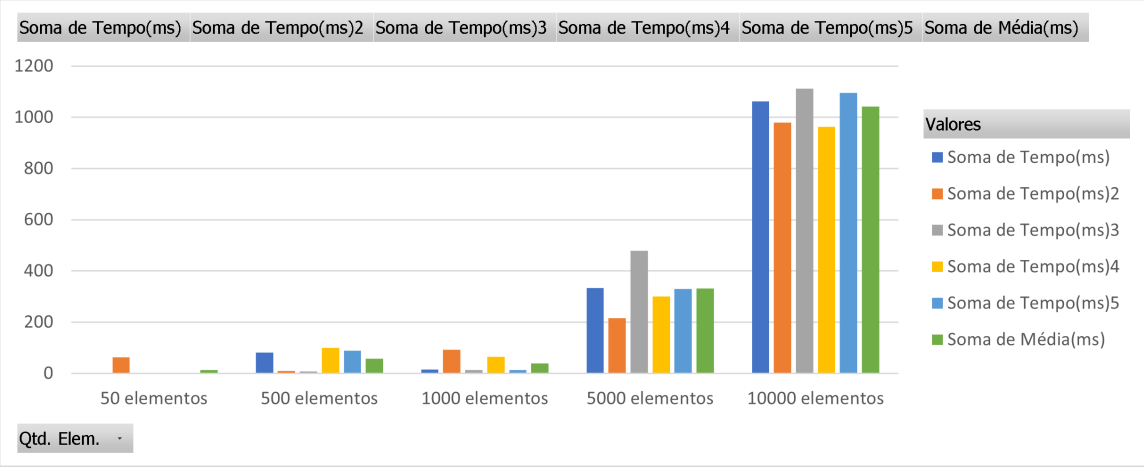

Figure 3: Tempo de Execução do Insert Sort

3.3 Insert Sort - Tabela de Trocas

Table 7: Trocas - Insert Sort

Qtd. elem.	Trocas	Trocas	Trocas	Trocas	Trocas
50 elementos	570	570	570	570	570
500 elementos	62918	62918	62918	62918	62918
1000 elementos	252534	252534	252534	252534	252534
5000 elementos	6170153	6170153	6170153	6170153	6170153
10000 elementos	24650866	24650866	24650866	24650866	24650866

O Insertion Sort compara cada elemento com os elementos anteriores na lista e realiza uma troca sempre que encontra um elemento menor que o atual. Portanto, o número de trocas é diretamente proporcional ao número de inversões ou desordens na lista.

3.4 Insert Sort - Tabela de Iterações

Table 8: Iterações - Insert Sort

Qtd. elem.	Iterações	Iterações	Iterações	Iterações	Iterações
50 elementos	2370	2370	2370	2370	2370
500 elementos	310418	310418	310418	310418	310418
1000 elementos	1160534	1160534	1160534	1160534	1160534
5000 elementos	30515153	30515153	30515153	30515153	30515153
10000 elementos	122620866	122620866	122620866	122620866	122620866

O número de iterações no algoritmo Insertion Sort varia dependendo do estado inicial da lista a ser ordenada. No pior caso, quando a lista está totalmente desordenada, o algoritmo realiza um grande número de comparações e movimentações de elementos, resultando em um número de iterações da ordem de n**2, onde 'n' é o número de elementos na lista. No entanto, em situações ideais, onde a lista já está parcialmente ordenada ou contém apenas alguns elementos fora de ordem, o número de iterações pode ser significativamente menor, tornando o Insertion Sort mais eficiente. Isso o torna uma escolha apropriada quando se lida com listas que podem estar quase ordenadas ou quando a simplicidade do algoritmo é mais importante do que o desempenho em si.

4 Shell Sort

4.1 Shell Sort - Tabela de Tempo

Foram realizados 5 testes e os mesmos foram realizados utilizando a plataforma Replit.

O algoritmo de ordenação Shell Sort é uma variação do algoritmo de inserção (Insertion Sort) que visa melhorar o desempenho deste último. Ele funciona dividindo a lista original em subconjuntos menores e ordenando-os separadamente com o Insertion Sort. Gradualmente, os subconjuntos se tornam maiores, até que a lista inteira esteja ordenada. O Shell Sort é conhecido por ser mais eficiente do que o Insertion Sort em termos de tempo de execução e é particularmente útil quando se lida com listas de tamanho moderado a grande.

O Shell Sort é conhecido por oferecer um desempenho aprimorado em relação ao Insertion Sort, especialmente em listas de tamanho médio a grande. Sua complexidade média depende da sequência de lacunas utilizada, mas em geral, é considerada mais eficiente do que o O(n**2) do Insertion Sort. O Shell Sort consegue reduzir o número de movimentos de elementos em comparação com o Insertion Sort, devido à sua abordagem de dividir e conquistar, onde os subconjuntos menores são ordenados antes de combinar todos eles para obter a lista final ordenada. No entanto, a escolha da sequência de lacunas adequada é crucial para otimizar o desempenho do Shell Sort em diferentes casos.

Table 9: Tempo de Execução - Shell Sort

Qtd. elem.	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Tempo(ms)	Média
50 elementos	0	0	1	0	0	0,2
500 elementos	9	8	39	93	81	46
1000 elementos	93	3	41	4	15	31,2
5000 elementos	491	332	525	396	590	466,8
10000 elementos	204	282	281	294	210	254,2

4.2 Shell Sort - Gráfico de Tempo

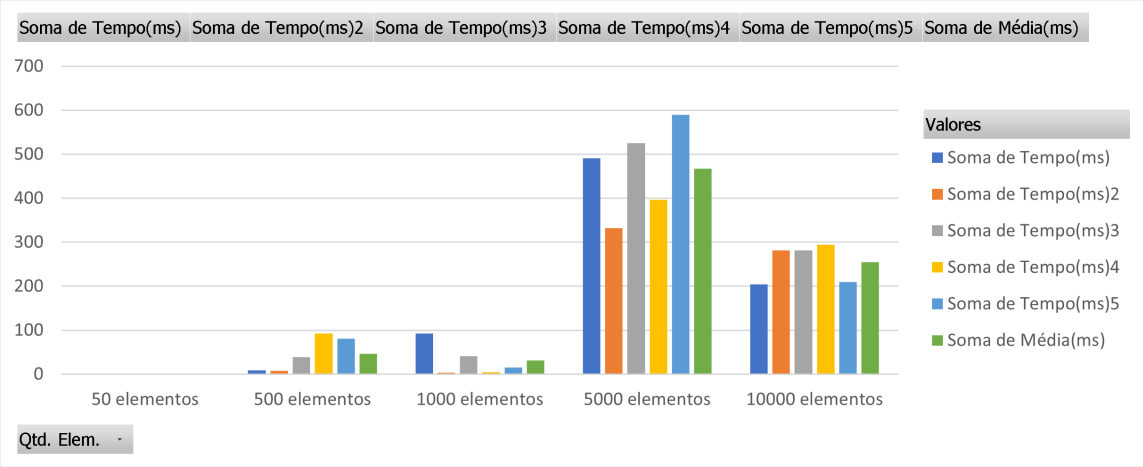

Figure 4: Tempo de Execução do Shell Sort

4.3 Shell Sort - Tabela de Trocas

Table 10: Trocas - Shell Sort

Qtd. elem.	Trocas	Trocas	Trocas	Trocas	Trocas
50 elementos	252	252	252	252	252
500 elementos	9338	9338	9338	9338	9338
1000 elementos	38172	38172	38172	38172	38172
5000 elementos	899898	899898	899898	899898	899898
10000 elementos	3794269	3794269	3794269	3794269	3794269

O número de trocas no algoritmo Shell Sort também está intrinsecamente relacionado à sequência de lacunas escolhida e à natureza da lista a ser ordenada. Em cada iteração do Shell Sort, à medida que os subconjuntos menores são ordenados, as trocas de elementos ocorrem entre os elementos adjacentes em cada

subconjunto. O objetivo das sequências de lacunas é minimizar o número de trocas necessárias para obter a ordenação final. Em geral, o Shell Sort tende a realizar menos trocas do que o Insertion Sort, uma vez que os subconjuntos menores são parcialmente ordenados antes da combinação em subconjuntos maiores. No entanto, o número de trocas pode variar significativamente com base na distribuição inicial dos elementos na lista. Em casos onde a lista já está parcialmente ordenada, o número de trocas pode ser muito baixo, tornando o Shell Sort bastante eficiente.

4.4 Shell Sort - Tabela de Iterações

Table 11: Iterações - Shell Sort

Qtd. elem.	Iterações	Iterações	Iterações	Iterações	Iterações
50 elementos	1350	1350	1350	1350	1350
500 elementos	140500	140500	140500	140500	140500
1000 elementos	450000	450000	450000	450000	450000
5000 elementos	14445000	14445000	14445000	14445000	14445000
10000 elementos	45430000	45430000	45430000	45430000	45430000

O número de iterações no algoritmo Shell Sort é diretamente influenciado pela escolha da sequência de lacunas utilizada para dividir a lista em subconjuntos menores. A sequência de lacunas é um aspecto crucial do Shell Sort e afeta a eficiência do algoritmo. Em cada iteração, o algoritmo reduz o espaçamento entre os elementos que estão sendo comparados e trocados. A quantidade de iterações necessárias para ordenar a lista depende da sequência de lacunas específica selecionada.