Universidade Federal de Viçosa
***Campus* Rio Paranaíba**

Laissa Rosa dos Santos- 9390

ANÁLISE DA COMPLEXIDADE DE DIFERENTES ALGORITMOS DE ORDENAÇÃO

Rio Paranaíba-MG
2025

Universidade Federal de Viçosa
***Campus* Rio Paranaíba**

Laissa Rosa dos Santos- 9390

ANÁLISE DA COMPLEXIDADE DE DIFERENTES ALGORITMOS DE ORDENAÇÃO

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmos da Universidade Federal de Viçosa- Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

RIO PARANAÍBA- MG
2025

RESUMO

Algoritmos de ordenação representam um dos pilares fundamentais da ciência da computação, sendo amplamente estudados tanto no âmbito teórico, quanto em aplicações práticas do cotidiano. Eles desempenham um papel importante na organização e manipulação de dados, permitindo que informações sejam processadas de forma mais eficiente em diferentes contextos, como em bancos de dados, sistemas de busca, análise de informações e até mesmo em tarefas simples do dia a dia da programação.

Dessa maneira, o estudo comparativo entre os diferentes métodos de ordenação existentes, torna-se indispensável para compreender não apenas o funcionamento de cada algoritmo, mas também, para identificar suas vantagens e limitações em contextos específicos. Essa análise possibilita ao programador escolher a técnica mais apropriada conforme a natureza do problema a ser resolvido, garantindo eficiência no processamento e otimizando recursos computacionais.

Palavras-chaves: Algoritmo, Ordenação, Complexidade.

ABSTRACT

Sorting algorithms are one of the fundamental pillars of computer science, being widely studied both in theoretical aspects and in practical applications of everyday life. They play an important role in the organization and manipulation of data, enabling information to be processed more efficiently across different contexts, such as in databases, search systems, information analysis, and even in simple tasks of daily programming. In this way, the comparative study of the various existing sorting methods becomes indispensable for understanding not only how each algorithm works but also for identifying their advantages and limitations in specific contexts. This analysis allows the programmer to choose the most appropriate technique according to the nature of the problem to be solved, ensuring efficiency in processing and optimizing computational resources.

Key-words: Algorithm, Sort, Complexity.

SUMÁRIO

1. INTRODUÇÃO

Algoritmos de ordenação são considerados, por muitos cientistas como “o problema fundamental da computação” [2] e desempenham papel crucial na organização e manipulação eficiente de dados em diversas aplicações computacionais. Dentre os métodos clássicos, destacam-se os algoritmos *Insertion Sort, Bubble Sort, Selection Sort* e *Shell Sort,* cada um com características operacionais distintas e adequadas a diferentes contextos de uso.

O *Insertion Sort* realiza a ordenação por meio da inserção sucessiva de elementos em suas posições corretas, simulando o processo de organização manual de cartas. Já o *Bubble Sort* baseia-se na troca repetida de pares adjacentes, fazendo com que os maiores valores “borbulhem” para o final do vetor a cada passagem. O *Selection Sort,* por sua vez, seleciona iterativamente o menor elemento da porção não ordenada do vetor e o posiciona na sequência correta, reduzindo o número de trocas em relação ao *Bubble Sort*. Por fim, o *Shell Sort* representa uma evolução do *Insertion Sort,* utilizando comparações entre elementos distantes e reduzindo gradualmente os intervalos, o que permite ganhos significativos de desempenho em vetores maiores.

Este trabalho propõe a comparação entre métodos clássicos de ordenação, a partir da análise do tempo de execução de cada algoritmo implementado em diferentes instâncias de entrada (10, 100, 1.000, 10.000, 100.000 e 1.000.000 elementos), bem como em distintas disposições (crescente, decrescente e aleatória). Além de apresentar o cálculo de complexidade para o melhor, médio e pior caso (quando possível).

As disposições dos conteúdos neste trabalho ocorrem da seguinte forma: a Seção 2 destaca o funcionamento individual dos códigos, enquanto a Seção 3 analisa a complexidade dos mesmos. A Seção 4 demonstra, graficamente, o tempo de execução de cada algoritmo e realiza uma análise comparativa entre eles. Por fim, a Seção 5 conclui este trabalho.

2. ALGORITMOS

2.1 *INSERTION SORT*

O algoritmo *Insertion Sort* (ou ordenação por inserção) funciona semelhante á maneira como as pessoas ordenam cartas de baralho com as mãos: pega-se uma carta por vez e insere-a na posição correta, e, para encontrar tal posição, é necessário comparar as cartas que já estão na mão, da direita para a esquerda (1, p. 34-35). A figura abaixo ilustra o processo:

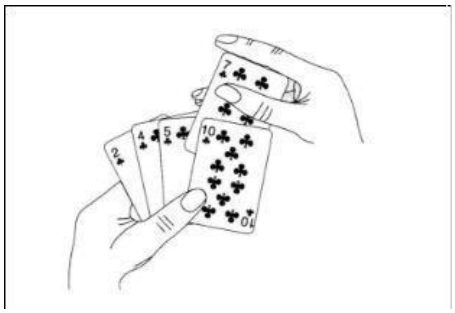


Figura 1- Ordenação por inserção
Fonte: (2, p. 12).

Esta comparação é feita a cada nova carta que se deseja inserir, mantendo- as sempre ordenadas. Analogamente, a ordenação por inserção, percorre o vetor a partir do índice 2, e movimenta uma posição a frente todos os valores que são maiores ao elemento do índice atual, até chegar na posição ideal. O passo a passo deste procedimento está ilustrado na Figura 2. Os índices dos elementos estão dispostos imediatamente abaixo de seus respectivos valores, iniciando em 1 — em contraste com a indexação padrão iniciada em 0 utilizada pelos sistemas computacionais — com o objetivo de facilitar a compreensão por parte do leitor.

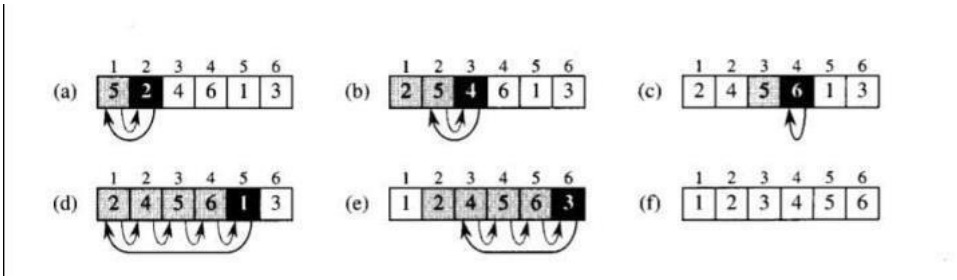


Figura 2- Funcionamento do algoritmo *Insertion Sort*
Fonte: (2, p. 12).

Os números sobre os retângulos representam os índices do vetor, e os valores armazenados aparecem dentro de cada retângulo. Ademais, as setas representam as “trocas” necessárias para a ordenação do array. De modo simplificado, para cada iteração (a)-(f), tem-se:
Passo A: comparação do valor de índice 2 do vetor com seu antecessor. Como 2 < 5, o 5 é movimentado uma “casa” á frente, trocando de posição com o 2.
Passo B: Verifica-se o valor no índice 3, como 4 < 5, eles trocam de posição.
Passo C: Compara-se o valor de índice 4[6], com o seu antecessor. Neste caso não houve troca, pois 6 > 5.
Passo D: Para ordenar o valor 1, de índice 5, é necessário movimentar uma "casa” á frente todos os seus antecessores [2, 4, 5, 6].

Passo E: O valor 3 precisa ser inserido em 2 e 4, por isso, movimenta-se os valores 4,5 e 6 para a frente.
Passo F: O vetor está com todas as posições ordenadas.

A Figura 3 apresenta o teste de mesa correspondente ao algoritmo, considerando como entrada o vetor [5, 2, 4, 6, 1, 3].

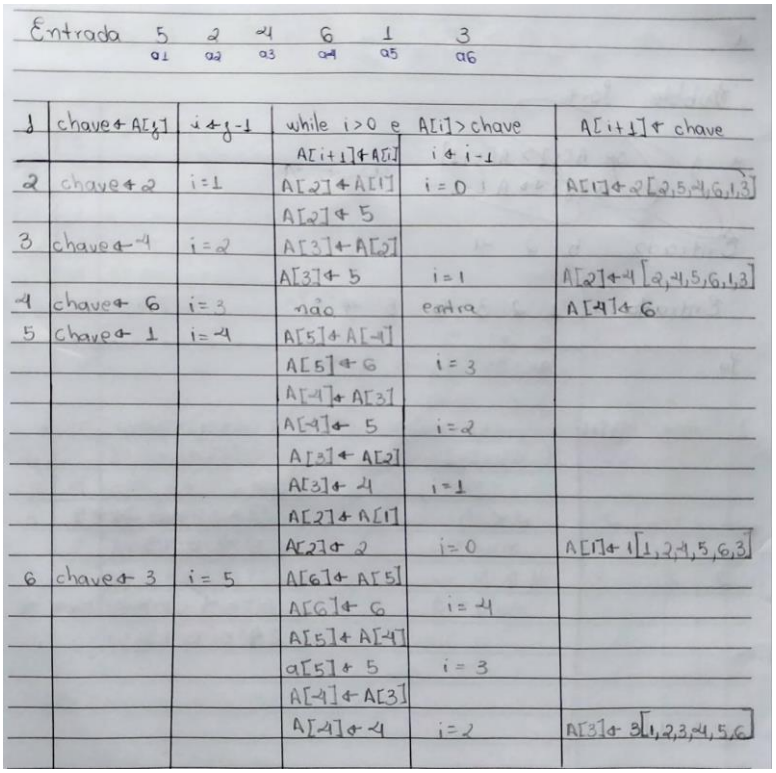


Figura 3- Teste de mesa *Insertion Sort*
Fonte: elaborado pelo autor (2025)

2.2 *BUBBLE SORT*

O algoritmo *Bubble Sort* realiza comparações entre elementos sucessivos de um vetor, trocando-os de posição sempre que estiverem fora de ordem, logo, a cada passagem, o maior elemento "sobe" para o final da lista. A figura a seguir expõem seu funcionamento:

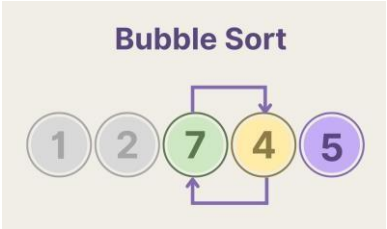


Figura 4- *Bubble Sort*
Fonte: 3

Considerando os valores da figura como um vetor em ordenação pelo algoritmo *Bubble Sort:*
Passo A) Compara se 1 > 2, como não, o vetor continua igual.
Passo B) Compara se 2 > 7, como não, o vetor continua igual.
Passo C) Compara se 7 > 4, como a condição é verdadeira, eles trocam de posição. NOVA

SEQUÊNCIA NO VETOR: [1,2,4,7,5].
Passo D) Compara se 7 > 5, como a condição é verdadeira, eles trocam de posição. VETOR ORDENADO: [1,2,3,5,7].

A figura 5 demonstra a execução do algoritmo com o teste de mesa:

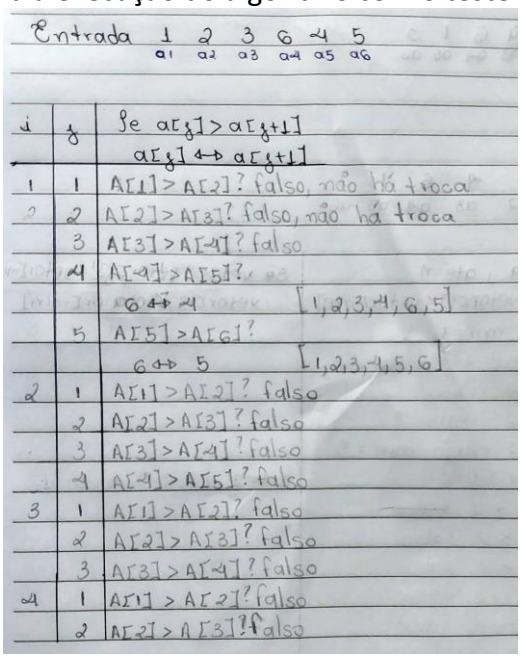


Figura 5- Teste de mesa *Bubble Sort*
Fonte: elaborado pelo autor (2025)

2.3 SELECTION SORT

O funcionamento do algoritmo *Selection Sort* baseia-se na identificação sucessiva do menor elemento da porção não ordenada do vetor. Inicialmente, o menor valor é localizado e trocado com o elemento da primeira posição. Em seguida, o processo é repetido para os n - 1 elementos restantes, depois para os n - 2, e assim sucessivamente, até que reste apenas um elemento na última posição, já ordenado por consequência. A figura 6 demonstra o fluxo da ordenação:

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 2	***A***	*R*	*D*	*E*	*N*	***O***
i = 3	*A*	**D**	***R***	*E*	*N*	*O*
i = 4	*A*	*D*	**E**	***R***	*N*	*O*
i = 5	*A*	*D*	*E*	**N**	**R**	O
i = 6	*A*	*D*	*E*	*N*	*O*	***R***

Figura 6- Funcionamento do *Selection Sort*
Fonte: (6, p. 72)

As letras em negrito indicam que houve troca entre elas. Detalhadamente:
Passo A) Na palavra “RDENA”, a letra A vem primeiro, por isso troca de posição com a letra O. Resultado: ARDENO.
Passo B) Agora, faz a análise a partir do restante da palavra: “RDENO”. A letra D vem primeiro, por isso, troca de posição com o R. Resultado: ADRENO.
Passo C) Analisando a palavra “RENO”, a letra E vem primeiro, por isso, troca de posição com

o R. Resultado: ADERNO.
Passo D) Na palavra “RNO” a letra N vem primeiro, por isso, troca de posição com o R. Resultado: ADENRO.
Passo E) No restante da palavra “RO”, o O troca de posição com o R. Resultado: ADENOR.

A figura 7 exibe o teste de mesa, com demonstração numérica para melhor compreensão do algoritmo.

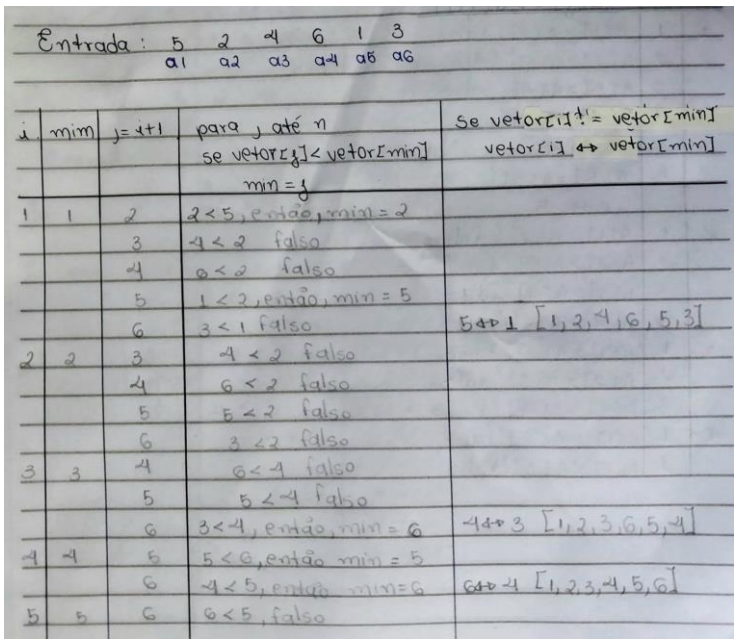


Figura 7- Teste de mesa *Selection Sort*
Fonte: elaborado pelo autor (2025)

2.4 SHELL SORT

O *Shell Sort* é um algoritmo de ordenação desenvolvido por Donald Shell em 1959, caracterizado como uma generalização do método ordenação por inserção. Seu funcionamento baseia-se na ideia de comparar e trocar elementos que estão a uma certa distância entre si, definida por uma sequência de incrementos. Inicialmente, os elementos são comparados em grandes intervalos, que vão sendo progressivamente reduzidos até que o algoritmo se comporte como uma ordenação por inserção tradicional. A imagem 8 ilustra seu fluxo de execução.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

Figura 8– Funcionamento do *Shell Sort*
Fonte: (6 , p. 76)

O h é o intervalo usado para comparar e reordenar os elementos. Simplificadamente, o passo a passo de execução consiste em:
Passo A) h = 4, logo, troco os elementos com 4 posições de distância: O troca com N e R

troca com A.
Passo B) h = 2, por isso: N troca com D. O, A e R já estão nas posições corretas, então não há trocas.
Passo C) h = 1, logo, A troca com D e N com o E. Assim, o vetor ficará ordenado.

A figura 9 demonstra a execução do algoritmo com o teste de mesa. Nesta imagem, o índice dos elementos começa em 0- tal qual a indexação padrão dos sistemas computacionais – diferente do que foi considerado anteriormente.

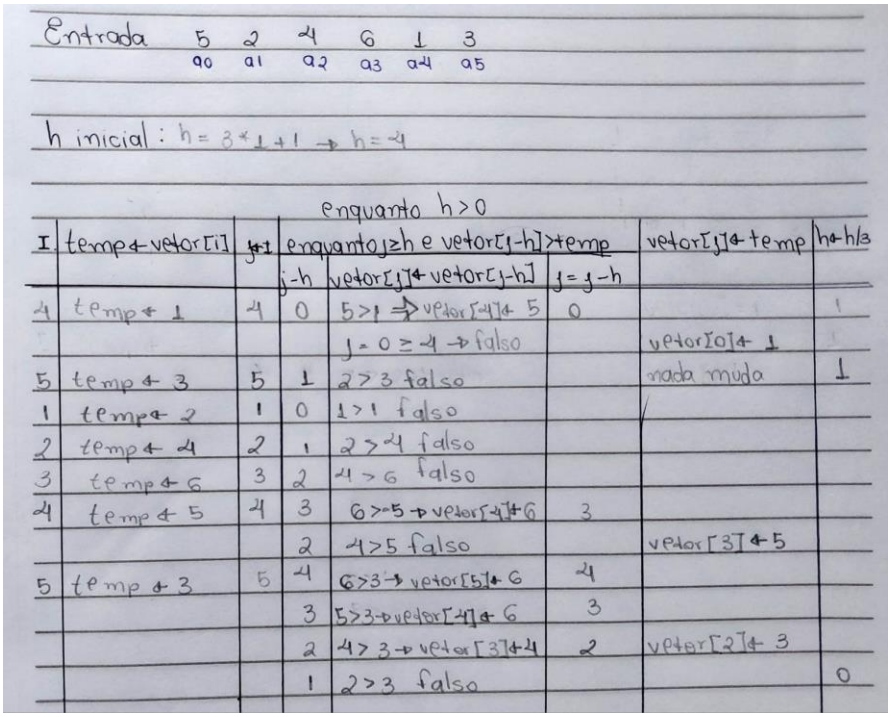


Figura 9- Teste de mesa *Shell Sort*
Fonte: elaborado pelo autor (2025)

A Figura 10 apresenta a evolução do vetor após cada movimentação. Os valores destacados correspondem às duplicações realizadas durante o processo de ordenação. Nesse método, a troca dos elementos não ocorre de maneira imediata: primeiramente, os valores são deslocados por meio de duplicação, e somente em seguida o conteúdo armazenado na variável *temp* é inserido em sua posição adequada.

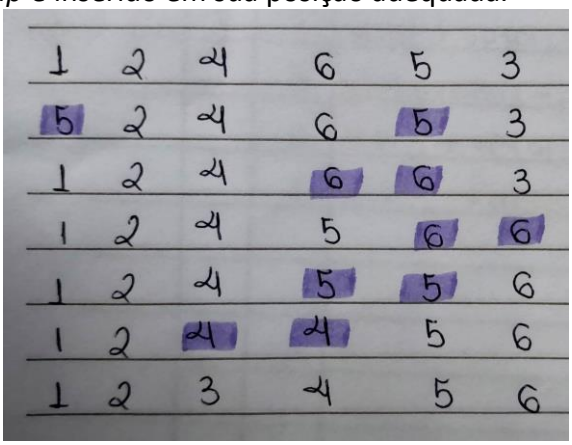


Figura 10- Estado do vetor no algoritmo *Shell Sort*
Fonte: elaborado pelo autor (2025)

3. ANÁLISE DA COMPLEXIDADE

A seguir, são examinadas as funções de custo associadas a cada algoritmo de ordenação. A análise de complexidade permite avaliar o desempenho dos métodos em diferentes cenários, destacando o comportamento no melhor, no pior e no caso médio.

3.1 INSERTION SORT

Sabe-se que, quando um laço de repetição do tipo *for* ou *while* é finalizado de maneira usual, ou seja, em decorrência do atendimento à condição imposta para sua execução, o cabeçalho é executado uma vez além do corpo do *loop.* Seja N o comprimento do vetor, tj o número de vezes em que o teste presente no laço *while* é executado para o valor correspondente de j e comentários são instruções não executáveis, considere a ilustração 11:

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
2 **do** *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *chave*	c_8	$n-1$

Figura 11- Pseudocódigo do algoritmo *Insertion Sort*
Fonte: (2, p. 12).

Observando-se o custo associado a cada instrução, bem como a quantidade de execução para cada instrução, chega-se a fórmula geral (figura 12):

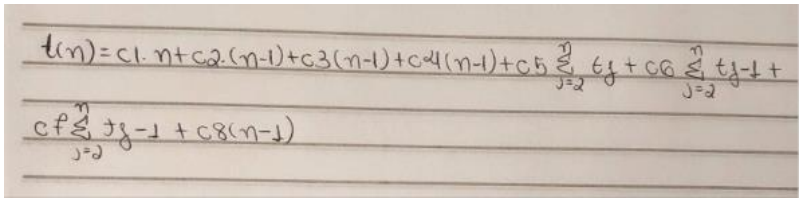


Figura 12- Fórmula Geral do Insertion Sort
Fonte: elaborado pelo autor (2025)

3.1.1 Melhor caso

O melhor caso ocorre quando o arranjo já está ordenado. Nesse sentido, a linha 5 do pseudocódigo apresentado na Figura 11 é efetuada N-1 vezes, visto que nunca é atendido a condição para percorrer o laço. Ademais, por este mesmo motivo, as linhas 6 e 7 também não serão executadas, resultando na seguinte equação abaixo (figura 13):

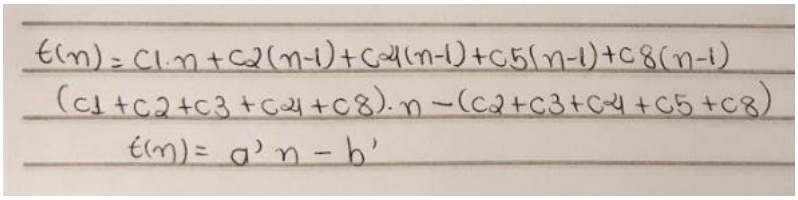


Figura 13- Cálculo do melhor caso do Insertion Sort
Fonte: elaborado pelo autor (2025)

A função custo considerada, resulta em t(n) = a'n + b' (sendo a' e b' constantes), ou seja, uma função linear para o melhor caso. Complexidade: O (n).

3.1.2 Médio caso

Ocorre quando os dados no vetor de entrada estão armazenados aleatoriamente. Sendo assim, para descobrir onde deve ser inserido um determinado elemento A, em média metade dos valores presentes no vetor serão maiores que A e a outra metade, menores. Logo, tj = j / 2. Considere a ilustração 14:

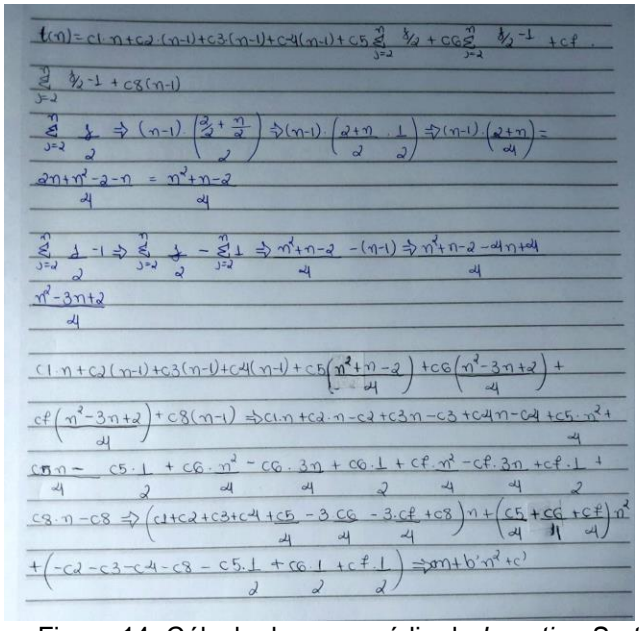


Figura 14- Cálculo do caso médio do *Insertion Sort*
Fonte: elaborado pelo autor (2025)

A função custo considerada, resulta em t(n) = a'n2 + b'n + c (sendo a', b' e c' constantes), ou seja, uma função quadrática para o caso médio. Complexidade: $O(n^2)$.

3.1.3 Pior caso

Quando o arranjo está ordenado de forma decrescente, configura-se o pior caso, uma vez que cada elemento A precisa ser comparado com todos os elementos do subarranjo já ordenado, conforme demonstrado na figura 15:

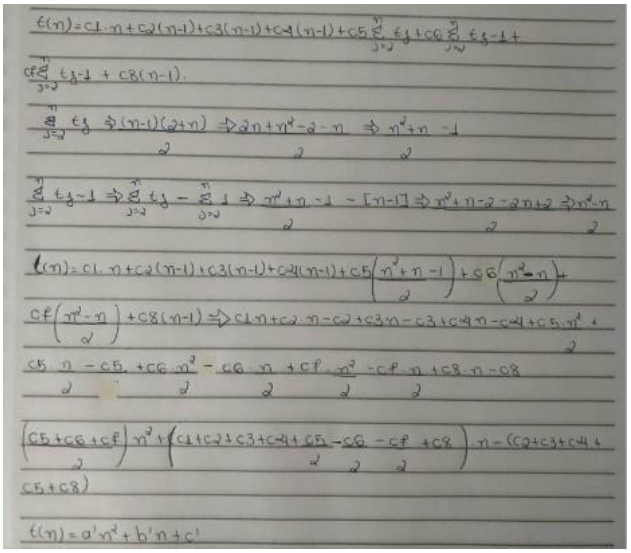


Figura 15- Cálculo do pior caso do *Insertion Sort*
Fonte: elaborado pelo autor (2025)

A função custo considerada, resulta em t(n) = a'n + b' (sendo a', b' e c' constantes), ou seja, uma função quadrática para o pior caso. Complexidade: O (n^2).

3.2 BUBBLE SORT

Para análise da complexidade do algoritmo *Bubble Sort,* observe o pseudocódigo presente na figura 16. O custo e a quantidade de vezes que uma instrução foi executada, também estão indicados em suas respectivas linhas.

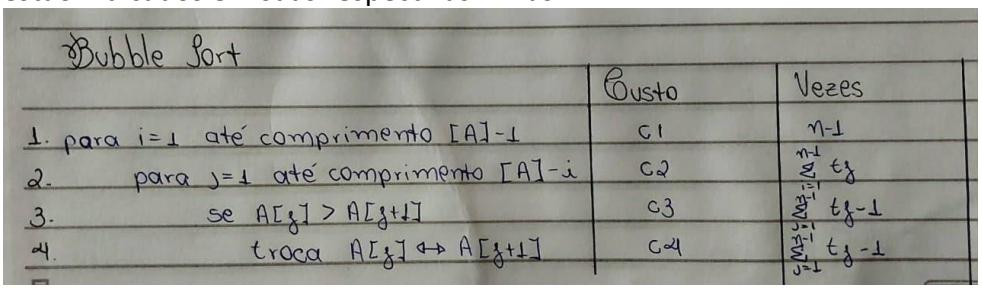


Figura 16- Pseudocódigo do algoritmo *Bubble Sort*
Fonte: elaborado pelo autor (2025)

Com base na correlação estabelecida entre o custo e a frequência de execução de determinada instrução, obtém-se a seguinte expressão geral (figura 17):

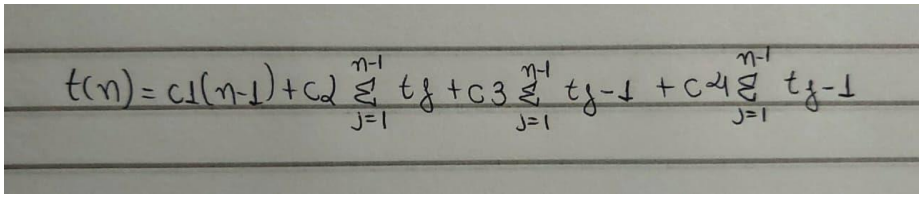


Figura 17- Fórmula geral do *Bubble sort*
Fonte: elaborado pelo autor (2025)

3.2.1 Melhor caso

O melhor caso para este algoritmo é quando os dados já estão ordenados. Nesse sentido, a linha 4 do pseudocódigo apresentado na figura 16 não será executada, ou seja, não haverá troca de posições. A imagem 18 apresenta o cálculo de complexidade associado a esse caso (os desenvolvimentos dos somatórios serão detalhados na seção 3.2.3).

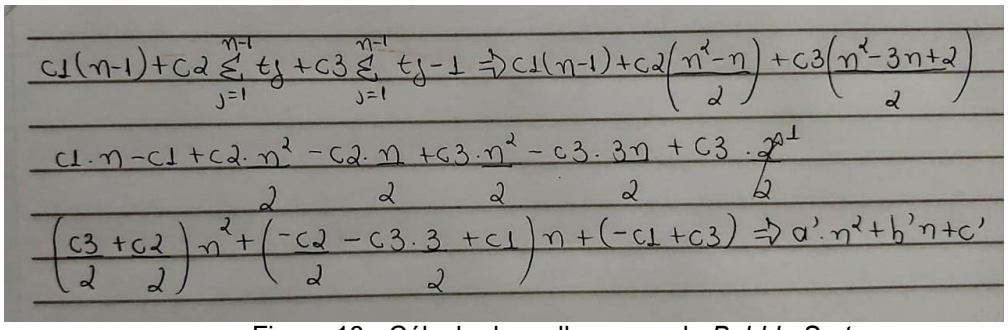


Figura 18 - Cálculo do melhor caso do *Bubble Sort*
Fonte: elaborado pelo autor (2025)

Conforme observa-se, o *Bubble Sort* possui complexidade quadrática mesmo no melhor caso, pois não reconhece quando o vetor já está ordenado. Nesse sentido, ele executa todas as comparações e apenas não realiza troca entre os elementos.

3.2.2 Médio Caso

Ocorre quando os valores estão dispostos aleatoriamente. Assim, o algoritmo não consegue prever quais elementos estão no lugar certo, então ele executa múltiplas passagens até garantir a ordenação completa. A figura 19 apresenta o cálculo da complexidade nesse caso:

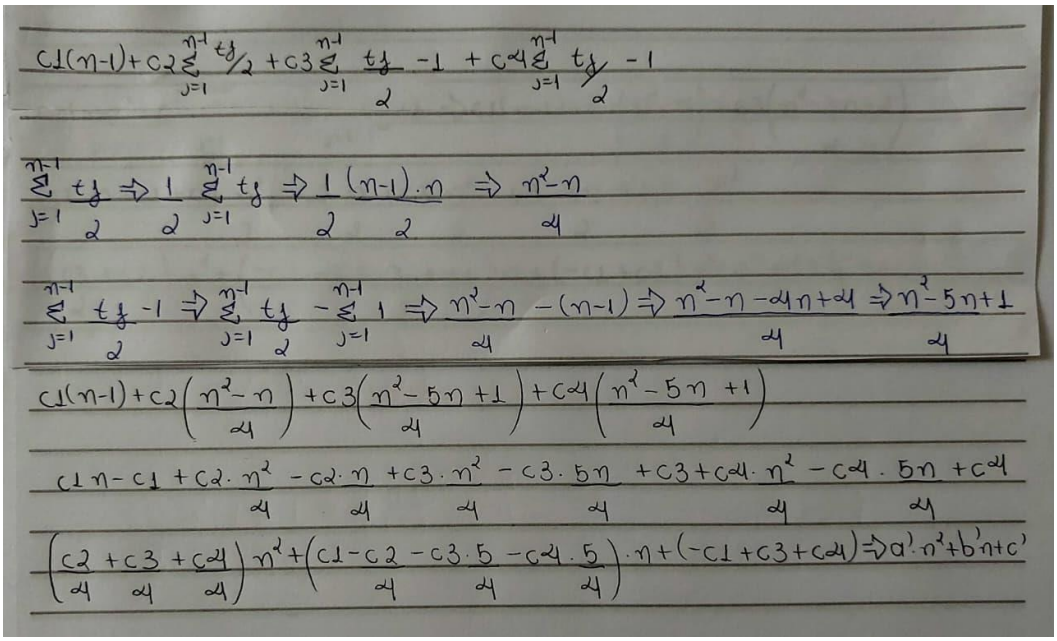


Figura 19- Cálculo do médio caso d*o Bubble Sort*
Fonte: elaborado pelo autor (2025)

Como a’, b’ e c’ são constantes, a complexidade desse algoritmo com entrada

aleatória é $O(n^2)$.

3.2.3 Pior Caso

No cenário em que os elementos estão dispostos em ordem inversa à desejada (decrescente), o algoritmo precisa realizar o máximo possível de trocas para colocar cada elemento em sua posição correta. A figura 20 demonstra o cálculo da complexidade para este caso:

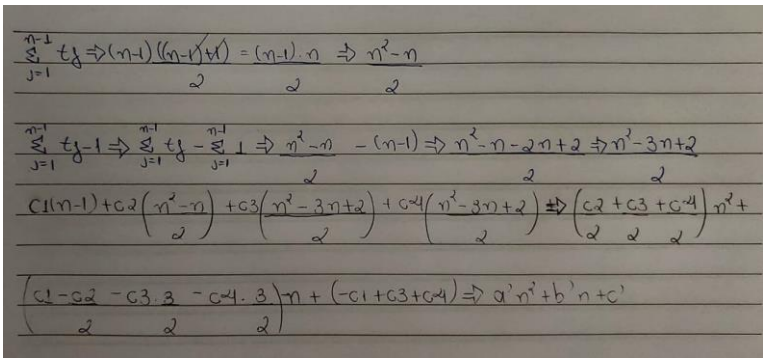
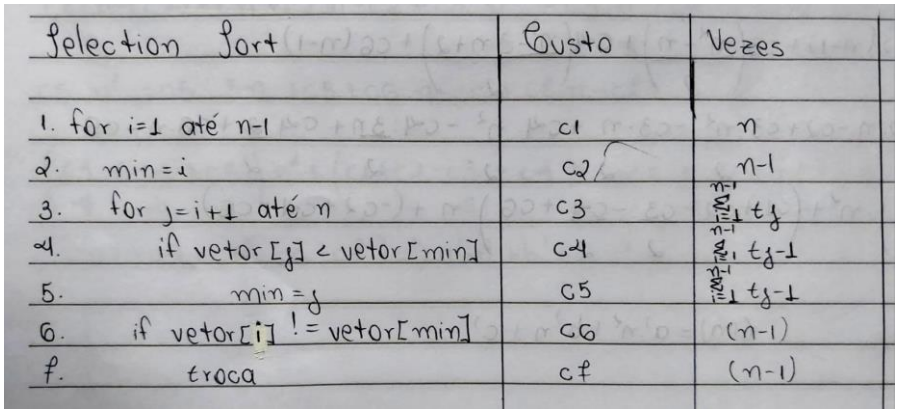


Figura 20- Cálculo do pior caso do Bubble Sort
Fonte: elaborado pelo autor

Para este caso, também se obtém complexidade quadrática, pois o algoritmo Bubble Sort realiza todas as comparações necessárias em todas as formas de entrada.

3.3 *SELECTION SORT*

A figura 21 demonstra o pseudocódigo, os custos e as vezes que cada instrução do algoritmo *Selection Sort* são executadas:



Selection Sort	Custo	Vezes
1. for i=1 até n-1	c1	n
2. min = i	c2	n-1
3. for j=i+1 até n	c3	$\sum_{i=1}^{n-1} t_j$
4. if vetor[j] < vetor[min]	c4	$\sum_{i=1}^{n-1} t_j - 1$
5. min = j	c5	$\sum_{i=1}^{n-1} t_j - 1$
6. if vetor[i] != vetor[min]	c6	(n-1)
f. troca	cf	(n-1)

Figura 21- Pseudocódigo do algoritmo Selection Sort
Fonte: elaborado pelo autor (2025)

Com base na correlação entre as colunas Custo e Vezes, obtém-se a seguinte fórmula geral (figura 22):

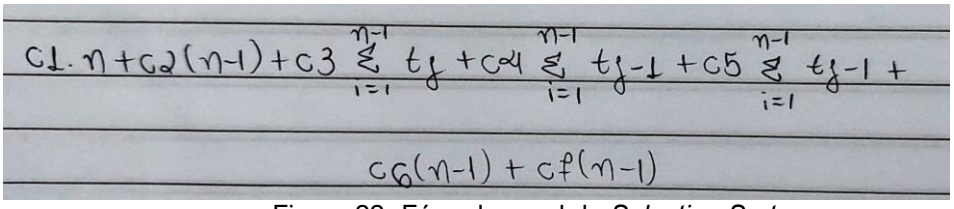


Figura 22- Fórmula geral do *Selection Sort*
Fonte: elaborado pelo autor (2025)

3.3.1 Melhor Caso

No melhor caso do algoritmo *Selection Sort,* as linhas 5 e 7 do pseudocódigo apresentado anteriormente não serão executadas. A figura 23 apresenta o cálculo da complexidade para esse caso:

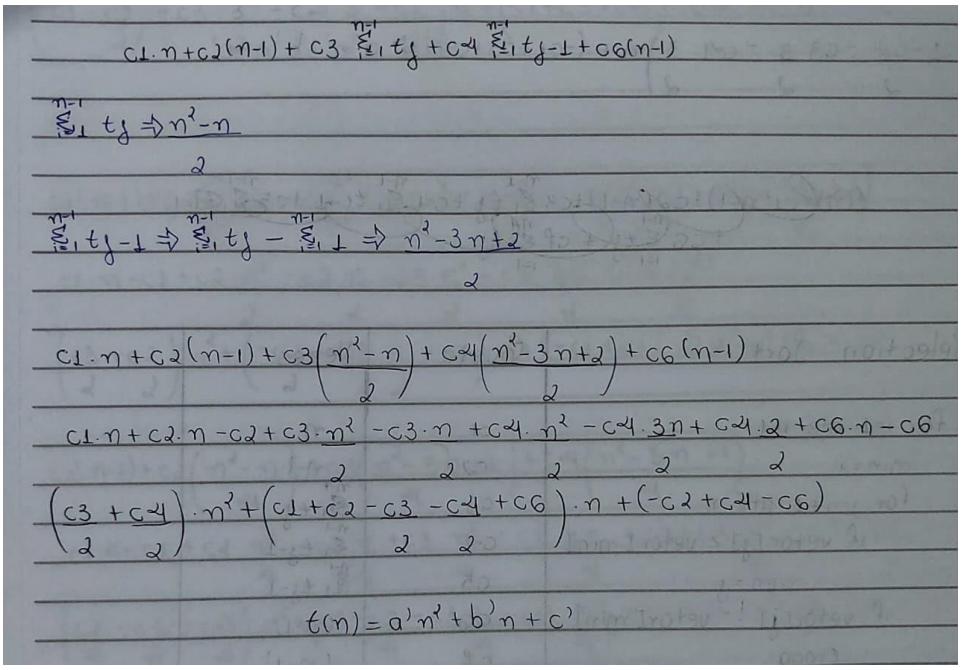


Figura 23- Cálculo do melhor caso do Selection Sort
Fonte: elaborado pelo autor (2025)

3.3.2 Médio Caso

Nesse caso os dados se encontram dispostos de forma aleatória. Por isso, tj é tj/2. A figura 24 detalha o cálculo da complexidade.

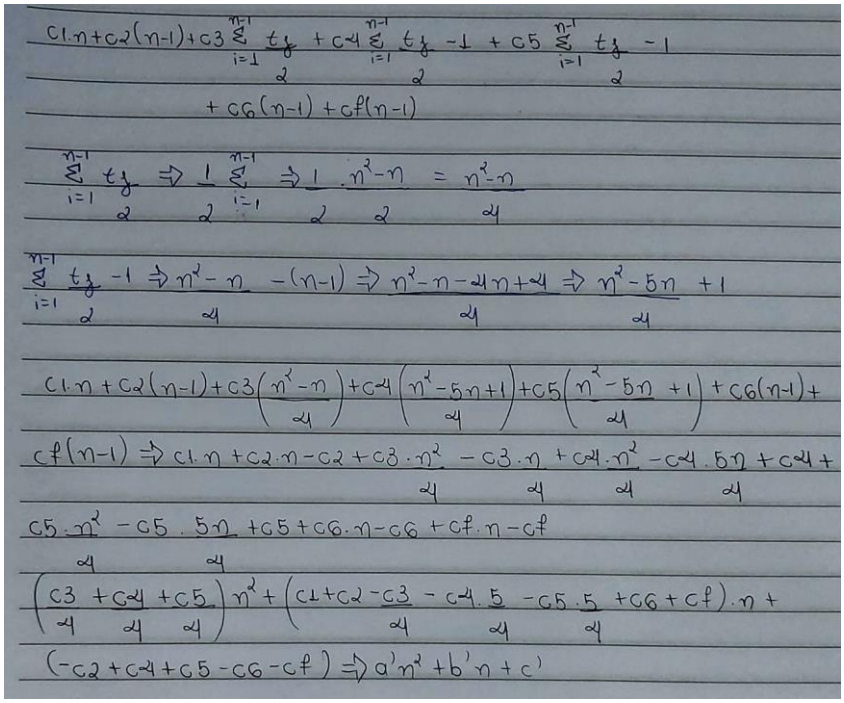


Figura 24- Cálculo do médio caso do Selection Sort
Fonte: elaborado pelo autor (2025)

3.3.3 Pior Caso

Ocorre quando o vetor de entrada esta disposto em ordem decrescentes, por isso, todas as comparações serão executas o máximo possível de vezes. A figura 25 demonstra o resultado do cálculo de complexidade.

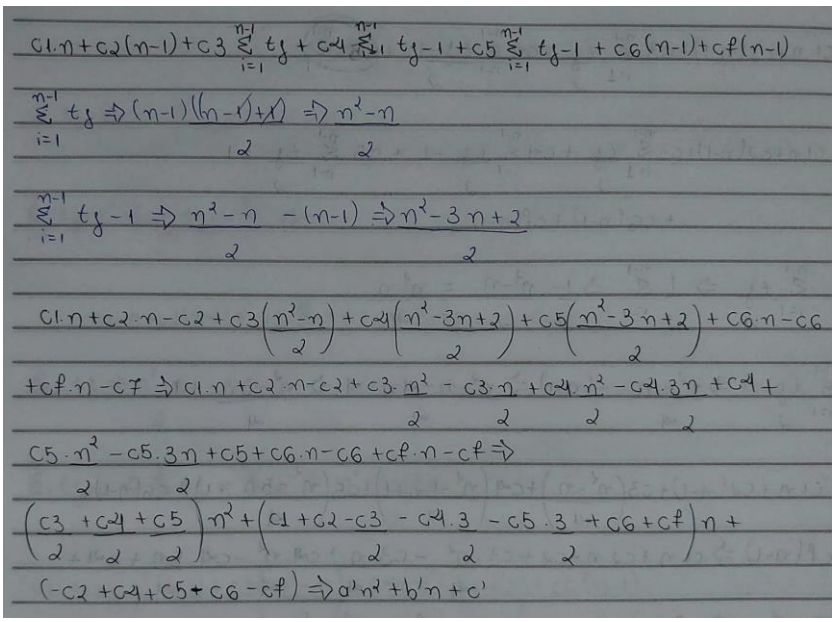


Figura 25- Cálculo do pior caso do *Selection Sort*
Fonte: elaborado pelo autor (2025)

3.4 *SHELL SORT*

A figura 26 demonstra o pseudocódigo e o custo associado a cada linha do Shell:

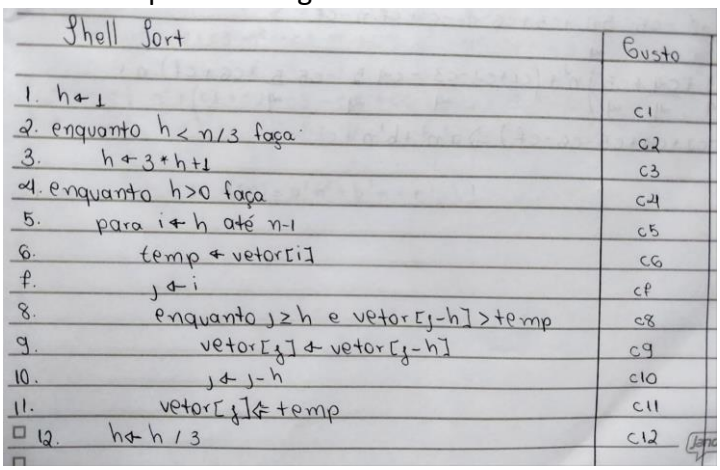


Figura 26-Pseudocódigo do algoritmo *Shell Sort*
Fonte: elaborado pelo autor (2025)

A eficiência do algoritmo *Shell Sort* ainda não é completamente compreendida, uma vez que sua análise formal permanece um desafio não solucionado [6]. A complexidade matemática envolvida na avaliação do desempenho do método é significativa, especialmente no que diz respeito à definição da sequência de incrementos utilizada. O pouco que se sabe até o momento indica que, para garantir maior eficiência, cada incremento da sequência não deve ser múltiplo do anterior. Entretanto, estima-se que, para o pior caso, a complexidade seja θ (n og^2 n) [5] ou θ (n$^{3/4}$) [4].

4. GRÁFICOS E TABELAS

Esta seção apresenta, por meio de gráficos e tabelas, os tempos de execução obtidos para cada algoritmo analisado, considerando diferentes tamanhos de entrada (10, 100, 1.000, 10.000, 100.000 e 1.000.000 elementos), bem como distintas disposições iniciais dos dados nos vetores (crescente, decrescente e aleatória).

4.1 INSERTION SORT

A Tabela 1 apresenta os tempos de execução do algoritmo *Insertion Sort* sob múltiplas entradas e em diferentes contextos.

	10	100	1000	10000	100000	1000000
Crescente	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Decrescente	0.000000	0.000000	0.005000	0.058000	5.360000	862.252000
Aleatório	0.000000	0.000000	0.000000	0.040000	2.468000	541.947000

Tabela 1- Tempos de execução do algoritmo *Insertion Sort*
Fonte: elaborado pelo autor (2025)

O Gráfico 1 permite uma análise visual dos dados da Tabela 1:

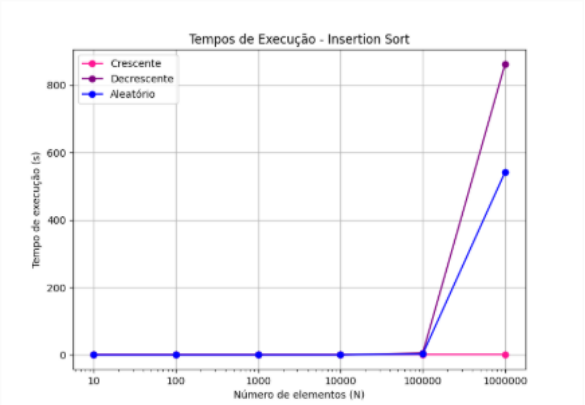


Gráfico 1- Análise dos tempos de execução do *Insertion Sort*
Fonte: elaborado pelo autor (2025)

A partir da análise dos elementos gráficos apresentados nesta seção, observa-se que, como esperado, os dados inicialmente dispostos em ordem crescente apresentam o menor tempo de execução. Dissonantemente, quando o vetor de entrada se encontra em ordem decrescente, é registrado o pior desempenho. O caso médio mantém-se relativamente constante até a instância de 1.000 elementos.

4.2 *BUBBLE SORT*

A tabela 2 apresenta os tempos de execução do algoritmo *Bubble Sort* em seus respectivos tipos e tamanhos de entradas:

	10	100	1000	10000	100000	1000000
Crescente	0.000000	0.000000	0.000000	0.082000	6.316000	469.369000
Decrescente	0.000000	0.000000	0.004000	0.151000	19.988000	1293.12900
Aleatório	0.000000	0.000000	0.011000	0.160000	33.682000	3056.86100

Tabela 2- Tempos de execução do *Bubble Sort*
Fonte: elaborado pelo autor (2025)

O gráfico abaixo (gráfico 2) permite análise comparativa entre os tempos de execução:

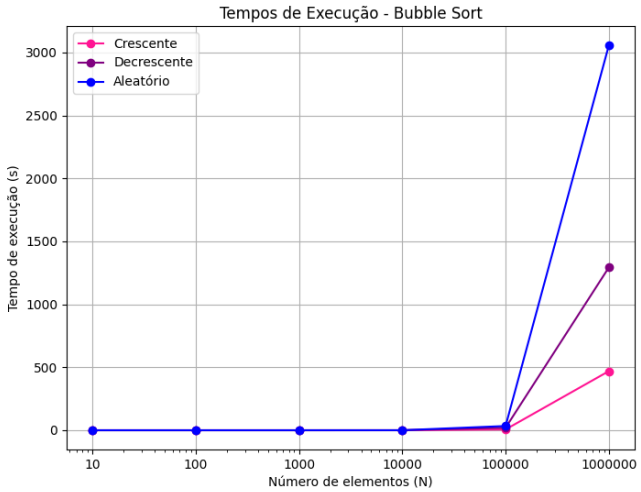


Gráfico 2- Análise dos tempos de execução do *Bubble Sort*
Fonte: elaborado pelo autor (2025)

Conforme evidenciado pelo gráfico, o tempo de execução correspondente à instância com valores aleatórios apresentou-se significativamente superior em relação aos demais tipos de entrada analisados. Ademais, a entrada ordenada de forma crescente manteve-se constante até o tamanho 1000, indicando um comportamento estável do algoritmo sob essa condição específica.

4.3 *SELECTION SORT*

A tabela 3 apresenta os tempos de execução relacionados ao algoritmo *Selection Sort*.

	10	100	1000	10000	100000	1000000
Crescente	0.000000	0.000000	0.000000	0.063000	6.233000	513.506000
Decrescente	0.000000	0.000000	0.000000	0.079000	6.311000	448.444000
Aleatório	0.000000	0.000000	0.011000	0.057000	4.078000	572.153000

Tabela 3- Tempos de execução do *Selection Sort*
Fonte: elaborado pelo autor (2025)

Visualmente, o gráfico 3 destaca o comportamento do algoritmo:

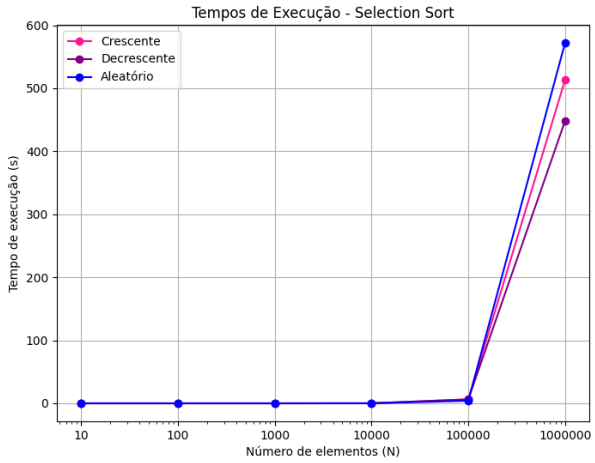


Gráfico 3- Análise dos tempos de execução do *Selection Sort*
Fonte: elaborado pelo autor (2025)

Com base na análise dos dados apresentados na tabela e na respectiva representação gráfica, observa-se que o algoritmo *Selection Sort* apresenta tempos de execução similares, independentemente da natureza da entrada fornecida.

4.4 *SHELL SORT*

Na Tabela 4, são dispostos os valores obtidos durante a execução do *Shell Sort*, permitindo uma análise quantitativa do desempenho.

	10	100	1000	10000	100000	1000000
Crescente	0.000000	0.000000	0.000000	0.0000000	0.0010000	0.04000000
Decrescente	0.000000	0.000000	0.000000	0.0000000	0.0800000	0.03600000
Aleatório	0.000000	0.000000	0.011000	0.0090000	0.0190000	0.27770000

Tabela 4- Tempos de execução do *Shell Sort*
Fonte: elaborado pelo autor (2025)

A representação gráfica torna mais clara a evolução dos tempos de execução:

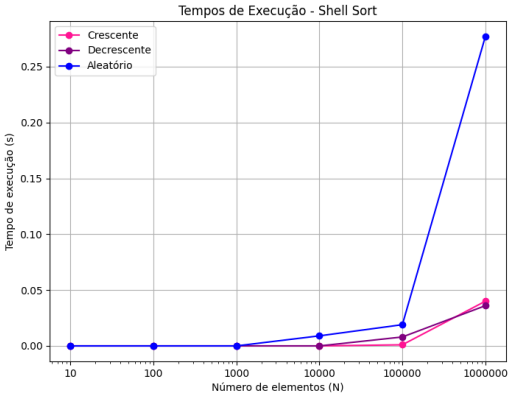


Gráfico 4- Análise dos tempos de execução do *Shell Sort*
Fonte: elaborado pelo autor (2025)

A análise dos dados apresentados nesta seção permite concluir que o algoritmo *Shell Sort* demonstra superior eficiência de ordenação em relação aos demais, apresentando tempos de execução consistentemente baixos para todos os tipos de entrada, situando-se no intervalo de menos de 1 segundo.

4.5 COMPARAÇÃO GERAL

As Tabelas 5 a 7 apresentam uma análise comparativa abrangente dos tempos de execução observados para os algoritmos de ordenação estudados. Essa avaliação foi conduzida sobre instâncias cujos tamanhos variam de 10 até 1.000.000 elementos, considerando três padrões distintos de entrada: crescente, decrescente e aleatória. O objetivo é evidenciar o desempenho relativo de cada algoritmo frente a diferentes volumes de dados e configurações iniciais, permitindo uma compreensão mais precisa de sua eficiência computacional em cenários variados. A seguir, elas são apresentadas em ordem crescente, decrescente e aleatória, respectivamente:

	10	100	1000	10000	100000	1000000
Insertion Sort	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
Bubble Sort	0.0000000	0.0000000	0.0000000	0.0820000	6.3160000	469.36900
Selection Sort	0.0000000	0.0000000	0.0000000	0.0630000	6.2330000	513.06200
Shell Sort	0.0000000	0.0000000	0.0000000	0.0000000	0.0010000	0.0400000

Tabela 5- Comparação dos tempos de execução (entrada crescente)
Fonte: elaborado pelo autor (2025)

	10	100	1000	10000	100000	1000000

Insertion Sort	0.0000000	0.0000000	0.020000	0.0600000	5.3600000	862.252000
Bubble Sort	0.0000000	0.0000000	0.0040000	0.1510000	19.988000	1293.1290
Selection Sort	0.0000000	0.0000000	0.0000000	0.0790000	6.3110000	448.44400
Shell Sort	0.0000000	0.0000000	0.0000000	0.0000000	0.0080000	0.0360000

Tabela 6- Comparação dos tempos de execução (entrada decrescente)
Fonte: elaborado pelo autor (2025)

	10	**10 0**	**10 00**	**10 000**	**10 0000**	**100 0000**
Insertion Sort	0.0000000	0.0000000	0.000000	0.0480000	2.4680000	541.947000
Bubble Sort	0.0000000	0.0000000	0.011000	0.1600000	33.682000	3056.8610
Selection Sort	0.0000000	0.0000000	0.0000000	0.0570000	4.0780000	572.15300
Shell Sort	0.0000000	0.0000000	0.0000000	0.002000	0.0190000	0.2770000

Tabela 7- Comparação dos tempos de execução (entrada aleatória)
Fonte: elaborado pelo autor (2025)

Os gráficos 5, 6 e 7 apresentados a seguir, constituem recurso visual essencial para a interpretação dos resultados experimentais, permitindo identificar padrões, contrastes e tendências que complementam a análise tabular previamente apresentada:

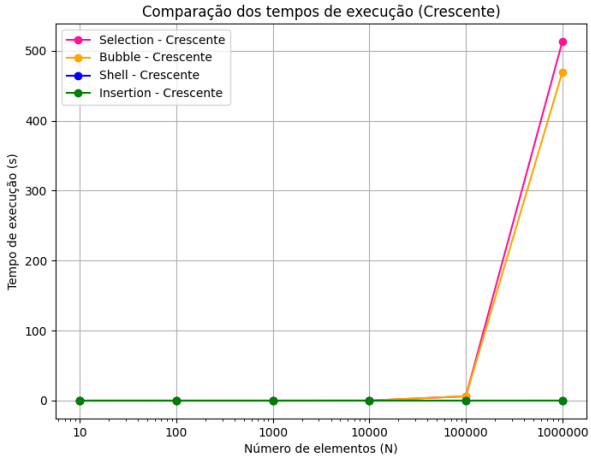


Gráfico 5- Comparação dos tempos de execução Crescente
Fonte: elaborado pelo autor (2025)

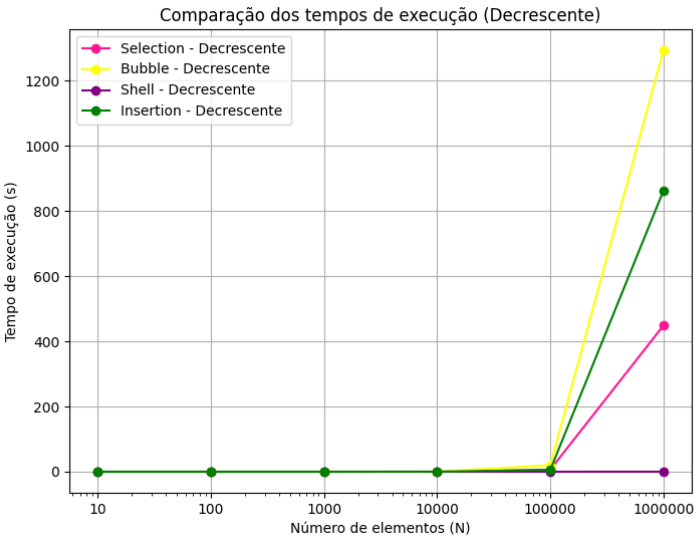


Gráfico 6- Comparação dos tempos de execução Decrescente
Fonte: elaborado pelo autor (2025)

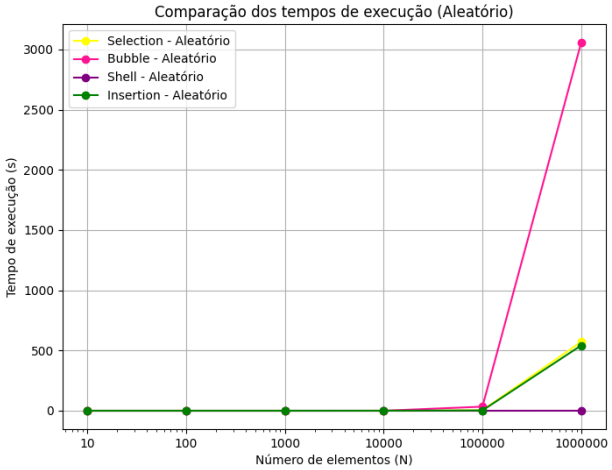


Gráfico 7- Comparação dos tempos de execução Aleatório
Fonte: elaborado pelo autor (2025)

Consoante aos dados apresentados anteriormente, o algoritmo *Shell Sort,* de forma geral, apresenta melhor desempenho que os demais analisados. Entretanto, de forma única, quando o vetor de entrada já se encontra ordenado, ele é superado pelo *Insertion Sort* (gráfico 5). O *Bubble Sort*, ao ser analisado pela ordenação de um vetor decrescente de tamanho 1000000, predomina o pior tempo de execução, ultrapassando 1200 segundos (gráfico 5). Em comparação com o segundo pior tempo de execução nas mesmas circunstâncias (*Insertion Sort)*, percebe-se uma diferença de aproximadamente 430 segundos (cerca de 7 minutos), oque torna esse algoritmo impróprio para ordenação de grandes instâncias de valores em ordem inversa ao desejado.

Ademais, a tabela 7 e o gráfico 7 apresentam os tempos de execução para cada algoritmo dada uma entrada randômica. Para estes dados, é impreciso inferir análise comparativa uniforme, visto que para execução de cada algoritmo foi gerado um conjunto numérico sem padrões, onde cada rotina de ordenação pode ou não ter sido "beneficiada" com dados parcialmente ordenados. Todavia, como é passível observar, o algoritmo *Shell* demonstra o melhor tempo de execução, enquanto os algoritmos *Bubble* e *Selection* apresentam os piores, respectivamente. Esta ocorrência se deve ao fato, de que os métodos *Bubble* e *Selection* realizam todas as comparações, independente do tipo de entrada.

4. CONCLUSÃO

Embora *Bubble Sort, Insertion Sort, Selection Sort* e *Shell Sort* compartilhem a mesma base de serem algoritmos de ordenação comparativos, seus desempenhos diferem bastante. O *Bubble* e o *Selection,* apesar de fáceis de implementar e úteis para fins didáticos, apresentam baixa eficiência em grandes conjuntos de dados devido ao elevado número de comparações e trocas – apresentando complexidade quadrática para todos os seus casos-. O *Insertion,* embora também possua complexidade quadrática no caso médio e no pior caso, mostra-se mais eficiente em listas pequenas ou quase ordenadas, sendo aplicado em algumas situações práticas. Já o *Shell Sort* representa um avanço significativo, reduzindo o número de movimentos e comparações por meio do uso de intervalos, e alcançando desempenho superior aos demais.

Em síntese, enquanto *Bubble, Insertion* e *Selection* têm maior relevância acadêmica e pedagógica, o *Shell Sort* se destaca como uma alternativa prática intermediária, oferecendo melhor equilíbrio entre simplicidade e desempenho.

REFERÊNCIAS

[1] BACKES, A. Algoritmos e estruturas de dados em C. 1. ed. Rio de Janeiro: LTC, 2023.

[2] CORMEN T. H.; LEISERSON C. E.; RIVEST R. L., CLIFFORD S. Algoritmos: teoria e prática. 2 ed. Rio de Janeiro: Elsevier Editora, 2002.

[3] FERGUSON, Natasha. Bubble Sort Algorithm. 4 nov. 2022. Imagem. Disponível em: <Bubble Sort Algorithm. Bubble sort is a sorting algorithm that… | by Natasha Ferguson | Medium > . Acesso em: 25 set. 2025.

[4] KNUTH D. E. "The Art of Computer Programming 3: Sorting and Searching". Addison-Wesley, EUA (1998).

[5] PRATT V. R. "*Shellsort and sorting networks*". Ph.D. thesis, Universidade de Stanford (1972).

[6] ZIVIANI, Nívio. Projetos de algoritmos: com implementação em Pascal e C. 4. ed. São Paulo: Pioneira, 1999.