Trabalho sobre Métodos de Ordenação - Algoritmos e Estruturas de Dados I

Arthur Batista Lage e Silva
Engenharia de Computação
CEFET-MG
Divinópolis, Minas Gerais
arthurlage2006@gmail.com

Heitor Henrique Zonho
Engenharia de Computação
CEFET-MG
Divinópolis, Minas Gerais
heitorzonho81388@gmail.com

Maria Eduarda Alves Barbosa
Engenharia de Computação
CEFET-MG
Divinópolis, Minas Gerais
mariaeduardaalvesbarbosa13@gmail.com

William Douglas Santos Branquinho Leão
Engenharia de Computação
CEFET-MG
Divinópolis, Minas Gerais
Wdsbleao@gmail.com

***Abstract*—Esse trabalho, da disciplina de Algoritmos e Estruturas de Dados I, tem o objetivo de abordar diferentes métodos de ordenação simples comumente usados para resolver problemas da computação, como Selection Sort, Gnome Sort e Insertion Sort, passando por diferenças de implementação com exemplos em diferentes linguagens de programação, custo de tempo e espaço e vantagens e desvantagens em determinados casos de uso.**

***Index Terms*—algoritmos, dados, ordenação, selection sort, gnome sort, insertion sort**

I. Introdução

Segundo os dicionários da língua portuguesa, a palavra "classificar" refere-se ao ato de reunir em classes e categorias ou dispor de forma ordenada. Entretanto, tal definição não é capaz de contemplar integralmente a importância deste conceito-chave para a ciência da computação. Neste contexto, a classificação está relacionada à organização de itens em ordem crescente ou decrescente, e influencia significativamente na rapidez e simplicidade de um algoritmo, tornando-o mais eficiente e maximizando a utilidade dos dados nele contidos. Neste artigo, serão abordados três métodos de ordenação: Selection Sort, Insertion Sort e Gnome Sort, suas vantagens, limitações e desempenho.

A história dos algoritmos de ordenação tem sua origem datada do século XIX com Charles Babbage, também conhecido como o "pai da computação", cuja Máquina Analítica (1837) introduziu conceitos essenciais de processamento sequencial e operações de leitura, escrita e manipulação de dados. O marco definitivo se deu em 1945, quando John Von Neumann, um dos maiores gênios do século XX, desenvolveu o Merge Sort, o primeiro algoritmo de ordenação executável em um computador real (o ENIAC). Esta ferramenta foi concebida para tratar dos dados militares durante a Segunda Guerra, provando que necessidades práticas impulsionam progressos teóricos. Este legado, que inclui desde o Insertion Sort do período dos cartões perfurados até o QuickSort (1960), consolidou a ordenação como pilar da computação moderna, transformando um problema crucial em um pilar da computação moderna.

O Selection Sort é um algoritmo de ordenação sintaticamente simples e vantajoso para situações que abarcam um número reduzido de elementos. Ele opera por meio de dois laços de repetição: um externo, responsável por posicionar o menor elemento da conjuntura, e um interno, que percorre os elementos restantes da sequência, atualizando o menor valor encontrado a cada iteração. Dessa forma, o algoritmo realiza uma busca comparativa entre os elementos da lista, a fim de localizar o menor (ou maior) valor e trocá-lo de posição com o elemento atual. Esse processo ocorre sucessivamente até que toda a composição esteja ordenada.

Diferentemente de algoritmos de busca e conquista, como o Merge Sort, que requerem alto poder de processamento devido à grande quantidade de gravações na memória, resultantes de seu método de ordenação recursivo. Assim, o Selection Sort realiza, no máximo, N – 1 trocas ou "*swaps*", o que acentua sua utilidade em contextos de memória limitada. Adicionalmente, essa implementação mostra-se ainda mais útil no que diz respeito à alocação de memória extra, uma vez que se caracteriza pelo uso mínimo de recursos adicionais, distanciando-se de algoritmos que exigem estruturas auxiliares durante o ordenamento.

O Insertion Sort é um algoritmo de ordenação cuja lógica se assemelha à utilizada na organização manual de cartas: insere-se um elemento por vez na posição adequada, com base nos elementos previamente ordenados à esquerda. Nesse sentido, ele inicia a ordenação a partir do segundo elemento (de índice 1), uma vez que, um array de apenas um elemento, não há o que comparar ou reordenar. Em seguida, o loop externo for itera a partir desse ponto, comparando cada elemento consecutivo com os previamente ordenados à esquerda. Ademais, sua aplicação mostra-se especialmente oportuna em cenários que envolvem valores pequenos de entrada, superando o Selection Sort em eficiência e agilidade, devido à sua análise comparativa e disposição imediata dos

componentes da sequência em detrimento de consecutivas buscas e comparações. Adicionalmente, o programa em questão se adapta melhor a sequências parcialmente ordenadas.

O Gnome Sort, também conhecido como *Stupid Sort*, baseia-se em uma lógica intuitiva de ordenação manual, anterior à sua formalização na área da Ciência da Computação. Embora seja limitado em termos de desempenho e consumo de memória, seu valor como recurso didático é reconhecido, especialmente por sua simplicidade conceitual. Assim como o Insertion Sort, o Gnome Sort mantém a porção esquerda do vetor ordenada ao longo da execução. No entanto, ao invés de inserir elementos diretamente em suas posições adequadas, ele retrocede (decrementa o índice) após cada troca para reavaliar o par anterior, garantindo a ordenação local antes de avançar novamente. Além disso, outra diferença está na ausência de laços de repetição aninhados (‘for‘ e ‘while‘), os quais são substituídos por incrementos e decrementos no índice de varredura, possibilitando uma ordenação baseada em comparações duplas seguidas por trocas repetidas. Ainda que seu uso prático seja restrito, o algoritmo oferece uma abordagem alternativa interessante para fins educacionais e experimentais.

Ao longo deste artigo, serão explorados de maneira detalhada os funcionamentos dos três códigos de ordenação citados anteriormente, as suas vantagens e desvantagens, e as suas aplicações. Tal análise fornece uma compreensão profunda sobre qual tipo de algoritmo seria mais indicado em cada cenário único de ordenação, influenciando em aspectos como o tempo de execução e memória utilizada.

II. Metodologia

A relevância dos algoritmos de ordenação na computação, no que diz respeito à melhoria da eficiência e redução do custo computacional de um código, está intrinsecamente relacionada à capacidade dessas ferramentas de simplificar a resolução de problemas. Esse fato decorre, em diversos cenários, da organização dos dados, viabilizando seu uso em algoritmos de busca, bancos de dados, estruturas de dados e métodos de divisão e conquista.

Sob essa ótica, a realização de uma análise detalhada que avalie o tamanho da entrada, seu comportamento e os requisitos de memória, mostra-se indispensável para definir qual algoritmo melhor se adequa a cada situação. Portanto, analisarei em detalhes os três algoritmos em questão (Insertion Sort, Selection Sort e Gnome Sort), destacando suas especificidades, categorizações e desempenho em diferentes contextos.

A. Selection Sort

Conforme previamente discutido, o Selection Sort baseia-se em uma busca comparativa visando encontrar o menor (ou maior) elemento da sequência. Sua execução requer a análise dos “n” constituintes da lista e realiza, no máximo, (n-1) comparações e trocas com o componente atual. Isto advém de dois laços de repetição alinhados, cuja execução independe da ordem inicial dos dados presentes no array.

Além disso, uma das principais características do Selection Sort que o distingue de algoritmos como o Insertion Sort é a uniformidade de sua complexidade temporal, ou seja, não obstante a disposição dos elementos do array (ordenado, parcialmente ordenado ou invertido), ele sempre opera com uma complexidade de tempo de $O(n^2)$. Isso se dá em decorrência da busca pelo valor mínimo (ou máximo) atual do conjunto, que ocorre a cada iteração do loop externo. Este processo exige um número fixo de comparações a cada passo, que decresce linearmente, resultando em um número de comparações que sempre coincide para uma função quadrática. Neste contexto, este processo pode ser representado a partir de uma dedução matemática que envolve a soma dos primeiros n-1 números naturais.

Adicionalmente, a propriedade do Selection Sort de realizar até n-1 trocas é especialmente benéfica em cenários nos quais as operações de escrita na memória são custosas, como em sistemas embarcados com memória limitada ou em situações onde a movimentação de grandes blocos de dados é computacionalmente cara.

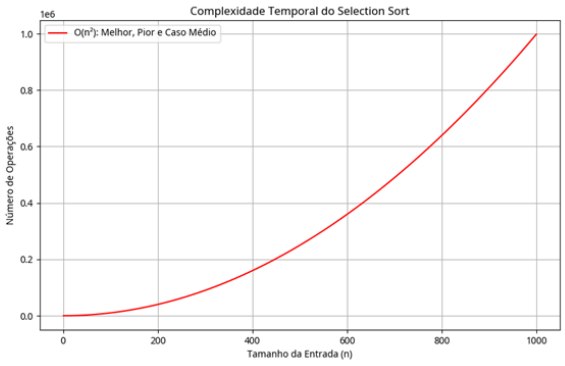
Em contrapartida, mesmo com a lista já ordenada e que haja uma preocupação em reduzir as operações de escrita para mitigar o alto custo temporal, o algoritmo demonstra pouca eficiência na otimização e desempenho. Logo, a implementação de um método de ordenação tal qual o Insertion Sort mostra-se mais eficiente e com melhorias de desempenho superiores quando comparada ao caso em estudo.

Demonstra claramente a sua natureza quadrática:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

Algorithm 1: SelectionSort

```
Input: Array arr de tamanho n
for i ← 0 to n − 2 do
    min_idx ← i;
    for j ← i + 1 to n − 1 do
        if arr[j] < arr[min_idx] then
            min_idx ← j;
    swap(arr[i], arr[min_idx]);
```



O gráfico acima demonstra a complexidade de tempo do Selection Sort.

B. Insertion Sort

A complexidade espacial de um algoritmo refere-se à quantidade de memória auxiliar que o código utiliza para executar suas operações, além do espaço inicial ocupado pelo array de entrada. Neste sentido, o Insertion Sort, tal como o Selection Sort, caracteriza-se como um programa *in-loco* (in-place), ou seja, possui complexidade espacial O(1) e não requer estruturas de dados adicionais, cujo tamanho é embasado no número de componentes a serem ordenados (n). Assim, ele opera por meio de trocas e deslocamentos efetuados diretamente no array de entrada e a única memória auxiliar necessária é utilizada para armazenar temporariamente o elemento atual, também conhecido como *key element*. Este recurso adicional independe da dimensão n da entrada e é fundamental durante o processo de comparação e inserção.

Exemplificativamente, ao deslocar um item para a direita da sequência, efetua-se uma manipulação dos demais elementos do array, não a criação de estruturas de dados auxiliares que cresçam em função de n. Portanto, a memória extra é estática, acarretando na notação O(1).

Ademais, em termos de complexidade temporal, o algoritmo apresenta variações significativas em função do tamanho da entrada de acordo com o estado inicial do array. O melhor caso possui complexidade temporal de O(n) e ocorre quando a sequência está totalmente ordenada. Neste cenário, o algoritmo ainda itera por todos os elementos da sequência, mas o loop interno (responsável por realizar comparações e deslocamentos) é executado um número mínimo de vezes. Neste cenário, o laço de repetição interno (‘while‘) é executado um número mínimo de vezes. Ele realiza um total de (n – 1) comparações, uma por iteração, sem deslocamentos, visto que os elementos à esquerda já estão ordenados e são menores que o elemento atual. A propósito de exemplo, em uma situação com valor de entrada igual a n = 4, são executadas 3 (n-1) comparações. Assim, esta ocasião pode ser representada pela expressão matemática a seguir:

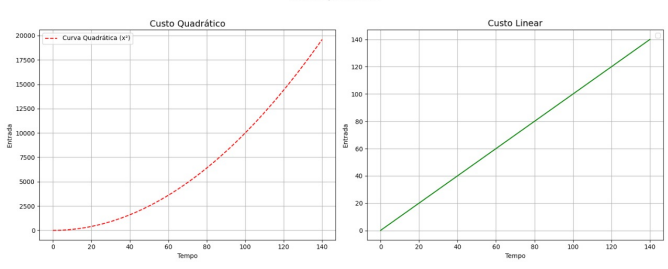
$$\sum_{i=1}^{n-1} 1 = n - 1$$

Já o pior caso ocorre quando a sequência está ordenada em ordem decrescente (inversa) e apresenta complexidade espacial de $O(n^2)$. Neste contexto, o algoritmo realiza comparações entre o elemento atual e os componentes dispostos à esquerda da sub lista. Em seguida, estes itens são movidos à direita do conjunto, maximizando o número de operações. Além disso, ao analisar a complexidade assintótica, considera-se sempre o termo de maior grau. Neste caso, o termo dominante é n^2. Portanto, as constantes e variáveis de menor grau são ignoradas, resultando na complexidade $O(n^2)$. Logo, uma entrada que contém 4 elementos realiza um total de 1 + 2 + 3 = 6 (ou 4*3/2) comparações.

O caso médio é o mais complexo de modelar, pois envolve o cálculo médio de todas as permutações possíveis de entrada. Nesse cenário, em que os elementos estão dispostos aleatoriamente, estima-se que, em média, cada elemento atual precise ser comparado com metade dos elementos já ordenados (cerca de i/2 comparações para o i-ésimo elemento), resultando em um total aproximado de n²/4 operações até ser corretamente posicionado. Isso acarreta em uma complexidade também quadrática, O(n²), apesar de possuir menos operações que o pior caso. Portanto, ambos os casos (pior e médio) apresentam crescimento quadrático.

Ao articularmos acerca de algoritmos de ordenação, há ainda outra forma de classificação essencial para uma análise bem feita destas estruturas: a estabilidade. Esta categoria diferencia os algoritmos baseando-se no comportamento da ordem relativa de elementos com valores iguais que se mantém após as ordenações. Dentre os algoritmos abordados neste artigo, o Insertion Sort e o Gnome Sort são considerados algoritmos estáveis. O Insertion Sort preserva a estabilidade ao inserir o key element (elemento atual) na posição correta na parcela ordenada do array, garantindo que elementos equivalentes não troquem de posição relativa. Analogamente, o Gnome Sort, ao comparar e deslocar elementos, usualmente mantém a ordem relativa de elementos idênticos visto que só avança quando a ordem está correta ou retorna para corrigir uma inversão, sem deslocar componentes de mesmo valor. Isso o torna interessante em situações que abarcam um número pequeno de dados e para fins educativos, embora seja o algoritmo menos eficiente dentre os descritos ao longo deste trabalho. Em oposição, o Selection Sort é um algoritmo instável. Esta instabilidade decorre do fato de que ao encontrar o menor elemento na parcela não ordenada e trocá-lo com o primeiro item da sequência ele não possui nenhum mecanismo que o impede de alterar a ordem relativa de elementos iguais que, embora não estivessem diretamente envolvidos na troca comparativa com o valor mínimo, estavam em posições relativas específicas antes da operação.

Para ilustrar as diferenças nas complexidades temporais do Insertion Sort, podemos esboçar as funções O(n) e $O(n^2)$ em um gráfico. Isso facilita a visualização do comportamento do número de operações em função do crescimento da entrada em cada cenário.



O gráfico acima demonstra a complexidade de tempo do Insertion Sort.

- **Linha Laranja**: Refere-se aos casos médio e pior $O(n^2)$, apresenta um crescimento quadrático. Para valores pequenos

de n, o comportamento das operações se equipara ao melhor caso, porém, à medida que esta entrada n cresce, a quantidade de operações cresce de forma bem mais ágil, fato que demonstra a ineficiência do algoritmo ao tratar de grandes conjuntos de dados não ordenados.

- **Linha Verde**: Refere-se ao melhor caso O(n), apresenta um crescimento linear. Para valores pequenos de entrada (n), o número de operações é baixo e aumenta proporcionalmente com n.

Para analisar a eficiência do algoritmo Insertion Sort, realizamos uma análise assintótica acerca do melhor, médio e pior caso. O pseudocódigo abaixo descreve a implementação base utilizada:

Algorithm 2: Insertion Sort

```
Input: Array arr de tamanho n
1 for i ← 1 to n − 1 do
2     atual ← array[i];
      ;              // Elemento atual a ser
       posicionado
3     j ← i − 1;
      ;      // Índice para comparar com os
       elementos ordenados
4     while j ≥ 0 e array[j] > atual do
5         array[j + 1] ← array[j];
          ;      // Desloca elementos maiores
           para a direita
6         j ← j − 1;
          ;  // Avalia o próximo elemento à
           esquerda
7     array[j + 1] ← atual;
      ;       // Insere o elemento atual na
       posição correta
```

Quando o array está ordenado, a condição ‘arr[j] ¿ key‘ dentro do loop ‘while‘ será falsa na primeira comparação para cada ‘i‘. Isso significa que o loop ‘while‘ será executado apenas uma vez (ou nenhuma vez, dependendo da implementação exata, mas para fins de análise, consideramos uma comparação para verificar a condição) para cada iteração do loop ‘for‘ externo.

C. Gnome Sort

O Gnome Sort, também conhecido como *Stupid Sort* ou *Garden Gnome Sort*, é um algoritmo de ordenação intuitivo, concebido pelo professor e pesquisador da ciência da computação Hamid Sarbazi-Azad na década de 2000. Ele fundamenta-se no princípio do gnomo de jardim que organiza seus vasos de flores: observa-se dois vasos, o que está posicionado à sua frente (elemento atual), e o seu antecessor. Em uma ocasião onde estes componentes já estejam ordenados corretamente, o gnomo avança um passo para comparar o par de itens seguinte; caso contrário, realiza-se uma troca e o gnomo recua para repetir a comparação, garantindo que a sequência fique devidamente arranjada. Consequentemente, uma vez que o gnomo se encontre na posição de índice igual a 0 no array, ele não realiza nenhuma troca, já que não há outro elemento a ser comparado, e avança. Neste cenário, o primeiro elemento é considerado ordenado. Seguindo esta lógica, se não houver vaso à frente, significa que o gnomo chegou ao fim da fila, que se encontra organizada.

Em adição, no quesito de complexidade espacial (análise de memória auxiliar utilizada), o Gnome Sort, bem como o Insertion Sort e o Selection Sort, configura-se como um algoritmo in-loco (in-place). Tal característica se justifica na ausência de estruturas de dados auxiliares durante a lógica de ordenação, operando de forma direta no array de entrada por meio de trocas e deslocamentos realizados em seus elementos constituintes. Neste contexto, a única memória auxiliar necessária armazena temporariamente um único elemento ao decorrer da operação de troca. Assim, independentemente do tamanho da entrada n, a quantidade de recursos adicionais extras utilizados permanece constante. Portanto, o espaço adicional é uma constante, o que nos leva à notação O(1).

Em relação à complexidade temporal, podemos analisar o comportamento do Gnome Sort nos baseando no seguinte pseudocódigo:

Algorithm 3: Gnome Sort

```
Input: Array arr de tamanho n
1 index ← 0;
2 while index < n do
3     if index == 0 then
4         index ← index + 1;
          ;      // Primeiro elemento, avança
5     else
6         if arr[index] ≥ arr[index − 1] then
7             index ← index + 1;
              ;        // Elemento na posição
               correta, avança
8         else
9             swap(arr[index], arr[index − 1]);
              ;             // Troca elementos
10            index ← index − 1;
              ;       // Recua para verificar
               novamente
```

O melhor caso ocorre quando o array de entrada se encontra completamente ordenado. Nesta situação, o algoritmo percorre o array uma vez, sempre avançando, uma vez que a condição $(arr[index] >= arr[index - 1])$ será sempre verdadeira (ou ‘index == 0‘ será verdadeiro no início). Desse modo, o bloco ‘else‘ (que contém o ‘*swap*‘ e o $(index = index - 1)$ nunca é executado e o algoritmo apenas incrementa o índice em cada iteração do loop ‘while‘.

Em um cenário onde o loop ‘while‘ externo é executado n vezes (índice que vai de 0 a n – 1), a cada iteração é efetuada uma verificação se o índice for igual a zero $(if\ index == 0)$ e outra em $(if\ arr[index] >= arr[index-1])$, totalizando duas

operações. O número total destas operações é proporcional ao valor n de entrada de modo que o algoritmo faça um único passo através do array, resultando em uma quantidade constante de operações para cada elemento. Assim, para $index = 0$, há uma verificação e 1 incremento e para $index = 1$, há uma comparação e um incremento, resultando em um número total de operações de aproximadamente 2n. Este raciocínio pode ser obtido a partir da expressão matemática a seguir:

$$C_{melhor}(n) \approx \sum_{k=1}^{n} c = c \cdot n$$

Onde c é uma constante que representa o número de operações por iteração. Portanto, a complexidade temporal no melhor caso é O(n).

Além disso, paralelamente ao Insertion Sort, o algoritmo em enfoque possui um pior caso em situações onde o array se encontra ordenado de forma decrescente (inversa). Neste contexto, a cada troca realizada, o índice é decrementado. Isto acarreta na reavaliação dos elementos revisados, fato que resulta em um alto número de comparações e trocas, uma vez que o algoritmo precisa se deslocar para trás e para frente do array. Assim, considerando um array ordenado em ordem decrescente para cada elemento $(arr[index] < arr[index–1])$, uma troca ocorre e o índice é reduzido. Desse modo, o algoritmo pode percorrer a parte previamente arranjada do array diversas vezes para cada elemento de seu posicionamento correto, podendo realizar um número de operações proporcional ao número de elementos já processados.

O total de operações pode ser estimado pelo somatório:

$$\sum_{i=1}^{n-1} i$$

Esta soma é dada pela fórmula $\frac{(n-1)n}{2}$.

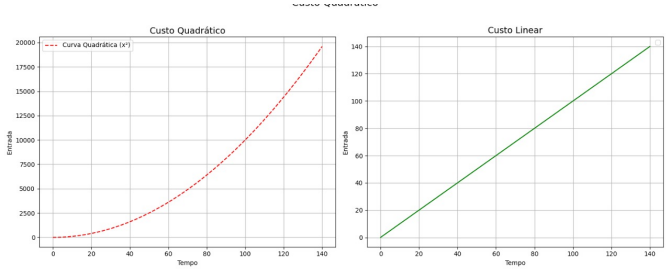
$$C_{pior}(n) = \frac{n^2 - n}{2}$$

Segundo Segundo Cormen et al. (2009), ao analisar a notação *Big O*, consideramos o termo de maior grau. Neste caso, é n^2. As constantes e termos de menor grau são desconsiderados. Logo, a complexidade temporal no pior caso é $O(n^2)$.

No caso médio do Gnome Sort, analogamente ao Insertion Sort, também é $O(n^2)$. Esta análise é complexa por abarcar probabilidades e permutações. A lógica geral equivale, em média, o algoritmo efetuará um número de comparações e trocas proporcional a n^2. Tal ocorrência se dá pelo fato de que para cada elemento, o algoritmo pode mover-se para trás e para frente no array várias vezes até encontrar a posição correta do componente.

Para enfatizar as diferenças nas complexidades temporais do Gnome Sort, podemos plotar as funções O(n) e $O(n^2)$ em um gráfico que demonstra o comportamento do número de operações em função do tamanho n da entrada em cada cenário

Como pode ser observado no gráfico:



O gráfico acima demonstra a complexidade de tempo do Gnome Sort.

A linha verde (Melhor Caso: $O(n)$) mostra um crescimento linear. Para pequenos valores de n, o número de operações é baixo e aumenta proporcionalmente com n. Já a linha laranja (Pior e Caso Médio: $O(n^2)$) exibe um crescimento quadrático. Para pequenos valores de n, o número de operações é semelhante ao melhor caso, mas à medida que n aumenta, o número de operações cresce muito mais rapidamente, fato que salienta a ineficiência do algoritmo para grandes conjuntos de dados não ordenados.

III. Aplicações

Conhecemos em detalhes três métodos de ordenação, seus comportamentos, desempenho, vantagens e desvantagens. Agora, irei discorrer acerca de suas utilidades práticas de forma aprofundada, capacitando uma compreensão mais ampla de cada algoritmo e sua importância para a computação e o desenvolvimento de soluções eficientes.

Assim como previamente discutido, o Selection Sort executa no máximo n-1 trocas. Esta característica é especialmente útil em sistemas de memória embarcada, ou seja, em sistemas computacionais especializados desenvolvidos para realizar uma ou mais funções de um conjunto maior, seja ele eletrônico ou mecânico. Estes agrupamentos apresentam memória de escrita lenta ou computacionalmente cara, a implementação deste método é altamente eficaz. Além disso, em contextos que abarcam pequenos conjuntos de dados, a diferença de desempenho entre algoritmos quadráticos é irrelevante, fato que torna o seu uso mais interessante devido à sua simplicidade.

O Insertion Sort apresenta diversos casos de uso, entre eles está sua aplicação em cenários que envolvem listas com um número reduzido de elementos, onde o *overhead* de algoritmos mais complexos supera a diferença de desempenho. Este fenômeno refere-se aos custos adicionais que um algoritmo ou sistema incide além de sua principal função para solucionar um problema. Tal ocorrência pode ser observada no tempo de inicialização, alocação de memória auxiliar ou operações de gerenciamento que não contribuem diretamente na resolução de impasses. Sob esta ótica, esse overhead pode fazer com que algoritmos que, embora notadamente eficientes na teoria (com melhor complexidade assintótica para grandes entradas), sejam efetivamente mais demorados do que algoritmos mais simples, em decorrência do consumo adicional envolvido em sua configuração e execução. Ademais, assim como precitado ao longo do artigo, seu desempenho aproxima-se de O(n) em cenários onde o conjunto está parcialmente ou quase

completamente ordenado, devido aos poucos deslocamentos necessários.

No entanto, seu vasto uso na Ciência da Computação não se limita a estas situações supracitadas descritas, sendo amplamente utilizado em algoritmos de ordenação mais avançados, como Timsort (utilizado em Python e Java). Este algoritmo híbrido combina o Merge Sort e o Insertion Sort, por meio da divisão da lista em pequenas partes (*runs*), ordenando essas parcelas menores usando o Insertion Sort, sucedida do agrupamento das sub listas ordenadas a partir de uma variação do Merge Sort. Esta lógica resulta em um algoritmo muito mais eficiente na prática, otimizando o desempenho geral para diferentes tamanhos e características de dados.

O Gnome Sort, apesar de não ser tão eficiente no que diz respeito à complexidade assintótica, destaca-se por sua simplicidade de implementação e compreensão. Isso o torna particularmente útil em contextos didáticos, servindo como uma excelente ferramenta para instruir os conceitos de ordenação. Além disso, para conjuntos de dados menores ou majoritariamente ordenados, seu desempenho pode ser eficaz e sua natureza intuitiva é recomendada em ocasiões onde a clareza do código importa mais que a otimização em si.

IV. Resultados

Os resultados adquiridos demonstram variações significativas no desempenho dos algoritmos Selection Sort, Insertion Sort e Gnome Sort quando implementados em diferentes estruturas de dados (Lista, Fila e Pilha) e em linguagens diversas (Python, Java, C, C++ e Ruby). Esta análise possibilita maior compreensão acerca do comportamento desses métodos de ordenação em diferentes cenários, salientando suas especificidades. Esse estudo foi realizado por meio de tempo de execução (em ms) para entradas de tamanhos distintos (n = 100, n = 1.000, n = 10.000, n = 100.000).

A análise dos gráficos do tempo de execução dos algoritmos Selection Sort, Insertion Sort e Gnome Sort, implementados em linguagem C, permite visualizar como o desempenho de cada método se comporta à medida que o tamanho da entrada cresce, evidenciando suas limitações e pontos fortes em diferentes escalas:

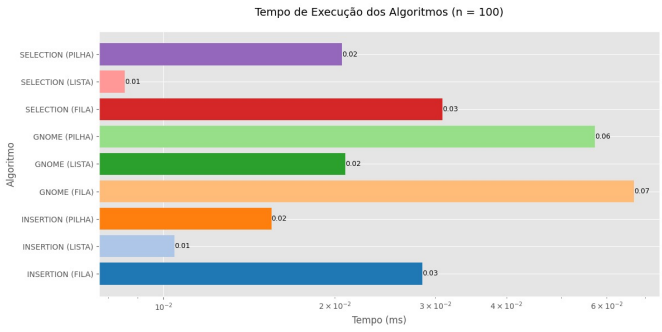


Gráfico dos tempos para cem elementos

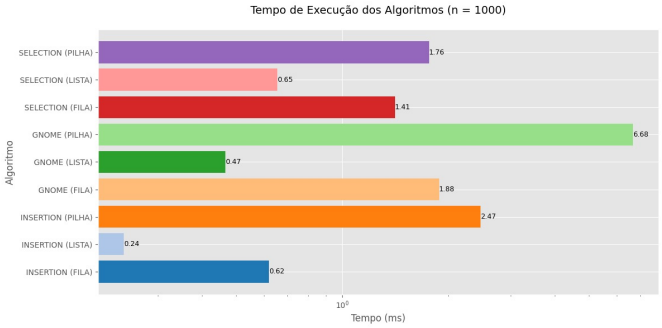


Gráfico dos tempos para mil elementos

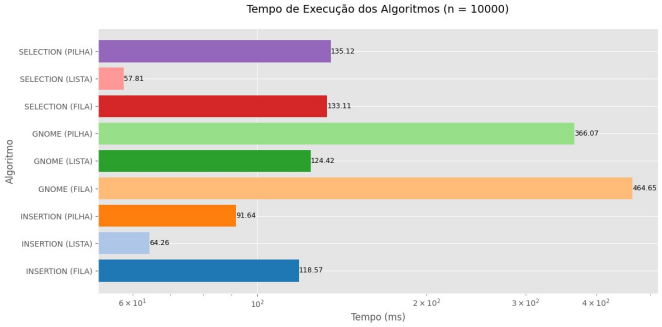


Gráfico dos tempos para dez mil elementos

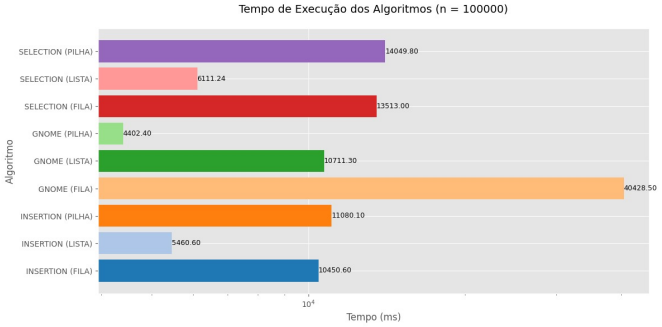


Gráfico dos tempos para cem mil elementos

A. Análise dos resultados do Gnome Sort

No gráfico com valor de entrada igual a n = 100.000, o Insertion Sort apresentou o menor tempo de execução em listas, equivalente a 5460.60 ms, seguido do Selection Sort com um total de 6111.24 ms. Além disso, o Gnome Sort apresentou a menor eficiência nessa estrutura (tempo de execução de 10711.30 ms). Essa ocorrência é justificada pelo caráter do algoritmo, que envolve movimentações e trocas consecutivas de elementos que nem sempre contribuem efetivamente para a ordenação.

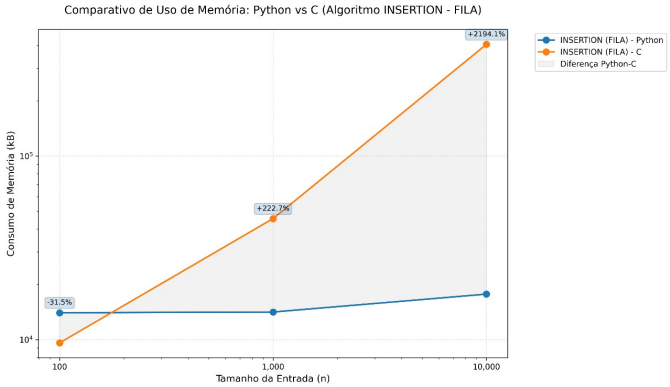
Ademais, no gráfico de entrada n = 10.000, o Gnome Sort na pilha (124.42 ms) obteve menor tempo de execução em comparação à sua implementação com fila (464.64 ms). Isso ocorre devido à característica intrínseca das pilhas (*LIFO* - Last In, First Out) que simplificam as operações de retrocesso recorrentes do Gnome Sort. Desse modo, o algoritmo obtém uma vantagem local, reduzindo o custo de acesso a elementos anteriores durante as checagens. Entretanto, essa vantagem

não é suficiente para compensar suas verificações e trocas redundantes, conservando sua complexidade média de $O(n^2)$, fato que o torna menos eficiente que algoritmos como o Insertion Sort (91.64 ms).

Além disso, o Gnome Sort obteve um pico 40 vezes mais lento que o Insertion Sort em filas. Isso se justifica, sobretudo, pela necessidade do algoritmo de efetuar comparações e trocas com os elementos adjacentes em conjunto da exigência de reconstrução da fila do zero a cada troca, fato que acrescenta operações O(n) por comparação. Assim, o custo computacional migra de um valor quadrático para uma escala cúbica (complexidade de $O(n^3)$).

B. Análise dos resultados do Insertion Sort

Segundo Tanenbaum e Austin (2013), execuções de programas em C são mais ágeis que em linguagens como Python devido ao seu aspecto compacto, ou seja, a presença de um compilador que traduz a linguagem de máquina (binário) ao passo que otimiza o programa, dispensando a necessidade de traduções em tempo real. No entanto, os resultados das testagens revelaram uma contradição neste aspecto: O Insertion Sort em Python (3.530,63 ms) foi mais rápido que em C (4.949,64 ms) para listas padrões com 100.000 elementos. Esta discrepância pode ser atribuída às otimizações internas das listas dinâmicas em Python, que são implementadas em C (*CPython*) e operam por meio de estratégias de alocação de memória inteligentes como o crescimento exponencial da capacidade, capaz de reduzir o número de alocações e cópias frequentes durante a inserção e operações em bloco para deslocamento, que se vale do uso de ponteiros, tornando as inserções e deslocamentos mais ágeis em decorrência destas movimentações a partir da manipulação de endereços de memória em detrimento dos dados em si. Paralelamente, a implementação em C, ainda que utilize vetores, pode sofrer com *overheads* de gerenciamento manual de memória como realocações recorrentes e cópias explícitas (como as realizadas via memmove). Ademais, a lista tradicional do Python se beneficia do uso de ponteiros para objetos, o que reduz o custo do deslocamento dos dados. Este cenário demonstra como estruturas de dados altamente otimizadas, mesmo em linguagens menos compactas, são capazes de superar implementações manuais em linguagens de baixo nível, quando abarcam operações complexas de deslocamento e inserção. Logo, a escolha de qual estrutura de dados utilizar e sua implementação interna são tão críticas quanto a seleção da linguagem para o desempenho de algoritmos frágeis a operações de memória, tal como o Insertion Sort. .



O gráfico acima demonstra a comparação da memória utilizada pelo C e pelo Python para o método insertion sort

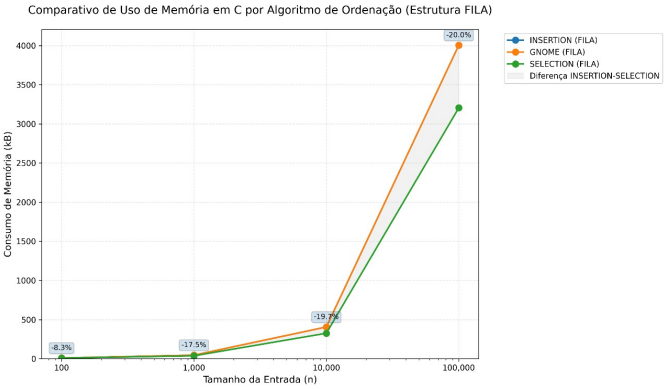
C. Análise dos resultados do Selection Sort

O Selection Sort demonstrou comportamento constante em sua complexidade $O(n^2)$ em todas as linguagens testadas, mas com variações expressivas devido aos métodos de implementação. Em linguagens compiladas, assim como C e C++, o algoritmo apresentou melhor desempenho em estruturas com ponteiros para grandes conjuntos de dados, por consequência direta da ausência de realocações de memória, o que mitigou o impacto sobre seu custo quadrático. Sob outra perspectiva, implementações que abarcam vetores foram mais rápidas apenas para entradas pequenas $(n < 10.000)$, sendo favorecido pela localidade de referência e acesso O(1). Esta vantagem decorre do dimensionamento das posições via índices, sem demandar navegações por encadeamento de ponteiros. Ademais, em Java, essa diferença foi ainda mais acentuada em virtude dos benefícios iniciais da aplicação de vetores, mas tornaram-se inviáveis para $n > 10.000$ graças ao custo das trocas manuais, enquanto execuções com ponteiros mantiveram desempenho estável.

Por outro lado, em linguagens interpretadas, assim como Python e Ruby, o Selection Sort revelou limitações significativas. Em Python, foram até cerca de 4 vezes mais lentos que em C para uma entrada de $n = 10.000$ (4.576 em filas, 1.190 em listas), e testes com $n > 10.000$ se tornaram impraticáveis com um tempo de execução superior a 30 minutos. O Ruby, por sua vez, apresentou um desempenho constante e baixo em todas as estruturas de dados, com tempos em torno de 3.890 ms para $n = 10.000$, evidenciando o *overhead* da interpretação dinâmica. Logo, os resultados adquiridos destacam que, embora o Selection Sort seja simples e possua baixo uso de memória $(O(1))$, sua eficiência na prática depende fortemente da linguagem e da estrutura de dados escolhida.

Outro aspecto relevante observado foi a superioridade das filas em C++ para valores de $n > 1.000$, se opondo à tendência geral de serem menos eficientes. Isso ocorre devido à implementação inata da fila (possivelmente alocação estática ou circular) minimizou operações de realocação, enquanto listas e pilhas sofreram com custos indiretos de ponteiros. Em contrapartida, em Java e Python, as filas apresentaram maior tempo de execução, especialmente devido à necessidade

de reconstrução completa após as trocas. Essas discrepâncias, salientam que, mesmo para algoritmos considerados pouco eficientes como o Selection Sort, otimizações de baixo nível e a escolha da estrutura podem alterar radicalmente o desempenho.



O gráfico demonstra uma comparação geral entre os 3 métodos de ordenação

V. Conclusão

Ao decorrer deste artigo, foram realizados estudos aprofundados acerca de três algoritmos de ordenação fundamentais, sendo estes o Selection, Insertion e Gnome Sort, analisando suas características operacionais, complexidades de tempo e espaço, além de suas aplicações práticas. O artigo foi fundamentado com uma breve contextualização histórica, até a discussão de suas vantagens e limitações e a influência das estruturas de dados no desempenho destes métodos. Adicionalmente, o estudo demonstrou que a escolha do algoritmo ideal é intrinsecamente ligada ao contexto específico da aplicação, ao tamanho e disposição inicial das conjunturas, e aos recursos computacionais disponíveis.

Além disso, pode-se observar que, apesar de todos os três algoritmos apresentarem uma complexidade temporal quadrática no pior caso, suas singularidades de implementação acarretam em comportamentos distintos. Por exemplo, o Selection Sort, com seu número mínimo de trocas, revela-se eficiente em sistemas com memória lenta ou cara, como sistemas embarcados. O Insertion Sort, por sua vez, diferencia-se pela eficiência em conjuntos de dados pequenos ou quase ordenados, e sua estabilidade o torna preferível em cenários nos quais a ordem relativa de elementos iguais deve ser mantida. Ademais, sua integração em algoritmos híbridos como o Timsort salienta sua relevância em otimizações de larga escala.

O Gnome Sort, embora conceitualmente simples e de utilidade didática, mostrou-se consistentemente menos eficiente entre os demais algoritmos, devido à sua abordagem (vai e volta) sucedida pelo seu maior custo por operação de troca. A análise de desempenho em diferentes estruturas de dados, como filas, pilhas e listas, reforçou que as restrições de acesso inerentes a essas estruturas podem exacerbar as ineficiências dos algoritmos, realçando a importância da escolha da estrutura em conjunto ao algoritmo de ordenação.

Em síntese, o entendimento aprofundado das propriedades de cada algoritmo de ordenação é crucial para o desenvolvimento de soluções computacionais eficientes. Não há, de fato, um algoritmo "melhor" absoluto, uma vez que a eficácia fundamenta-se na capacidade de discernir qual ferramenta mais se adequa e aperfeiçoa o código, com base nos seus requisitos e limitações de cada problema. Assim, este estudo reitera que a ordenação ainda é um campo dinâmico e essencial para a ciência da computação, promovendo a inovação e otimização contínua em diversos sistemas.

References

[1] IKIPEDIA. Selection sort. [s.d.]. Disponível em: https://en.wikipedia.org/wiki/Selection_sort. Acesso em: 22 maio 2025.

[2] HARGAVA, Aditya. Grokking algorithms: an illustrated guide for programmers and other curious people. 1. ed. Nova York: Manning, 2016.

[3] EEKSFORGEEKS. Selection Sort. GeeksforGeeks, [s.d.]. Disponível em: https://www.geeksforgeeks.org/selection-sort-algorithm-2/. Acesso em: 22 maio 2025.

[4] ICKGRUNE.COM. *Gnome Sort*. Disponível em: ¡https://www.dickgrune.com/Programs/gnomesort.html¿. Acesso em: 10 jun. 2025. GEEKSFORGEEKS.

[5] ILVA, A., ZONHO, H., BARBOSA, M., LEÃO, W. (n.d.). Implementação dos códigos do seminário de AEDS 1. Disponível em: ¡https://github.com/arthur-lage/seminario-aeds1/¿. Acesso em: 28 jun. 2025.

[6] nome Sort – A Stupid One. Disponível em: ¡https://www.geeksforgeeks.org/dsa/gnome-sort-a-stupid-one/¿. Acesso em: 10 jun. 2025.

[7] ATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *Gnome Sort. In: *Dictionary of Algorithms and Data Structures. Disponível em: ¡https://xlinux.nist.gov/dads/HTML/gnomeSort.html¿. Acesso em: 10 jun. 2025.

[8] HARIF UNIVERSITY OF TECHNOLOGY. *Stupid Sort: A New Sorting Algorithm*. Disponível em: ¡https://sina.sharif.edu/ azad/stupid-sort.PDF¿. Acesso em: 11 jun. 2025.

[9] IKIPEDIA. *Gnome Sort*. Disponível em: ¡https://en.wikipedia.org/wiki/Gnome_sort¿. Acesso em: 12 jun. 2025.

[10] IKIPEDIA. Insertion sort. Disponível em: https://pt.wikipedia.org/wiki/Insertion_sort. Acesso em: 12 jun. 2025.

[11] OUTUBE. Lógica do algoritmo de ordenação insertion sort. Disponível em: https://www.youtube.com/watch?v=7GUwzd_h3pI. Acesso em: 13 jun. 2025.

[12] IKIPEDIA. Thomas H. Cormen. Disponível em: https://en.m.wikipedia.org/wiki/Thomas_H._Cormen. Acesso em: 11 jun. 2025.

[13] ANENBAUM, A. S.; AUSTIN, T. Organização estruturada de computadores. 6. ed. São Paulo: Pearson Prentice Hall, 2013.

[14] IKIPEDIA. Sorting algorithm – History. Disponível em: https://en.wikipedia.org/wiki/Sorting_algorithm#History. Acesso em: 6 jun. 2025

[15] EEKSFORGEEKS. Timsort Algorithm – Data Structures and Algorithms Tutorials. Disponível em: https://www.geeksforgeeks.org/dsa/timsort/. Acesso em: 11 jun. 2025.

[16] ARG, Sakshi; SINGH, Himani. Sorting Algorithms – A Comparative Study. ResearchGate, 2017. Disponível em: https://www.researchgate.net/publication/315662067_Sorting_Algorithms_-_A_Comparative_Study. Acesso em: 6 jun. 2025.