UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Bruno Henrique Zara
João Victor Millane
Vinicius Augusto Borgue

Análise do algoritmo Selection Sort

São José do Rio Preto
2023

Bruno Henrique Zara
João Victor Millane
Vinicius Augusto Borgue

Análise do algoritmo Selection Sort

Relatório sobre análise de complexidade do algoritmo Selection Sort para a disciplina de Projeto e Análise de Algoritmos do Bacharelado de Ciência da Computação da Universidade Estadual Paulista (UNESP).

Orientador: Profa. Vitoria Zanon Gomes

São José do Rio Preto
2023

Sumário

1 Introdução

A análise de algoritmos é um campo fundamental na computação e desempenha um papel crucial no desenvolvimento de sistemas computacionais eficientes. Em um mundo cada vez mais dominado pela tecnologia, onde grandes volumes de dados precisam ser processados em tempo real, entender e analisar algoritmos torna-se uma necessidade.

Dessa forma, a análise de algoritmos proporciona uma compreensão profunda do desempenho dos algoritmos em diferentes cenários e condições. Ela permite avaliar a eficiência de um algoritmo, predizer seu comportamento sob diferentes entradas e identificar padrões de desempenho. Ao entender como os algoritmos se comportam em termos de tempo de execução e uso de recursos, podemos fazer escolhas mais precisas sobre quais algoritmos utilizar em determinadas situações.

Dentre toda a imensidão de algoritmos criados, os algoritmos feitos para ordenação de dados compõem uma operação fundamental na computação e são essenciais em várias aplicações. Existem vários algoritmos de ordenação, cada um com suas próprias características e eficiência. O algoritmo *Selection Sort*, abordado por esse relatório, é um dos métodos mais simples e intuitivos para ordenar uma lista de elementos.

Portanto, nesse relatório, exploraremos o funcionamento do algoritmo *Selection Sort*, analisando principalmente sua complexidade nos piores e melhores casos. Além disso, examinaremos possíveis cenários em que o *Selection Sort* pode ser uma escolha apropriada quando comparado a outros algoritmos de ordenação.

2 Algoritmo

2.1 Problemas de ordenação

O algoritmo do *Selection Sort* se encaixa na classe de problemas de ordenação. A ordenação é uma operação fundamental na computação e tem uma ampla gama de aplicações em várias áreas. Ela envolve organizar elementos em uma sequência específica, geralmente em ordem crescente ou decrescente, com base em algum critério de comparação. Existem diferentes algoritmos de ordenação adequados para diferentes situações, cada um com suas vantagens e desvantagens. Exemplos de sequências ordenadas:

- Ordenação crescente

- Ordenação decrescente

2.1.1 Tipos de algoritmo de ordenação

Dentro dos algoritmos de ordenação, existem várias abordagens e métodos utilizados para cumprir o propósito da ordenação. Cada algoritmo possui suas próprias características e podem ser agrupados em determinadas classes de acordo com seu funcionamento. No caso do *Selection Sort*, é considerado um algoritmo de ordenação por comparação, geralmente instável e interna (os dois últimos dependem da implementação).

2.1.1.1 Ordenação por comparação

A ordenação comparativa é o tipo mais comum de ordenação, onde os algoritmos comparam elementos da lista para determinar a ordem correta. Em suma, os algoritmos de ordenação comparativa se baseiam em operações de comparação entre pares de elementos para decidir a posição relativa deles na lista ordenada. Exemplos de algoritmos de ordenação comparativa comuns incluem o *Bubble Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*.

2.1.1.2 Ordenação não comparativa

A ordenação não comparativa, por outro lado, não se baseia em comparações diretas entre elementos para determinar a ordem. Em vez disso, esses algoritmos utilizam técnicas diferentes para ordenar os elementos. Um exemplo clássico de ordenação não comparativa é o Radix Sort.

2.1.1.3 Ordenação estável

Um algoritmo de ordenação é considerado estável quando preserva a ordem relativa dos elementos iguais na lista ordenada. Ou seja, se dois elementos têm chaves iguais e um deles aparece antes do outro na lista não ordenada original, um algoritmo de ordenação estável garantirá que esses dois elementos permaneçam nessa ordem relativa na lista ordenada resultante.

Como exemplo, suponha que um algoritmo de ordenação estável receba a seguinte lista como entrada:

3[a]	4[b]	3[c]	1[d]

Nesse caso, iremos ordenar a lista de forma crescente apenas com base nos números. Como o algoritmo é estável, a ordem 3[a], 3[c] é mantida, ou seja, elementos com a mesma chave aparecerão com a mesma ordem na lista final. O resultado é:

1[d]	3[a]	3[c]	4[b]

A estabilidade de um algoritmo de ordenação é definida por alguns fatores. Certos algoritmos, como o *Merge Sort*, são estáveis por natureza, sem necessitar de mudanças no código. Outros algoritmos, como o *Quick Sort*, podem necessitar de mudanças ou verificações no código original para serem estáveis.

2.1.1.4 Ordenação instável

Por sua vez, um algoritmo de ordenação é considerado instável se ele não garantir a preservação da ordem relativa dos elementos iguais na lista ordenada. Isso significa que, ao ordenar uma lista com elementos iguais, um algoritmo instável pode alterar a ordem relativa desses elementos na lista final.

No mesmo exemplo anterior, ao ordenar aquela lista, o algoritmo poderia ter alterado a ordem de 3[a], 3[b], não mantendo a ordem relativa entre elementos de uma mesma chave na lista. O resultado poderia ser:

1[d]	3[c]	3[a]	4[b]

Como já dito anteriormente, o *Quick Sort* original é um exemplo de algoritmo de ordenação instável. No entanto, é importante pontuar novamente que a estabilidade pode ser alcançada ao modificar comparações ou partes do código.

2.1.1.5 Ordenação interna

A ordenação interna refere-se ao processo de ordenar dados que estão completamente carregados na memória principal (RAM) do computador. Quando todos os dados que precisam ser ordenados cabem na memória principal, os algoritmos de ordenação podem acessar diretamente esses dados na RAM, facilitando a manipulação e a ordenação eficiente.

Exemplos de algoritmos de ordenação interna, como *Bubble Sort*, *Merge Sort*, *Quick Sort* e *Insertion Sort*, trabalham com dados que estão presentes na memória RAM e podem facilmente comparar, acessar e trocar elementos devido à sua localização na memória principal.

2.1.1.6 Ordenação externa

A ordenação externa, por outro lado, é necessária quando os dados a serem ordenados são muito grandes para caber completamente na memória principal. Nesse caso, os dados são armazenados em dispositivos de armazenamento secundário e não podem ser totalmente acessados de uma vez na memória RAM.

Algoritmos de ordenação externa, como *Merge Sort* externo, são projetados para lidar com grandes volumes de dados que não cabem na memória principal. Esses algoritmos dividem e ordenam os dados em blocos gerenciáveis que podem ser manipulados na memória principal. Em seguida, eles mesclam esses blocos ordenados para criar a lista ordenada final, sem a necessidade de carregar todos os dados na memória RAM simultaneamente.

2.1.2 Aplicações da ordenação

Algoritmos de ordenação possuem aplicações diversas dentro da computação, como:

1. Bancos de dados
    - Em sistemas de gerenciamento de banco de dados, a ordenação é usada para organizar registros com base em critérios específicos, facilitando a recuperação eficiente de dados.
2. Sistemas de busca
    - Algoritmos de ordenação são usados em motores de busca para classificar os resultados de pesquisa com base na relevância, na data ou em outros critérios.

2.2 Funcionamento

O *Selection Sort* funciona selecionando repetidamente o menor (ou maior, dependendo da ordem desejada) elemento da lista não ordenada e trocando-o com o primeiro elemento não ordenado. O processo é repetido até que toda a lista esteja ordenada. Para simplificar, os passos do *Selection Sort* podem ser descritos da seguinte forma:

1. Entradas
    - O algoritmo pode receber vários parâmetros para controle. No mínimo, precisa-se saber o vetor com os elementos que serão ordenados e seu tamanho.
2. Divisão do vetor
    - O vetor/lista não ordenado é dividido em duas partes: a parte ordenada e a parte não ordenada. No início, a parte ordenada está vazia, enquanto a parte não ordenada contém todos os elementos do vetor.
3. Seleção do menor elemento
    - O algoritmo percorre a parte não ordenada do vetor para encontrar o índice que possui o menor elemento.
4. Troca de elementos
    - Após encontrar o menor elemento na parte não ordenada, ele é trocado com o primeiro elemento da parte não ordenada, colocando-o na posição correta na parte ordenada. Agora, a parte ordenada aumenta em um elemento, e a parte não ordenada diminui em um elemento.
5. Repetição
    - Os passos 2 e 3 são repetidos para o restante do vetor não ordenado até que não existam mais elementos na parte não ordenada.
6. Lista ordenada
    - Quando todos os elementos são movidos da parte não ordenada para a parte ordenada, o vetor está totalmente ordenado.

Como exemplo de funcionamento, suponha que temos o seguinte vetor com tamanho $n = 4$ como entrada do nosso algoritmo e queremos ordená-lo de forma crescente.

4	1	2	3

Para isso, começamos em $i = 0$ (primeira posição do vetor), e procuramos pelo índice com o menor elemento no subvetor à direita, que começa em $i+1$ e vai até n. Após realizada essa busca, encontramos que o menor elemento é 1, que está na posição $i = 1$. Dessa forma, trocamos $i = 0$ com $i = 1$, resultando no seguinte vetor:

1	4	2	3

Após isso, repetimos o mesmo procedimento, mas agora para $i = 1$. Nesse caso, encontramos que o menor elemento está em $i = 2$, então trocamos $i = 1$ por $i = 2$. Da troca, temos o seguinte vetor:

1	2	4	3

Analogamente, executamos a iteração para $i = 2$, encontrando $i = 3$ como o menor valor do vetor. Dito isso, trocamos $i = 2$ por $i = 3$, resultando no seguinte vetor:

1	2	3	4

Por fim, após executarmos todas as comparações do algoritmo e chegarmos em $i = n - 1$, teremos o nosso vetor completamente ordenado de forma crescente.

2.3 Código

Abaixo, temos o código do *Selection Sort* escrito na linguagem de programação *Python*:

```
1  def selection_sort(array, size):
2      for ind in range(size - 1):
3          min_index = ind
4
5          for j in range(ind + 1, size):
6              # seleciona o menor elemento a cada iteração
7
8              if array[j] < array[min_index]:
9                  min_index = j
10
11          # troca os elementos para ordenar o array
12         array[ind], array[min_index] = array[min_index], array[ind]
```

Como já comentado, o código em *Python* possui todos os passos necessários para o funcionamento correto do algoritmo:

1. Parâmetros: A função recebe o vetor com os elementos a serem ordenados e o seu tamanho.
2. Laço externo: O laço externo percorre de $i = 0$ até $i = size - 1$ (todo o comprimento do vetor).
3. Variável de índice mínimo: É atribuído o número do índice atual a uma variável que guarda o índice do menor valor do subvetor não ordenado.
4. Laço interno: O laço interno percorre a parte não ordenada do vetor, buscando pelo índice de menor elemento.

5. Comparação: É feito uma comparação para verificar se o valor do índice atual é menor que o valor do índice mínimo guardado.

6. Troca de valores: Depois de achar o índice com o menor valor na parte não ordenada, ocorre uma troca de elementos, posicionando esse valor no índice i.

7. Retorno: Após o fim da função, o algoritmo ordenou totalmente o vetor recebido.

2.4 Prova de corretude

Para provar a corretude do algoritmo de *Selection Sort*, pode-se seguir uma abordagem formal chamada "invariantes". Invariantes são condições que devem ser verdadeiras antes, durante e após a execução do algoritmo.

Nesse caso, podemos provar a corretude de um algoritmo utilizando a técnica da invariante de laço. Para isso, definimos essa propriedade (invariante) que é verdadeira antes e depois de cada iteração do laço (*loop*). Em seguida, deve-se provar três etapas:

1. Inicialização: A invariante é verdadeira antes da primeira iteração do laço.

2. Manutenção: Se a invariante é verdadeira antes de uma iteração do laço, ela é verdadeira também antes da próxima iteração.

3. Finalização: Quando o laço termina, a invariante implica na corretude do algoritmo.

Para o algoritmo de *Selection Sort*, a invariante do laço pode ser definida como: "Invariante: Após a k-ésima iteração do laço externo, os primeiros k elementos no vetor estão ordenados.".

Dessa forma, abaixo, vamos provar a corretude do algoritmo utilizando essa invariante de laço.

2.4.1 Inicialização

Antes da primeira iteração do laço (quando $k = 0$), não há elementos na parte ordenada, e a invariante é verdadeira, pois um vetor vazio ou com apenas um elemento naturalmente é um vetor ordenado.

2.4.2 Manutenção

Agora, suponha que a invariante seja verdadeira para uma iteração do laço, ou seja, após a k-ésima iteração, os primeiros k elementos estão ordenados.

Durante a $(k + 1)$-ésima iteração, o algoritmo encontra o menor elemento na parte não ordenada e o troca com o elemento na posição k. Agora, os primeiros $k + 1$ elementos estão ordenados, aumentando o tamanho do nosso vetor de valores ordenados. Dessa forma, durante a iteração e execução do algoritmo, mantemos o nosso vetor sempre ordenado, garantindo que a invariante seja conservada.

2.4.3 Finalização

Quando o laço termina (quando $k = n - 1$, onde n é o tamanho do vetor), a invariante implica que toda a lista está ordenada, pois após a n-ésima iteração, os n primeiros elementos estão ordenados.

Portanto, a invariante de laço prova que o algoritmo de *Selection Sort* funciona corretamente, garantindo que o vetor esteja ordenado após a conclusão do algoritmo.

3 Complexidade

3.1 Melhores e piores casos

3.1.1 Pior caso

No *Selection Sort*, o pior caso ocorreria quando temos uma lista na ordem inversa àquela que queremos obter (quando queremos uma lista ordenada de forma crescente e temos uma decrescente, por exemplo).Para calcular o pior caso do algoritmo, primeiro vamos analisar a sua complexidade de tempo. Nesse caso, definimos o custo das operações principais como:

1. Declarações de variáveis e chamadas de função: sem custo na soma.
2. Operações aritméticas, lógicas, atribuições e leitura/escrita de arquivos: custo 1.
3. Estruturas condicionais: custo do caminho mais longo (if/else).
4. Estruturas de repetição: custo de uma iteração multiplicado pelo número de iterações.

Com isso definido, iremos analisar linha por linha do código fornecido na seção anterior para calcular sua complexidade.

Nas linhas 2-3, temos:

```
1     for ind in range(size - 1):
2         min_index = ind
```

A atribuição, com custo 1, é executada $n-1$ vezes pelo algoritmo, resultando em custo $n-1$.

Nas linhas 5-9, temos:

```
1     for j in range(ind + 1, size):
2         # seleciona o menor elemento a cada iteração
3
4         if array[j] < array[min_index]:
5             min_index = j
```

Nessa estrutura de repetição, temos uma atribuição com custo 1 por iteração e uma comparação com custo 1 (operação lógica). No entanto, o laço é executado $(size - ind - 1)$ vezes, e também repetido mais $n-1$ vezes pelo laço mais externo anterior. Isso significa que, na primeira execução, o laço interno executa $n-1$ vezes, na segunda, $n-2$, até que chegue na ultima com execução $n-(n-1)=1$. Logo, a quantidade de vezes que esse laço interno executa pode ser escrito como:

$$(n-1)+(n-2)+(n-3)+...+1=\sum_{i=1}^{n-1} i=\frac{n\times(n-1)}{2}$$

Dessa forma, a soma da quantidade de execuções da comparação e atribuição seria $n\times(n-1)$. Por aqui, já podemos observar que o algoritmo terá, no mínimo, complexidade quadrática em seu tempo de execução;

Nas linhas 11-12, temos:

```
1     # troca os elementos para ordenar o array
2     array[ind], array[min_index] = array[min_index], array[ind]
```

Que é equivalente a:

```
1     aux = array[ind]
2     array[ind] = array[min_index]
3     array[min_index] = aux
```

Nesse caso, temos três atribuições, sendo repetidas $n-1$ vezes pelo laço mais externo, resultando em uma quantidade de $3 * (n-1)$.

No final, a soma de cada um dos custos é aproximadamente:

$$T(n) = (n-1) + (n \times (n-1)) + (3 \times (n-1)) = n^2 + 3n - 4$$

Utilizando a notação Big-O, temos que a nossa complexidade no pior caso do algoritmo é:

$$\mathcal{O}(f(n)) = \mathcal{O}(n^2 + 3n - 4)$$
$$\mathcal{O}(f(n)) = \mathcal{O}(n^2) + \mathcal{O}(3n) + \mathcal{O}(4)$$
$$f(n) = \mathcal{O}(n^2)$$

3.2 Melhor caso

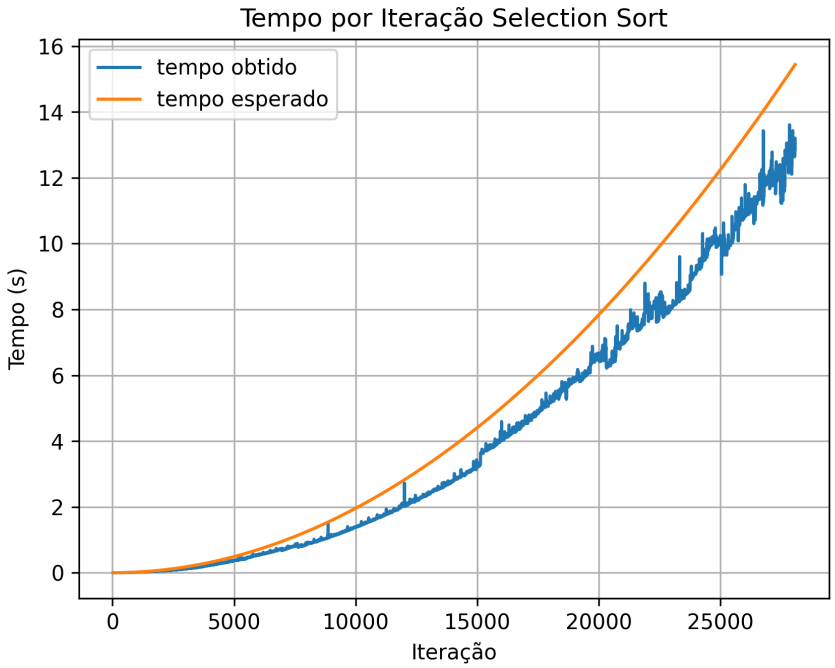
No *Selection Sort*, o melhor caso ocorreria quando temos uma lista previamente ordenada com a sequência correta que queremos. No entanto, o algoritmo original não é projetado para detectar uma lista previamente ordenada. Dessa forma, em todos os casos, é necessário percorrer toda a lista na procura do menor elemento para fazer a seleção e a troca, independentemente do elemento em questão já estar na posição desejada.

Nessa situação, embora o algoritmo nunca ache um elemento que seja menor que o elemento a ser substituído na lista não ordenada, ele ainda realiza o mesmo número de comparações para cada posição da lista, pois precisa garantir sempre que o elemento alvo é o menor de todos.

Dessa forma, o número de comparações se mantém igual ao do pior caso, sendo $(n-1)$ para a primeira busca, $(n-2)$ para a segunda e assim por diante. Como o número de comparações do algoritmo se mantém, a curva de complexidade também possui a mesma taxa de crescimento, continuando com a complexidade de $\mathcal{O}(n^2)$.

3.3 Gráficos de complexidade

Figura 1 – Comparação entre complexidade de tempo teórica e real



Nessa seção, foi construído um gráfico para comparar o tempo de complexidade teórica e o real. Para construir a curva de tempo de execução real, foi utilizado o algoritmo já citado anteriormente, executando-o e medindo seu tempo com várias iterações, para n suficientemente pequenos e grandes (mais do que $n = 25000$), aumentando de 5 em 5 a quantidade de valores a serem ordenados.

É importante notar que, a partir de um certo valor de iteração (aproximadamente 20000), os valores de tempo, embora ainda crescentes, apresentam uma variação bem maior se comparados com as primeiras iterações. Isso pode ocorrer por diversas razões, dentre elas: os recursos do sistema, como memória e processamento (*hardware*), podem influenciar os tempos de execução; se a complexidade for alta (por exemplo, $\mathcal{O}(n^2)$ ou pior), o aumento exponencial no tempo de execução para entradas maiores pode resultar em variações grandes.

Para construir a curva de tempo de execução teórico, foi utilizado a função de tempo esperada obtida nos cálculos de complexidade do melhor e pior caso do algoritmo, nesse caso, $f(n) = n^2$. No entanto, como estamos utilizando a notação Big-O ($\mathcal{O}$), devemos conseguir exibir a curva de tempo teórico como um limite superior da função real. Para isso, encontramos valores de c1 e n0, tal que:

$$0 \leq f(n) \leq c_1 \cdot g(n), \forall n \geq n_0$$

Dessa forma, os dois valores obtidos foram:

- $c1 = 0.00014$
- $n0 =$ iteração 25

Nesse caso, para todo valor de n maior que 25, podemos garantir que o limite superior assintótico é válido e $c_1 \dot{g}(n)$ é sempre maior ou igual a $f(n)$, lembrando que as medidas foram feitas com quantidades múltiplas de 5.

4 Conclusão

De forma geral, com base na análise de complexidade feita, pode-se concluir que o *Selection Sort*, assim como outros algoritmos de ordenação, possui seus pontos fracos, fortes e casos de uso em que ele é mais adequado.

Em relação aos pontos fortes do algoritmo, podemos citar:

1. Implementação simples
    - O *Selection Sort* é um dos algoritmos de ordenação mais simples de entender e implementar. Essa simplicidade torna-o uma escolha popular para fins educacionais e em situações onde a simplicidade do código é essencial.
2. Espaço adicional mínimo
    - O *Selection Sort* opera *in-place*, o que significa que ele não requer espaço adicional para armazenar, por exemplo, uma cópia da lista durante o processo de ordenação. Isso faz com que seja mais eficiente em termos de uso de memória, pois o algoritmo não consome espaço adicional proporcional ao tamanho da lista.

Em relação aos pontos fracos, podemos citar:

1. Complexidade quadrática
    - O *Selection Sort* tem uma complexidade de tempo $\mathcal{O}(n^2)$ no pior e melhor caso. Isso significa que o tempo de execução do algoritmo aumenta quadraticamente com o número de elementos na lista.
2. Falta de adaptação
    - O *Selection Sort* não é um algoritmo adaptativo. Isso significa que, mesmo que a lista já esteja parcialmente ordenada, o Selection Sort ainda faz o mesmo número de comparações. Não importa, por exemplo, se 90% da lista já está ordenada, ele não levará em conta o nível de ordenação da lista.

Dito isso, agora podemos indicar em qual caso o *Selection Sort* pode ser melhor aplicado.

O *Selection Sort* pode ser eficiente para ordenar pequenas listas, especialmente quando o número de elementos é muito pequeno. Isso ocorre porque, nesses cenários, como a ordenação é mais simples, algoritmos que utilizam métodos complexos, como divisão de listas ou recursão (*Quick Sort* e *Merge Sort*, por exemplo), podem introduzir instruções desnecessárias, ocasionando *overhead* na execução.

Por outro lado, como o *Selection Sort* possui essa complexidade de tempo $\mathcal{O}(n^2)$, seu desempenho deteriora rapidamente à medida que o número de elementos na lista aumenta. Para grandes conjuntos de dados, o Selection Sort se torna extremamente lento e ineficiente quando comparado a algoritmos de ordenação mais rápidos, como *Merge Sort* ou *QuickSort*. Além disso, em sistemas que exigem operações em tempo real, onde as respostas devem ser geradas em milésimos de segundos, o *Selection Sort* não é uma escolha adequada, pois seu desempenho igualitário em diferentes tipos de entrada pode levar a atrasos inaceitáveis.

Portanto, para escolher se o *Selection Sort* é viável ou não para uma implementação, torna-se necessário avaliar em qual desses casos o problema se encaixa, focando principalmente no *trade-off* entre simplicidade x eficiência.

Referências

HEINEMAN, G.; POLLICE, G.; SELKOW, S. *Algorithms in a Nutshell*. [*S.l.*]: O'Reilly Media, 2008. (In a Nutshell (O'Reilly)). ISBN 9781449391133.