Análise da Ordenação por Seleção de Raiz Quadrada

Giovanni Lucas O. da Silva[1]

[1]Instituto Federal de Brasília Campus Taguatinga (IFB)
72146-050 – Brasília – DF – Brazil

giovanni61540@estudante.ifb.edu.br

Abstract. *This work aims to analyze two possible implementations of the square root selection sorting algorithm. Both an empirical approach, through practical experimentation, and an analytical analysis will be used to understand the performance and efficiency of this algorithm in different scenarios. By combining these approaches, we hope to obtain a comprehensive understanding of these implementations of this sorting algorithm.*

Resumo. *Este trabalho tem como objetivo analisar duas possibilidades de implementação do algoritmo de ordenação por seleção de raiz quadrada. Para sua análise sera usado tanto uma abordagem empírica, através de experimentação prática, quanto uma análise analítica serão utilizadas para compreender o desempenho e a eficiência deste algoritmo em diferentes cenários. Ao combinar essas abordagens, esperamos obter uma compreensão abrangente destas implementações deste algoritmo de ordenação.*

1. Introdução

A ordenação é uma das operações mais fundamentais na ciência da computação, é o processo de organizar elementos de uma coleção, como um vetor, lista ou registros, em uma sequência específica, geralmente em ordem crescente ou decrescente e é essencial para a eficiência de muitos algoritmos e sistemas. Conforme discutido por Cormen, a ordenação serve como base para uma ampla variedade de aplicações, desde a organização de dados até a otimização de algoritmos de busca e fusão de listas. [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein 2009]

Os métodos de ordenação utilizam uma chave de ordenação onde este item é utilizado para fazer a comparação com outros elementos e por meio deste sabemos se um determinado elemento está ou não na frente de outros baseado em uma regra bem definida como a numérica ou lexicográfica (ordem alfabética), sendo que independente da regra ela pode ser crescente como um vetor $A = \{1, 2, 3, 4, 5\}$ ou $B = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$ e decrescente como $A = \{5, 4, 3, 2, 1\}$ ou $B = \{\mathrm{e}, \mathrm{d}, \mathrm{c}, \mathrm{b}, \mathrm{a}\}$, com isso podemos encontrar e posicionar os nossos elementos de forma que todos respeitem a regra e assim teremos no final um vetor ordenado.

Para alcançar seu resultado final os métodos podem utilizar diferentes técnicas para alcançar esse vetor ordenado, podendo ser mais ou menos perfomáticas de acordo com seu algoritmo ou gastar ou não mais memória em sua utilização, outro conceito básico de ordenação na programação seria a de ela ser estável ou não, isto é "um algoritmo de ordenação é considerado estável se a ordem dos elementos com chaves iguais não muda

durante a ordenação" [André Backes 2021] isto é, ele preserva a ordem relativa original dos valores.

"As técnicas de ordenação também fornecem excelentes ilustrações das ideias gerais envolvidas na análise de algoritmos — as ideias usadas para determinar as características de desempenho dos algoritmos para que uma escolha inteligente possa ser feita entre métodos concorrentes." [Donald E. Knuth 1998]

Muitos métodos de ordenação diferentes foram inventados e cada um deles têm suas características principais de funcionamento e neste trabalho será implementado um método com duas estratégias diferentes para sua implementação. Este método se chama Ordenação por Seleção de Raiz Quadrada e ele busca dividir em blocos um vetor e aplicar técnicas para ordena-lo.

Analizaremos 2 formas de implementa-lo e suas diferenças, melhorias e com isso chegar a conclusão de como estas maneiras diferem o resultado final de cada algoritmo.

2. Ordenação por Seleção de Raiz Quadrada

O algoritmo de ordenação por seleção de raiz quadrada que iremos analisar, daqui em diante referenciado como *sqrtsort*, organiza os elementos de um vetor V seguindo os passos descritos abaixo:

Figura 1. Vetor V de tamanho $n = 7$

1. **Divisão do Vetor:** O vetor V é dividido em partes (blocos) cada uma contendo $\lfloor\sqrt{n}\rfloor$ elementos, onde n é o número total de elementos do vetor. Se o tamanho total n não for múltiplo de $\sqrt{n}$, a última subseção terá $n \mod \lfloor\sqrt{n}\rfloor$ elementos.

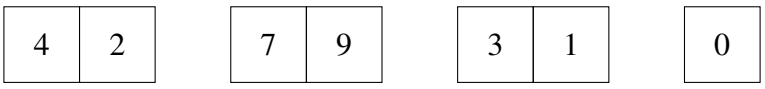

Figura 2. Vetor dividido em blocos de $\sqrt{7}$ e último bloco de tamanho $7 \mod \lfloor\sqrt{7}\rfloor$

2. **Identificação dos Maiores Elementos:** É identificado o maior elemento de cada bloco.

Figura 3. Blocos com os maiores elementos identificados

3. **Construção do Vetor Ordenado:** Entre os maiores elementos de cada bloco, o maior é selecionado e inserido em um vetor final ordenado. Após a inserção, o elemento é removido do bloco de origem.

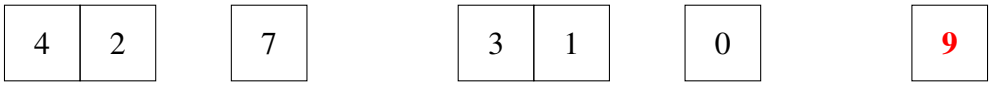

Figura 4. Blocos já atualizados e vetor com o elemento 9 já retirado do bloco de origem

4. **Repetição do Processo:** O processo de identificação e inserção dos maiores elementos continua até que todas os blocos iniciais estejam vazios restando apenas um vetor final ordenado.

Figura 5. Vetor final com os elementos ordenados após as repetições

Este projeto explora duas abordagens para implementar o *sqrtsort*: Utilizar um método de ordenação quadrática para resolver o passo 2, neste caso será adotado o algoritmo de inserção, e a outra forma é a utilização de uma estrutura *Heap* para fazer o mesmo trabalho.

2.1. Implementações

Antes de começar a analisar veremos o funcionamento básico dos métodos que foram utilizados e como foi utilizado cada um deles para chegar nas implementações do *sqrtsort*.

2.1.1. Utilizando a ordenação por inserção

O algoritmo ordenação por inserção conhecido também por *Insertion Sort*, é um algoritmo bastante simples para fazer ordenações, ele tem como premissa inserir certo elemento de um vetor para ocupar uma certa posição de forma que o elemento anterior é menor e o próximo é maior, fazendo isso para todas as posições do array, colocando a cada iteração do algoritmo um elemento em sua posição correta naquele instante, até que o vetor esteja ordenado, e ele tem custo $\Theta(n^2)$ [Donald E. Knuth 1998] , por isso chamado de algoritmo quadrático.

Se aplicarmos este algoritmo em todos os blocos do exemplo anterior, seus elementos ficaram ordenados da seguinte forma:

Figura 6. Blocos com os elementos ordenados

Com esses elementos ordenados podemos achar o seu maior elemento em cada bloco simplesmente analisando sua última posição e com isso fazer comparações entre as últimas posições para achar quem é o maior entre eles e assim poder iniciar a montagem do vetor final, removendo o maior elemento e colocando no vetor final, preenchendo ele de traz para frente gerando assim um vetor ordenado de forma crescente.

2.1.2. Utilizando a estrutura de Heap

Heap é uma estrutura de dados que utiliza uma estrutura de array que pode ser visualizada como uma árvore binária quase completa [Srini Devadas 2011] onde seus nós deveram ser maiores que seus filhos quando implementada como *Max Heap* e esse conceito que será utilizado para desenvolver a tarefa de achar o maior elemento entre os blocos, e para isso teremos de transformar esses blocos de arrays em estruturas *Heap* e dai utilizar outras funções características da estrutura para chegar no resultado final.

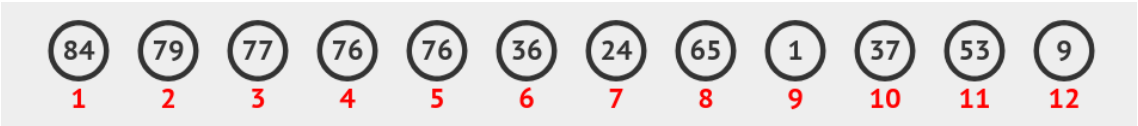

Figura 7. Vetor V em estrutura heap

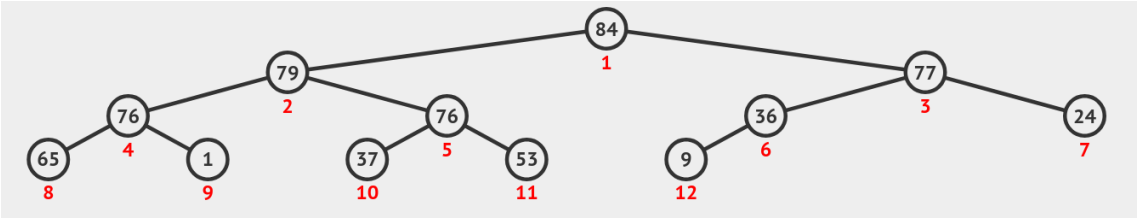

Figura 8. Vetor V na sua visualização de árvore

Entre estas funções estão:

- **Make Heap**: Transforma um vetor V em uma heap. Esta operação tem um custo de $\Theta(n)$ e é necessária antes de realizar qualquer outra operação na heap. A transformação organiza o vetor de maneira que ele satisfaça as propriedades de uma heap onde cada representação de nó é maior que seus filhos.
- **Insert Heap**: Insere um elemento x em uma heap. O custo desta operação é $\Theta(\log n)$, pois envolve comparar o elemento inserido com os elementos já presentes na heap e movê-lo para a posição correta, mantendo as propriedades da heap.
- **Remove Heap**: Remove o maior elemento da heap. Além disso, reestrutura a heap para manter suas propriedades. Em outras palavras, se x for o maior elemento da heap, ele será removido, e o próximo maior elemento $y \leq x$ será movido para o topo da heap, o que significa que será o próximo a ser removido, a menos que um novo elemento $z > y$ seja inserido antes. Esta operação também tem custo $\Theta(\log n)$.

Com isso, se trabalharmos com a estrutura *Heap*, poderemos utilizar a função de remoção para achar o maior entre os elemetos e colocar ele no vetor final.

3. Análise dos Algoritmos

A análise dos algoritmos envolve o estudo e a avaliação do desempenho dos algoritmos, e neste trabalho será feito utilizando a notação assintótica, que é utilizada para descrever

o comportamento temporal de um algoritmo em termos de sua eficiência e complexidade. Esta notação nos ajuda a ter uma aproximação do que se pode encontrar na realidade, pois "ela considera o comportamento de funções no limite" [Flavio moura 2023], o real desempenho vai depender da capacidade de cada computador.

3.1. Implementação com *Insertion Sort*

Para a implementação com o *Insertion Sort*, ordenaremos cada bloco de tamanho $\sqrt{n}$ utilizando o *Insertion Sort* que é conhecido por ter um tempo de execução quadrático no pior caso, que como explicado anteriormente para ordenar um bloco de tamanho n ele tem custo $\Theta(n^2)$, já que o tamanho de cada bloco é $\sqrt{n}$, então o tempo de ordenação para um único bloco usando *Insertion Sort* será $\Theta((\sqrt{n})^2)$, que simplifica para $\Theta(n)$.

Como estamos dividindo o vetor em blocos de $\sqrt{n}$ então teremos um número total de blocos igual a $\frac{n}{\sqrt{n}}$ que é o mesmo que $\sqrt{n}$. Como já sabemos que o tempo de ordenação de cada bloco é $\Theta(n)$, então, o tempo total necessário para ordenar todos os blocos é de $\Theta(n)$ vezes $\Theta(\sqrt{n})$, o que resulta em uma complexidade final de $\Theta(n\sqrt{n})$ para ordenar todos os vetores.

Agora com os vetores ordenados podemos fazer comparações constantes para achar o maior elemento entre eles e inserir no vetor solução, onde iremos comparar entre os ultimos elementos dos vetores e inserir no vetor solução, com isso faremos $\sqrt{n} - 1$ comparações n vezes resultando em um final de $n\sqrt{n}$ comparações o que leva também a uma complexidade de $\Theta(n\sqrt{n})$, podemos com isso afirmar que o algoritmo tem complexidade de $\Theta(n\sqrt{n}) + \Theta(n\sqrt{n})$ que resulta em uma complexidade assintótica final de $\Theta(n\sqrt{n})$.

3.2. Implementação com a estrutura *Heap*

Para essa implementação vamos utilizar as funções apresentadas anteriormente para fazer a ordenação, ainda na ideia dos blocos de tamanho $\sqrt{n}$, agora vamos tranformar eles em um estrutura *Heap* utilizando a função *Make Heap* que tem complexidade $\Theta(n)$ que neste caso, dado o tamanho de n, será $\Theta(\sqrt{n})$ e como essa função será aplicada em $\sqrt{n}$ blocos podemos dizer que a complexidade de deixar todos os blocos em estrutura *Heap* se torna $\sqrt{n} * \Theta(\sqrt{n})$ que se torna $\Theta(n)$.

Após isso será utilizado um vetor auxiliar para guardar os valores das primeiras posições de cada bloco, pois na estrutura *Heap* o maior elemento fica na primeira posição, e após isso, trasformar também esse vetor em *Heap* utilizando o *Make Heap* uma unica vez para este vetor de tamanho $\sqrt{n}$, logo teremos para essa operação, o custo de $\Theta(\sqrt{n})$.

Com isso feito podemos agora adicionar o primeiro elemento do vetor auxiliar na última posição do vetor final, e assim vamos utilizar as funções de inserção e remoção em *Heaps*, onde ambas tem tmepo de execução $\Theta(log\ n)$, removemos do vetor auxiliar o primeiro elemento para colocar no vetor final e depois inserimos neste vetor auxiliar o próximo numero do bloco onde foi retirado. como faremos n vezes essa função até todos os blocos ficarem vazios, toda essa operação terá custo de $n * \Theta(log\sqrt{n}) = \Theta(n\ log\ n)$ e ao final teremos um vetor ordenado.

O tempo de toda essa função será a soma de todos estes custos, mas como estamos analizando em termos assintóticos ela terá tempo $\Theta(n\ log\ n)$ pois dentre todas ela será a função que dominará as outras em custo de execução.

4. Análise empírica

4.1. Metodologia

Na realização dos experimentos foi adotada a linguagem de programação “C” para a produção dos algoritmos de ordenação, utilizando na compilação destes programas a flag de otimização “-O3”.

Os experimentos foram conduzidos utilizando um script “Bash” para realizar a execução dos códigos cinco vezes e realizar a média aritmética dos tempos de execução das linhas de código referentes a cada ordenação.

Um vetor de tamanho n foi criado para a utilização dos algoritmos e sua população foi efetuada de forma pseudo-aleatória utilizando a função $rand()$, com seu tamanho podendo assumir os valores de 10^4, 10^5, 10^6, 10^7 e 10^8.

4.2. Resultados

Os resultados encontrados para cada tamanho de n foram atribuídos em uma tabela para a geração de um gráfico para melhor visualização.

Tabela 1. Tabela com os valores de execução em segundos em relação ao tamanho do vetor.

Tamanho do Vetor	Tempo Ordenação Inserção	Tempo Ordenação Heap
10000	0.000780	0.000797
100000	0.021991	0.009385
1000000	0.774853	0.137027
10000000	93.88395	3.624790
100000000	1202.59	66.493118

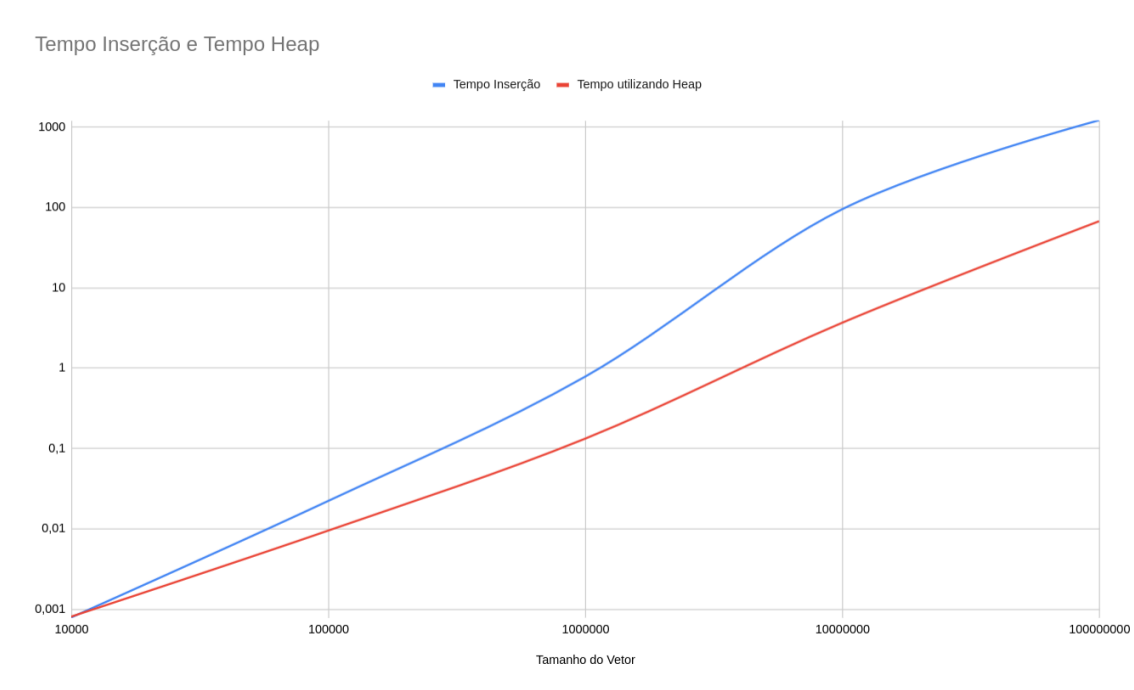

Figura 9. Gráfico com as buscas feitas pelo vetor.

Pelos dados encontrados é possível analizar que a utilização da inserção na implementação desta busca faz com que ela tenha um custo maior do que o uso da estrutura $Heap$, e podemos perceber que elas até que se acompanham quando o vetor é bem pequeno, e quando ele cresce a diferença entre elas cresce junto.

Com o que conseguimos achar na análise dos algoritmos propostos, as curvas parecem estar de acordo, enquanto a curva da implementação com inserção cresce a um custo $\Theta(n\sqrt{n})$ a utilizando a estrutura ***Heap*** cresce a um custo menor de $\Theta(n\ log\ n)$.

4.3. Complexidade ciclomática

O valor da complexidade ciclomática obtida automaticamente pelo site SonarCloud gerou que a ordenação por seleção de raiz quadrada utilizando a inserção têm o valor ciclomático 18 e a implementação por ***Heap*** 27, analisando estes resultados empíricos fornecidos por cada implementação é possível notar que a medida que o algoritmo fica mais complexo isto é, ele tem a possibilidade de percorrer mais caminhos, ele não altera o resultado final de desempenho dos algoritmos, pois mesmo o numero aumentando em 9, o desempenho não aumenta com alguma relação, a analise ciclomática é mais uma métrica de construção do código do que uma relação com seu custo. "A complexidade ciclomática é definida como a medição da quantidade de lógica de decisão em uma função de código-fonte [NIST235]. Simplificando, quanto mais decisões precisarem ser tomadas no código, mais complexas serão.". [Microsoft 2022]

4.4. Código

O código utilizado nestes experimentos se encontram neste endereço: `https://github.com/Giovanni-LOS/Sqrt_Sort`.

4.5. Ambiente experimental

As especificações da máquina podem impactar o desempenho dos algoritmos implementados, especialmente o tempo de execução e o uso de memória. As especificações de hardware e software utilizadas para a execução dos algoritmos de busca e ordenação:

Todos os testes foram executados e realizados em um notebook com o sistema operacional Arch Linux x86_64 com processador Intel i7-1165G7 (8) @ 4.700 GHz de décima primeira geração.

5. Discussão

Com essas informações podemos discorrer sobre o comportamento real e o esperado destes testes:

Partindo de uma comparação teórica podemos dizer que por um algoritmo ter complexidade de $\Theta(n\sqrt{n})$, implica que, para grandes valores de n, o número de operações aumenta mais rapidamente que em comparação ao algoritmo com $\Theta(n\ log\ n)$ de custo.

Ambos os algoritmos também foram implementados e testados empiricamente, e observou-se que o algoritmo com complexidade $\Theta(n\sqrt{n})$ apresentou um tempo de execução maior do que o algoritmo com complexidade $\Theta(n \log n)$. A seguir, discutimos as razões teóricas para esse comportamento:

O algoritmo com complexidade $\Theta(n\sqrt{n})$ realiza um número de operações que cresce de acordo com a função $f(n) = n\sqrt{n}$:

$$f(n) = n\sqrt{n} = n^{3/2}$$

Esta função representa uma taxa de crescimento polinomial com um expoente maior que 1. À medida que n aumenta, o número de operações realizadas pelo algoritmo cresce significativamente.

Por outro lado, o algoritmo com complexidade $\Theta(n \log n)$ realiza um número de operações que cresce de acordo com a função $g(n) = n\log(n)$. Essa função cresce mais lentamente do que qualquer função polinomial com grau maior que 1:

$$g(n) = n \log n$$

Isso implica que a função $f(n)$ eventualmente ultrapassa a função $g(n)$ para valores suficientemente grandes de n.

Na análise empírica, observou-se que o algoritmo com complexidade $\Theta(n\sqrt{n})$ levou mais tempo para ser executado do que o algoritmo com complexidade $\Theta(n \log n)$ a medida que n crescia. Este resultado está de acordo com a análise teórica, já que a função $f(n) = n\sqrt{n}$ cresce mais rapidamente do que $g(n) = n\log(n)$.

Para valores pequenos de n, a diferença entre $n\sqrt{n}$ e $n \log n$ pode não ser significativa, mas à medida que n aumenta, o número de operações realizadas pelo algoritmo com complexidade $\Theta(n\sqrt{n})$ cresce mais rapidamente, resultando em um tempo de execução maior.

6. Considerações Finais

Embora a análise assintótica ignore fatores constantes, na prática, a diferença entre $\Theta(n\sqrt{n})$ e $\Theta(n \log n)$ é significativa o suficiente para ser observada empiricamente. Cenários de teste com grandes valores de n, como os realizados neste trabalho, amplificam essa diferença, explicando o tempo de execução maior do algoritmo com complexidade $\Theta(n\sqrt{n})$.

O comportamento observado na análise empírica está em conformidade com as previsões teóricas. O algoritmo com complexidade $\Theta(n\sqrt{n})$ requer naturalmente mais tempo de execução do que o algoritmo com complexidade $\Theta(n \log n)$ para grandes valores de n. Isso se deve ao fato de que a função $n\sqrt{n}$ cresce muito mais rapidamente do que $n \log n$, resultando em um aumento significativo no número de operações necessárias.

Agora, mesmo que o algoritmo de ordenação *Insertion Sort* seja quadrático, com essa implementação conseguimos fazer uma melhora em seu custo, saindo de $\Theta(n^2)$ para $\Theta(n\sqrt{n})$, já se analizarmos um algoritmo de ordenação que utiliza também as funções da *Heap* para vetores inteiros, o *Heap Sort*, que têm complexidade $\Theta(n \log n)$, com essa estratégia de divisão ele não sofreu nenhuma melhora de complexidade.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[3] Backes, A. (2021). *Aula 06 - Ordenação*. Faculdade de Computação, UFU. Disponível em https://www.facom.ufu.br/ backes/gsi011/Aula06-Ordenacao.pdf. Acesso em: 18 ago. 2024.

[4] S. Devadas. 6.006 Introduction to Algorithms, Fall 2011, Lecture 4: Heaps and Heap Sort. Massachusetts Institute of Technology, 2011. Disponível em: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-4-heaps-and-heap-sort/. Acesso em: agosto de 2024.

[5] MOURA, Flávio. *PAA 2023 - Aulas 05 e 06*. 2023. Disponível em: http://flaviomoura.info/files/paa-2023-1-aulas05e06.pdf. Acesso em: [20 ago. 2024].

[6] Microsoft Docs. Métricas de código: Complexidade ciclomática. Disponível em: https://learn.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022. Acesso em: 27 ago. 2024.