



**FACULDADE DE TECNOLOGIA DE MAUÁ**

**DESENVOLVIMENTO DE SOFTWARE MULTIPLATAFORMA**

**ESTRUTURA DE DADOS**

**PROFª. LUCIANA SILVA ZAPPAROLLI**

**ATIVIDADE**

**Complexidade de algoritmos: comportamento assintótico e notações Big O, Omega e Theta.**

**NOME:** GIOVANNA APARECIDA VILLANOVA.

**RA:** 1131392223020.

**SEMESTRE:** 2°.

MAUÁ

2023

# INTRODUÇÃO

A complexidade de algoritmos é um campo fundamental da ciência da computação que estuda o desempenho dos algoritmos em termos de tempo e espaço de execução. É uma medida importante para entender a eficiência dos algoritmos e como eles podem ser melhorados.

# ALGORITMOS

AlgorItmos são um conjunto de instruções computacionais bem definidas que podem receber um ou mais valores, processar esses valores (informações) e fornecer ou não uma resposta, muitas vezes padronizadas e alinhadas conforme as necessidades ou objetivos computacionais definidos. Essas instruções, feitas quase que majoritariamente em linguagens de alto nível, dependem do fim a que serão atribuídas como por exemplo: cálculos matemáticos, análise de padrões, análises de acessos, fluxos de vendas, cálculos de rotas, definição prioridades, checagens de inserção de dados, entre muitos outros. O que diferem esses algoritmos são, geralmente, suas complexidades e porte. Existem tambem algumas categorias técnicas. Vejamos algumas das mais utilizadas:

* Greedy Algorithms;
* Dynamic Programming;
* Divide and conquer;
* Backtracking; • Search and Sorting.

Essas especificidades, que são muitas vezes quase imperceptíveis, são mensuráveis. Como mensurá-las será abordado mais a frente nessa pesquisa.

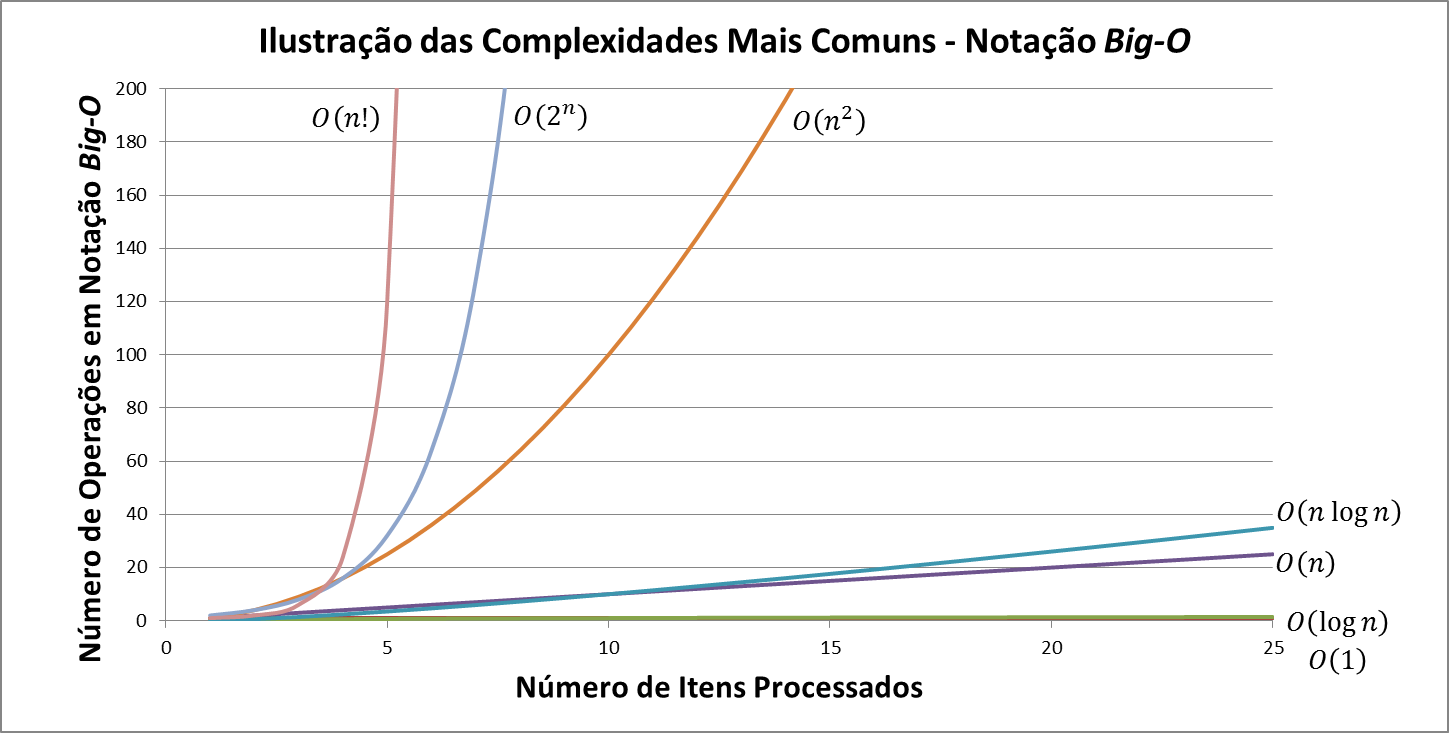
# ANÁLISE DE COMPLEXIDADE E NOTAÇÕES

Dentro das inúmeras técnicas para mensurar essas grandezas, abordaremos mais especificamente

as citadas abaixo:

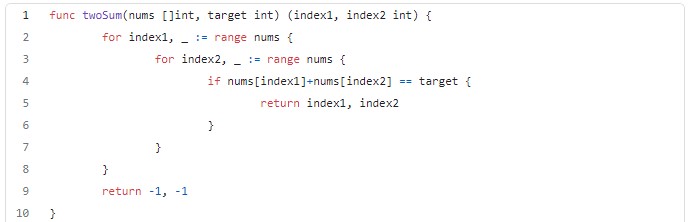
* Big O;
* Omega; • Theta.

Dentre essas técnicas, a mais difundida é a notação Big O (ou Grande O em tradução livre). Essa técnica é usada para medir a relação tempo/espaço máximo que um algorItmo pode levar para ser executado em relação ao tamanho da entrada de dados que foi apontada ao algorItmo base. Vejamos abaixo um gráfico ilustrativo:



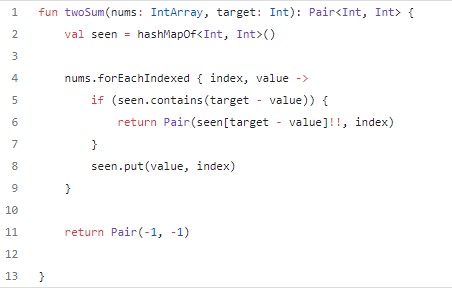
É possível observar no gráfico que temos diversos tipos de algoritmos, com distintos resultados entre eles e a relação Operação/Processo, ou seja, nem sempre o número de processos realizados são executados com a mesma quantidade de operações. Essa, como outras notações, são utilizadas para tentar chegar a um número ótimo de eficiência.

É importante salientar que a linguagem escolhida também interfere na eficiência de um projeto de T.I., por isso é extremamente importante fazer a melhor escolha possível para uma melhor eficiência. Veja o exemplo abaixo de um código em Golang que verifica se existem 2 números cuja soma resulte na entrada específica (alvo) e retorna as posições dos números encontrados:



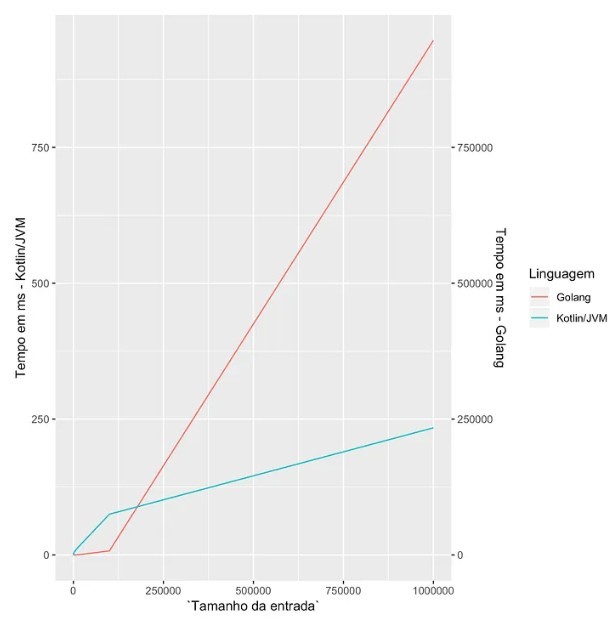
*Figura 1: Algoritmo em Golang*

Vejamos agora a mesma solução, só que na linguagem Kotlin:



*Figura 2: Algoritmo em Kotlin*

Definindo as duas soluções com um array de 1000 posições, o algoritmo em Go se mostrou mais rápido, alguns milissegundos a menos que a solução em Kotlin. Mas será que realmente a solução em Go é a melhor? Analisemos o gráfico a seguir:



*Figura 3: Variação do tempo de execução dos algoritmos de acordo com o tamanho da entrada.*

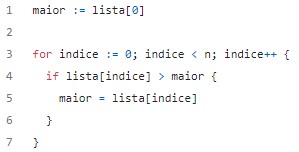
Um melhor algoritmo é aquele que consegue a menor variação de tempo e espaço durante seu processamento, chamamos isso de complexidade. Vemos no gráfico acima que quando esses algorItmos tendem ao infinito, a solução em Kotlin tem o melhor desempenho, mostrando-se uma melhor solução.

A Análise de Complexidade nos ajuda na hora de mensurar/medir qual a velocidade de processamento de uma solução ou programa. Esse processamento pode se dar de maneiras diversificadas, como por exemplo: operações de soma, multiplicação, registros em bases de dados, consultas, montagens de listas para exibição, filtros de pesquisas, etc. Mas o que é sabido que a base fundamental de um programa de computador é a computação.

Entender essas nuances nos permite estudar e entender os funcionamentos dos algoritmos e suas devidas complexidades.

# CONTAGEM DE INSTRUÇÕES

O passo inicial para identificar a complexidade de um algoritmo é saber contar, mesmo que superficialmente, o tanto de instruções dentro dele. Vejamos abaixo um exemplo de uma solução que que encontra o maior elemento dentro de um array com N elementos:



Vamos contar quantas instruções esse algoritmo executa, para ser possível analisar a sua complexidade.



A primeira instrução define o primeiro elemento da lista a uma variável, nesse caso a variável ***maior.***

Em seguida, caso o *array* não esteja vazio, o *loop* será inicializado:



Após o *loop* ser inicializado, temos mais duas intruções:



E ao final da cada *loop,* teremos mais duas instruções: Incremento do contador do laço:



Verificação se já foi percorrido todo o *array:*



Desconsiderando o corpo do laço *for-loop,* podemos chegar a função matemática f(n)=4+2n, é possível prever através dessa equação que o número de instruções mudará conforme a entrada de dados.

# COMPORTAMENTO ASSINTÓTICO

Seria totalmente maçante e improdutivo ficar medindo o número de instruções a cada bloco de código, ainda mais que isso varia muito entre linguagens e compiladores, pensando nisso ficou convencionado que o importante é analisar apenas o fator que mais cresce conforme a entrada de dados para processamento.

Em um exemplo f(n) = 10n + 5, fica claro que o fator 5 é uma constante, invariável. Porem o termo 10 varia conforme n, removendo o termo 10, ficamos com a seguinte função f(n) = n, ou seja, f(n) varia, basicamente, conforme a entrada n, isolando os termos conseguimos analisar assintóticamente a complexidade do algoritmo.

Vejamos alguns exemplos abaixo:

**f(n) = 250**: utilizando a mesma regra, temos **f(n) = 1**, pois 250 está sendo multiplicado por 1; **f(n) = 50n² + 2n + 200:** utilizando a mesma convenção temos **f(n) = n².**

# NOTAÇÃO BIG O

Essa notação foi criada para delimitar assintóticamente o crescimento do algoritmo. Utilizando a convenção citada acima, podemos dizer que um algoritmo cresce na notação ***“Big o O(n)”*** onde O(n) corresponde a f(n), ou seja, no pior dos casos um algoritmo crescera conforme a ordem de “n”.

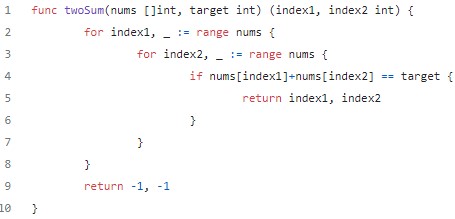
Levando em consideração o tamanho do algoritmo, podemos analisar que ele cresce conforme o número de laços que ele possui, ou seja, um laço f(n) = n, 2 laços f(n) = n², 0 laços f(n) = 1. Mas isso nem sempre é aplicável, pois estamos analisando em função dos laços.

# COMPLEXIDADE DE ESPAÇO

Até aqui analisamos a complexidade em função do tempo de processamento, veremos a partir de agora a complexidade em função do espaço.

Embora a complexidade de espaço ser diferente da complexidade de tempo, ainda sim utilizamos a notação Big O.

Vamos analisar novamente o exemplo Two Sum em Go. Vejamos:



Para cada item do array, todos os outros itens serão percorridos, se o valor da soma dos elementos atuais for igual ao **target** o valor é retornado. Levando em conta os *loops*, no pior dos casos o algoritmo ira percorrer **n\*n** vezes, para informar que existem dois itens que não correspondem ao **target**, denotando então O(n²), e como essa aplicação não demanda espaço fisico na memória, temos sua complexidade de espaço O(1).

# BUSCA BINÁRIA E MERGE SORT

A busca binária é um algoritmo de busca eficiente utilizado para encontrar um valor específico em um conjunto de dados ordenados. A ideia básica da busca binária é dividir o conjunto de dados pela metade repetidamente até que o valor desejado seja encontrado ou seja determinado que ele não está presente.

A complexidade de tempo da busca binária é O(log n), onde n é o número de elementos no conjunto de dados. Isso significa que a quantidade de tempo necessária para encontrar um elemento usando a busca binária aumenta de forma logarítmica com o tamanho do conjunto de dados. Em outras palavras, a busca binária é muito mais rápida do que algoritmos de busca linear para conjuntos de dados grandes.

Além disso, a busca binária é uma técnica muito versátil e pode ser aplicada a diferentes tipos de dados, como números inteiros, números reais, caracteres e objetos. No entanto, é importante notar que a busca binária só funciona em conjuntos de dados ordenados, portanto, se os dados não estiverem ordenados, será necessário primeiro classificá-los antes de aplicar a busca binária.

Em resumo, a busca binária é um algoritmo eficiente de busca com complexidade de tempo O(log n) que pode ser aplicado a diferentes tipos de dados, mas apenas funciona em conjuntos de dados ordenados.

Um exemplo comum de aplicação da busca binária é em sistemas de busca em dicionários ou glossários, onde as palavras estão ordenadas alfabeticamente. Para encontrar uma determinada palavra em um dicionário, podemos utilizar a busca binária da seguinte forma:

Dividir o dicionário ao meio, encontrando a palavra que está exatamente no meio da lista. Se a palavra que procuramos for igual a essa palavra, paramos a busca e retornamos o índice onde ela foi encontrada.

Caso a palavra que procuramos venha antes da palavra no meio da lista, descartamos a metade superior do dicionário e repetimos o processo na metade inferior.

Se a palavra que procuramos for depois da palavra no meio da lista, descartamos a metade inferior do dicionário e repetimos o processo na metade superior.

Continuamos repetindo esses passos até encontrar a palavra ou determinar que ela não está presente no dicionário.

Merge Sort é um algoritmo de ordenação eficiente que utiliza a estratégia "dividir e conquistar". A ideia básica do Merge Sort é dividir a lista de elementos em sub-listas menores e ordená-las recursivamente. Em seguida, combina essas sub-listas menores em uma única lista ordenada.

O Merge Sort possui uma complexidade de tempo de O(n log n), o que o torna muito mais rápido do que algoritmos de ordenação mais simples, como o Bubble Sort ou o Selection Sort, especialmente para conjuntos de dados maiores.

O processo de ordenação com Merge Sort ocorre em três etapas principais:

Divisão: a lista de elementos é dividida em duas sub-listas de tamanhos aproximadamente iguais. Esse processo é repetido recursivamente até que cada sub-lista contenha apenas um elemento.

Conquista: as sub-listas são ordenadas recursivamente usando o Merge Sort.

Combinação: as sub-listas ordenadas são combinadas em uma única lista ordenada.

A combinação é feita comparando os elementos das sub-listas e colocando-os em ordem crescente em uma nova lista. Esse processo é repetido até que todas as sub-listas sejam combinadas em uma única lista ordenada.

Por exemplo, se quisermos ordenar a lista [38, 27, 43, 3, 9, 82, 10], o processo de ordenação com Merge Sort seria o seguinte:

Dividimos a lista ao meio e ordenamos recursivamente cada metade:

[38, 27, 43, 3] -> [38, 27] -> [38], [27] -> [27, 38]-> [43, 3] -> [43], [3] -> [3, 43]

[9, 82, 10] -> [9], [82, 10] -> [82], [10] -> [10, 82]-> [9, 10, 82]

Agora, combinamos as duas sub-listas ordenadas em uma única lista ordenada:

[27, 38, 3, 43] -> [3, 27, 38, 43] [9, 10, 82] -> [9, 10, 82]

Combinamos as duas sub-listas ordenadas em uma única lista ordenada:

[3, 27, 38, 43, 9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82]

Em resumo, o Merge Sort é um algoritmo de ordenação eficiente com complexidade de tempo O(n log n) que utiliza a estratégia "dividir e conquistar" para ordenar listas de elementos. Ele é capaz de lidar com conjuntos de dados grandes e é um dos algoritmos de ordenação mais utilizados em programação.

# NOTAÇÃO ÔMEGA

A notação omega é uma notação utilizada em análise de complexidade de algoritmos para descrever o limite inferior de tempo de execução de um algoritmo. Ela representa o melhor caso possível de desempenho de um algoritmo em um determinado problema.

Formalmente, dizemos que uma função f(n) é de ordem omega(g(n)), denotada como f(n) = Omega(g(n)), se existem constantes positivas c e n0 tais que para todo n >= n0, temos f(n) >= c \* g(n).

Isso significa que, para um dado tamanho de entrada n, a função f(n) é pelo menos tão grande quanto uma constante, vezes a função g(n). Em outras palavras, a função g(n) é um limite inferior assintótico para a função f(n).

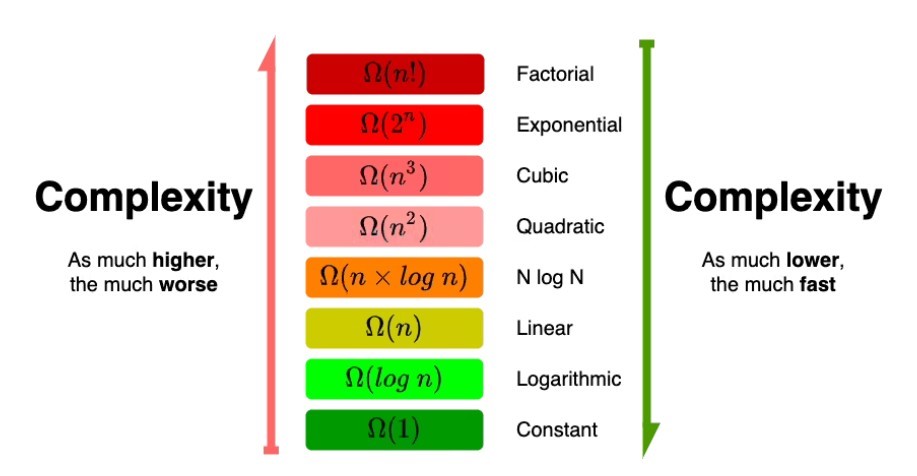
Assim como as notações big O e theta, a notação omega é usada para descrever a complexidade de tempo de algoritmos em relação ao tamanho da entrada. No entanto, enquanto a notação big O descreve um limite superior para o tempo de execução de um algoritmo e a notação theta descreve um limite superior e inferior, a notação omega descreve apenas um limite inferior.

Por exemplo, se um algoritmo de busca em uma lista ordenada sempre encontra o elemento que procura em no máximo log2(n) comparações, podemos dizer que o tempo de execução desse algoritmo é de ordem omega(log n), já que é impossível fazer menos do que log n comparações para encontrar um elemento em uma lista ordenada.

Assim como a notação Big O, a notação Ômega trabalha a função em relação limite a outra, mas nesse caso, em relação ao limite inferior, logo podemos interpretar da seguinte maneira: f(n) cresce minimamente em ordem de g(n), levando em consideração todo e qualquer valor de entrada de dados.



Abaixo temos uma figura que mostra o escalonamento de complexidade de algoritmo com a notação Ômega. Vejamos:

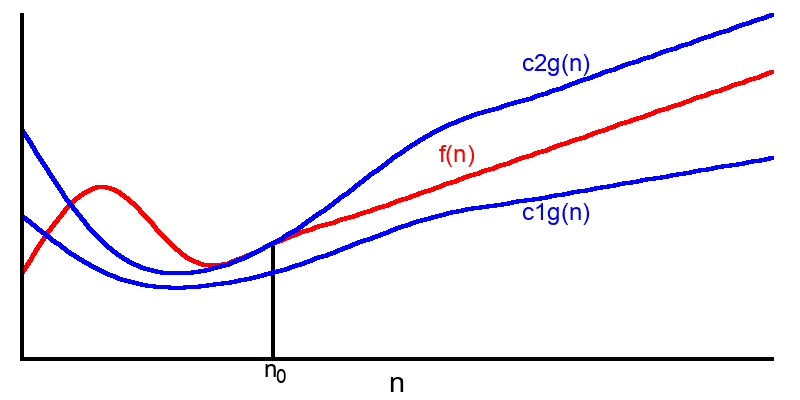


Quanto mais alta a complexidade, maior o tempo de processamento. Inversamente verdadeira também.

Em resumo, a notação omega é uma notação usada em análise de complexidade para descrever o limite inferior de tempo de execução de um algoritmo em relação ao tamanho da entrada. É uma ferramenta útil para entender as limitações intrínsecas de um algoritmo em um determinado problema.

# NOTAÇÃO THETA

Quando falamos de notação THETA (Θ), diferentemente das anteriores, falamos da união dos dois conceitos anteriores, sendo usados simultaneamente, ou seja, quando analisamos a complexidade de um algorItmo em Θ, analisamos seu comportamento assintótico em relação ao seu limite inferior e superior. Vamos analisar a figura abaixo:



Nessa figura temos um algoritmo representado pela função f(n), sendo analisado temos ele em Θ(g(n)), sendo assim a função g(n) limita a função f(n) tanto superior quanto inferiormente. Em suma, a função Θ(g(n)) se trata da intersecção entre O(g(n)) e Ω(g(n)).

Tendo em vista o exposto, e assumindo que a notação theta é uma intersecção, e ao invés de utilizarmos uma constante utilizarmos duas, podemos deduzir a seguinte equação:



Assim, podemos ler que f(n) cresce no mínimo como c1g(n) e no máximo como c2g(n) para qualquer valor de n sendo n > n0. Ou seja, f(n) é O(g(n)) e Ω(g(n)). Assim, dizemos que f(n) = Θ(g(n)).

Adotando essa definição, podemos dizer que quando o comportamento assintótico de um algoritmo tem complexidade de tempo e espaço em Θ, assumimos que se trata de um algorItmo com limites firmes, tanto acima como abaixo, se tratando de um algoritmo

Adotando a complexidade theta definimos, geralmente, um algoritmo firme, robusto e confiável, pois podemos inferir tanto em seus limites inferiores quanto superiores, tendo clareza da sua complexidade tanto de tempo, quanto espaço.

Em resumo, a complexidade de algoritmos é uma área importante da ciência da computação que se concentra em entender o desempenho dos algoritmos em termos de tempo e espaço de execução. É uma ferramenta essencial para a construção de algoritmos eficientes e melhorias de desempenho.

# Bibliografia

Introdução à Complexidade de Algoritmos | by Wilder Pereira | Nagoya Foundation | Medium – data do acesso 04/04/2023 13:57

[https://medium.com/nagoya-foundation/introdu%C3%A7%C3%A3o-%C3%A0-complexidade-de-algoritmos4a9c237e4ecc](https://medium.com/nagoya-foundation/introdu%C3%A7%C3%A3o-%C3%A0-complexidade-de-algoritmos-4a9c237e4ecc)

Introdução a notação Theta, acesso em 17/04/2023 as 09:12 - <https://algol.dev/notacao-theta/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). Algorithms (1st ed.). McGraw-Hill Higher Education.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.

Goodrich, M. T., & Tamassia, R. (2015). Algorithm Design and Applications (1st ed.). Wiley.

GeeksforGeeks. (n.d.). Complexity Analysis of Algorithms. Retrieved from

https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/

Khan Academy. (n.d.). Asymptotic Notation. Retrieved data do acesso 04/04/2023 13:57 https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation