Projeto e Análise de Algoritmos (2025-1)

Flávio L. C. de Moura*

23 de abril de 2025

1 A complexidade de algoritmos

Como avaliar o tempo de execução de um algoritmo? Antes de respondermos esta pergunta, devemos fazer outra: por que avaliar o tempo de execução de um algoritmo? Em outras palavras, qual a necessidade de estudar análise de algoritmos? Como veremos, diversas razões para isso. Inicialmente, observe que um algoritmo tem como objetivo resolver um problema específico, como ordenação, busca ou cálculo de um determinado valor. No entanto, para um mesmo problema, podem existir múltiplos algoritmos diferentes. Por exemplo, no caso da ordenação há diversos algoritmos conhecidos, como *Insertion Sort*, *Merge Sort*, *Selection Sort*, *Heapsort*, *Quicksort*/, etc. Qual deles escolher? O objetivo da análise de algoritmos é fornecer as ferramentas teóricas que nos permitam responder essa pergunta com base em critérios matemáticos, garantindo a escolha da solução mais eficiente para cada situação.

O que significa dizer que um algoritmo é eficiente? Podemos analisar a eficiência de um algoritmo de duas formas: eficiência temporal e eficiência espacial. No primeiro caso, estamos interessados no tempo de execução do algoritmo, enquanto que no segundo caso, queremos analisar a quantidade de espaço extra (memória) que é utilizado pelo algoritmo durante sua execução [2, 1, 4, 3]. A forma de determinar a eficiência de um algoritmo deve permitir a comparação de algoritmos distintos que resolvam o mesmo problema. Inicialmente, poderíamos pensar em utilizar o tempo de execução de um programa que implementa um algoritmo, mas esta não é uma boa medida porque depende tanto do computador (*hardware*) quanto da implementação (*software*). Precisamos de um método que nos informe sobre a eficiência do algoritmo independentemente do computador em que ele venha a ser implementado, da linguagem de programação e do estilo de programação utilizados. O método deve ser preciso e geral de forma que possa ser utilizado para diversos algoritmos e aplicações.

Antes de apresentarmos o ferramental matemático que utilizaremos, precisamos estabelecer alguns critérios:

- **Independência de *hardware*:** A escolha do algoritmo deve ser baseada em sua eficiência teórica, não no desempenho de um hardware específico;

*flaviomoura@unb.br

- **Independência de *software*:** A análise deve ser independente da linguagem de programação ou das habilidades do programador;
- **Princípio da invariância:** Diferentes implementações de um mesmo algoritmo não devem variar em eficiência além de um fator constante multiplicativo.

Uma possibilidade é contar todas as operações realizadas pelo algoritmo, mas veremos que esta abordagem possui alguns problemas, além de ser muito trabalhosa.

Busca Sequencial

Considere o problema de buscar um elemento em um vetor arbitrário. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n - 1]$ contendo n elementos e o valor x procurado, e faz a busca sequencial de x em A : se A possui uma ocorrência de x então o algoritmo retorna a posição $0 \leq i \leq n - 1$ da primeira ocorrência encontrada. Caso contrário, o algoritmo retorna o valor -1.

Algorithm 1: SequentialSearch($A[0..n - 1], x$)

```

1  $i \leftarrow 0$ ;
2 while  $i < n$  and  $A[i] \neq x$  do
3   |  $i \leftarrow i + 1$ ;
4 end
5 if  $i < n$  then
6   | return  $i$ ;
7 else
8   | return -1
9 end

```

Observe que a execução do algoritmo não demanda espaço adicional, ou seja, o espaço utilizado para a sua execução é o espaço alocado para armazenar o vetor A e nada mais. Neste caso, dizemos que a complexidade em espaço do algoritmo SequentialSearch é constante. Uma complexidade, seja em tempo ou espaço, constante é a melhor situação que podemos ter, ou seja, a mais eficiente possível. As classes básicas de eficiência que utilizaremos para analisar algoritmos são listadas a seguir em ordem crescente de complexidade em função do tamanho n da entrada[2]:

Classe	Nome
1	constante
$\log n$	logarítmica
n	linear
$n \cdot \log n$	linearítmica
n^2	quadrática
n^3	cúbica
n^k	polinomial ($k \geq 1$ e finito)
a^n	exponencial ($a \geq 2$)
$n!$	fatorial

A análise da complexidade de tempo do algoritmo SequentialSearch não é tão imediata quanto a análise feita para a complexidade de espaço, ainda que seja simples. Podemos começar com a seguinte pergunta: qual o custo de execução de cada linha do algoritmo SequentialSort? A linha 1 faz uma atribuição, cujo custo não depende do tamanho n do vetor A , e portanto é razoável dizer que este custo é constante, digamos c_1 , uma constante positiva. Observe que esta constante não depende do parâmetro n , mas do computador e da linguagem de programação. As linhas 2-4 constituem um laço cujo corpo contém apenas uma atribuição. Ainda que o custo da linha 3 possa ser o mesmo da linha 1, vamos denotá-lo pela constante positiva c_3 . Quantas vezes a linha 3 é executada? Isto depende tanto do vetor A quanto da chave x . De fato, se x ocorre na primeira posição de A , isto é, se $A[0]$ é igual a x então a condição do laço é executada uma única vez, mas a linha 3 não é executada nenhuma vez independente de existirem outras ocorrências de x em A . Esta é a situação constitui o melhor caso possível, e por isso é chamada de *análise do melhor caso*. Se $A[0] \neq x$ e x ocorre na segunda posição de A então a linha 3 é executada uma única vez, enquanto que a linha 2 é executada duas vezes. Em geral, observe que a linha que define um laço é sempre executada uma vez a mais do que as linhas que compõem o seu corpo. Por fim, se x não ocorre no vetor A então a linha 2 será executada $n + 1$ vezes enquanto que a linha 3 será executada n vezes. Esta situação vai configurar a *análise do pior caso*. Por fim, o condicional da linha 5 será executado uma única vez a um custo constante, digamos c_5 , e apenas uma das linhas 6 ou 8 será executada uma única vez. Juntando todas estas informações podemos então dividir a análise em 2 casos:

Análise do melhor caso na busca sequencial

Como vimos anteriormente, o melhor caso ocorre quando o elemento procurado ocorre na primeira posição do vetor A . Os custos associados por linha nesta situação são apresentados na seguinte tabela:

Linha	Custo	Observação
1	c_1	
2	c_2	
3	0	não é executada
5	c_5	
6	c_6	
8	0	não é executada
Total	$c_1 + c_2 + c_5 + c_6$	

Denotando por $T_b(n)$ o custo no melhor caso (*best case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_b(n) = c_1 + c_2 + c_5 + c_6$. Neste caso dizemos que o custo da busca sequencial é constante em função do tamanho n da entrada.

Análise do pior caso na busca sequencial

Agora vamos compilar as informações discutidas anteriormente considerando que o laço da linha 2 é executado o maior número de vezes possível, o que acontece quando o elemento procurado não

ocorre no vetor:

Linha	Custo
1	c_1
2	$c_2 \cdot (n + 1)$
3	$c_3 \cdot n$
5	c_5
6	0
8	c_8
Total	$c_1 + c_2 \cdot (n + 1) + c_3 \cdot n + c_5 + c_8$

Denotando por $T_w(n)$ o custo no pior caso (*worst case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_w(n) = c_1 + c_2 \cdot (n + 1) + c_3 \cdot n + c_5 + c_8$. Neste caso dizemos que o custo da busca sequencial é linear em função do tamanho n da entrada. Antes de refinarmos a análise e apresentarmos as definições precisas das análises de melhor e pior caso, vejamos um outro exemplo considerando agora o problema da ordenação de um vetor.

Vejamos um outro exemplo antes de refinarmos nossa análise.

Ordenação por inserção

Nesta seção estamos interessados em ordenar $n > 0$ números naturais em ordem crescente. Suponha que estes números estão armazenados no vetor $A[0..n - 1]$. Então, ao final do processo queremos obter uma permutação de $A[0..n - 1]$, digamos $A'[0..n - 1]$ tal que $A'[i - 1] \leq A'[i]$, para todo $1 \leq i < n$. Estudaremos diversas formas distintas de abordar este problema nas próximas seções, mas aqui vamos iniciar com o algoritmo de ordenação por inserção (InsertionSort), cujo pseudocódigo é dado a seguir:

Algorithm 2: InsertionSort($A[0..n - 1]$)

```
1 for  $j = 1$  to  $n - 1$  do
2    $key \leftarrow A[j]$ ;
3    $i \leftarrow j - 1$ ;
4   while  $i \geq 0$  and  $A[i] > key$  do
5      $A[i + 1] \leftarrow A[i]$ ;
6      $i \leftarrow i - 1$ ;
7   end
8    $A[i + 1] \leftarrow key$ ;
9 end
```

Faremos uma análise da complexidade do algoritmo de ordenação por inserção semelhante análoga à feita para a busca sequencial. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada que neste caso é o número n de elementos a serem ordenados. Novamente assumiremos que cada linha do pseudocódigo é executada em tempo constante, mas este tempo pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde

ao tempo de execução da i -ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O laço **for** da linha 1 é executado n vezes, enquanto que o corpo do laço é executado $n - 1$ vezes, uma vez para cada $j = 1, \dots, n - 1$. Denotaremos por t_j o número de vezes que o teste do laço **while** da linha 4 é executado, de forma que temos o seguinte custo por linha:

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1 \cdot n$
2	c_2	$n - 1$	$c_2 \cdot (n - 1)$
3	c_3	$n - 1$	$c_3 \cdot (n - 1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4 \cdot \sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1} (t_j - 1)$	$c_5 \cdot \sum_{j=1}^{n-1} (t_j - 1)$
6	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$	$c_6 \cdot \sum_{j=1}^{n-1} (t_j - 1)$
8	c_8	$n - 1$	$c_8 \cdot (n - 1)$

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot \sum_{j=2}^n t_j + c_5 \cdot \sum_{j=2}^n (t_j - 1) + c_6 \cdot \sum_{j=2}^n (t_j - 1) + c_8 \cdot (n - 1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, um vetor que tenha mais elementos a serem reposicionados terá um custo maior para ser ordenado. Portanto, a análise do melhor caso se dá quando o vetor já estiver ordenado pois $t_j = 1$, para todo $2 \leq j \leq n$:

$$\begin{aligned} T_b(n) &= c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot (n - 1) + c_8 \cdot (n - 1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8) \cdot n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

ou seja, uma função linear de n . Por outro lado, a análise do pior caso se dá quando o vetor estiver ordenado decrescentemente pois $t_j = j$ (por que?), e portanto

$$T_w(n) = c_1 \cdot n + (c_2 + c_3 + c_8) \cdot (n - 1) + c_4 \cdot \left(\frac{(n-1) \cdot n}{2}\right) + (c_5 + c_6) \cdot \left(\frac{(n-2) \cdot (n-1)}{2}\right)$$

ou seja, uma função quadrática de n .

A forma de análise feita para InsertionSort acima, assim como para SequentialSearch na seção anterior, apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador, da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades, e fazer uma análise que seja independente destes aspectos, consiste na utilização de uma notação adequada, a *notação assintótica*, que considera o comportamento de funções no limite, isto é, para valores suficientemente grandes do parâmetro n . A ideia é que possamos pegar uma função como $T_w(n) = c_1 + c_2 \cdot n + c_3 \cdot (n - 1) + c_6 + c_8$ que expressa o custo no pior caso do algoritmo de busca sequencial, e dizer que ela cresce como n , sem a necessidade

de considerar as constantes. Faremos isto considerando o conjunto das funções que são limitadas superiormente por um múltiplo constante de n . Observe que podemos facilmente construir uma cota superior para a função $T_w(n)$ da seguinte forma $T_w(n) = c_1 + c_2 \cdot n + c_3 \cdot (n - 1) + c_6 + c_8 \leq c_1 \cdot n + c_2 \cdot n + c_3 \cdot n - c_3 + c_6 \cdot n + c_8 \cdot n \leq (c_1 + c_2 + c_3 + c_6 + c_8) \cdot n \leq c \cdot n$ para qualquer constante $c \geq c_1 + c_2 + c_3 + c_6 + c_8$ e $n \geq 1$. Neste caso, dizemos que a função $T_w(n)$ é $O(n)$, ou seja, que $T_w(n)$ é de ordem n . Formalmente, temos a seguinte definição para o conjunto $O(g(n))$ que contém todas as funções que são da ordem de $g(n)$:

Definição 1.1. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.\}$*

Observe que uma função $f(n)$ pode estar em $O(g(n))$ mesmo que $f(n) > g(n), \forall n$. O ponto importante é que $f(n)$ tem que ser limitada por um múltiplo constante de $g(n)$. A relação entre $f(n)$ e $g(n)$ para valores pequenos de n também é desconsiderada. Intuitivamente, os termos de menor ordem de uma função assintoticamente positiva podem ser ignorados na determinação da cota superior porque são insignificantes para valores grandes do parâmetro n . Assim, quando n é grande qualquer porção ou fração do termo de maior ordem é suficiente para dominar os termos de menor ordem.

Normalmente, escrevemos $T(n) \in O(n^2)$ para dizer que $T(n)$ é $O(n^2)$ já que $O(n^2)$ é um conjunto. No entanto, é comum encontramos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$ (Ver [2]). A conveniência do uso da igualdade será vista posteriormente, mas o importante aqui é entender que esta igualdade é unidirecional, e portanto não pode ser confundida com a igualdade tradicional. Por exemplo, escrevemos $T(n) = O(n^2)$, mas $O(n^2) = T(n)$ não é correto. O número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece: por exemplo, na expressão $\sum_{i=1}^n O(i)$ contém apenas uma função anônima (a função que tem parâmetro i), e portanto esta expressão não é o mesmo que $O(1) + O(2) + \dots + O(n)$ (que não possui uma interpretação clara). A notação assintótica também pode aparecer do lado esquerdo de uma equação: $2n^2 + O(n) = O(n^2)$. Neste caso, independentemente da forma como as funções anônimas são escolhidas do lado esquerdo da equação, existe uma forma de escolher funções anônimas do lado direito da equação de forma que a equação se verifique. No caso do exemplo acima, temos que para qualquer $f(n) = O(n)$, existe uma função $g(n) = O(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$.

Equações também podem ser encadeadas como em $2n^2 + 3n + 1 = 2n^2 + O(n) = O(n^2)$, e podem ser interpretadas separadamente de acordo com as regras anteriores. Assim, a primeira equação nos diz que existe alguma função $f(n) = O(n)$ para a qual a equação se verifica para todo n . A segunda equação nos diz que para toda função $g(n) = O(n)$, existe uma função $h(n) = O(n^2)$ tal que a equação se verifica para todo n . Este encadeamento é transitivo, ou seja, podemos concluir que $2n^2 + 3n + 1 = O(n^2)$.

O lema a seguir nos permite utilizar limites para utilizar a notação assintótica:

Lema 1.2. Uma função $f(n) = O(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, incluindo o caso em que $c = 0$.

Demonstração. Exercício. □

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lceil \lg n \rceil}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lg n} \leq 2^{\lceil \lg n \rceil + 1} = 2 \cdot 2^{\lceil \lg n \rceil} = 2 \cdot g(n), \forall n$. No entanto, o limite $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 1.3. 1. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = O(g(n))$ e $g(n) = O(f(n))$;

2. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $g(n) \neq O(f(n))$;

3. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$ então $f(n) \neq O(g(n))$, mas $g(n) = O(f(n))$.

Demonstração. Exercício. □

No caso de InsertionSort, a análise do pior caso nos dá a função

$$T_w(n) = c_1 \cdot n + (c_2 + c_3 + c_8) \cdot (n - 1) + c_4 \cdot \binom{(n-1) \cdot n}{2} + (c_5 + c_6) \cdot \binom{(n-2) \cdot (n-1)}{2}$$

que é $O(n^2)$.

Como exercício, mostre em detalhes que a complexidade do pior caso de InsertionSort é $O(n^2)$.

Assim, considerando as expressões (ou polinômios) construídas(os) até agora, observamos que a classe de complexidade é obtida considerando-se o monômio de maior grau sem levar em conta o coeficiente. Portanto, a construção do polinômio a partir do custo de cada linha do algoritmo não é uma estratégia eficiente porque no final consideraremos apenas a parcela mais significativa, ou seja, o monômio de maior grau. Vamos então buscar diretamente a parte do algoritmo que nos dá este monômio de maior grau. Observando a Tabela 9 concluímos que o termo quadrático vem da linha 4, mais precisamente da comparação $A[i] > key$ que é executada em cada iteração do laço **for**. Então podemos fazer uma análise bem mais direta do que a feita anteriormente para chegarmos à mesma conclusão. Como durante a i -ésima iteração do laço **for**, a linha 4 é executada i vezes, temos:

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

A análise do melhor caso também pode ser feita da mesma forma considerando que a cada iteração do laço **for**, a linha 4 é executada uma única vez:

$$T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

Da mesma forma, na busca sequencial o custo linear do pior caso pode ser obtido calculando diretamente o número de comparações feitas na linha 2. A notação O nos dá uma cota superior para o custo de execução de algoritmos, mas ela também pode ser utilizada para estabelecer uma cota para a complexidade de espaço utilizado durante a execução de um algoritmo. Tanto a busca sequencial quanto o algoritmo de ordenação por inserção não necessitam de espaço adicional de armazenamento, e portanto, em ambos os casos a complexidade é constante, ou seja, é igual a $O(1)$. Dizemos que algoritmos de ordenação que não demandam espaço adicional fazem a ordenação *in place*. Posteriormente estudaremos algoritmos que necessitam de espaço adicional. Na tabela abaixo, resumimos as análises feitas até agora:

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$

Uma ferramenta bastante útil na análise assintótica é conhecida como *regra do máximo*:

$$O(f(n) + g(n)) = O(\max(f(n), g(n))) \quad (1)$$

Depois de alguns exercícios, e de apresentarmos mais alguns detalhes sobre a notação assintótica, estudaremos um pouco da chamada análise do caso médio. A análise do melhor caso nos dá uma ideia de situações específicas em que o algoritmo tem a melhor performance possível, mas a análise do melhor caso não costuma ser muito informativa e normalmente não é relevante. A análise do pior caso, por outro lado, tem bastante relevância e será explorada exhaustivamente nas próximas seções. Ela é importante porque nos fornece o pior cenário possível para o algoritmo. Com isto sabemos que o algoritmo não pode ter um comportamento menos eficiente do que o apresentado pela análise do pior caso. No entanto, esta análise pode ser excessivamente pessimista considerando uma situação mais realista. Por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A análise do caso médio pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis, o que não corresponde à média entre as análises do melhor e pior casos.

Por fim, é importante ter em mente que a notação assintótica nos permite analisar a taxa de crescimento, ou ordem de crescimento do tempo de execução de um algoritmo, e portanto as simplificações feitas na obtenção da cota superior não devem ser esquecidas em situações práticas. Por exemplo, considere dois algoritmos A e B com complexidades, respectivamente, iguais a $O(n^2)$ e $O(n^3)$. Qual dos dois algoritmos é mais eficiente? Para valores grandes de n certamente o algoritmo

A é mais eficiente, mas devemos levar em consideração que no cálculo destas classes de complexidade diversas constantes foram ignoradas. Se soubéssemos, por exemplo, que o algoritmo A realiza $100.n^2$ operações, enquanto que o algoritmo B realiza $5.n^3$ operações para resolver o mesmo problema, então agora sabemos que para $n < 20$ o algoritmo B é mais eficiente.

Exercício 1.4. Considere os pseudocódigos a seguir e complete a tabela:

Algorithm 3: BubbleSort($A[0..n - 1]$)

```

1 for  $i = 0$  to  $n - 2$  do
2   for  $j = 0$  to  $n - 2 - i$  do
3     if  $A[j + 1] < A[j]$  then
4       swap  $A[j]$  and  $A[j + 1]$ ;
5     end
6   end
7 end
```

Algorithm 4: SelectionSort($A[0..n - 1]$)

```

1 for  $i = 0$  to  $n - 2$  do
2    $min \leftarrow i$ ;
3   for  $j = i + 1$  to  $n - 1$  do
4     if  $A[j] < A[min]$  then
5        $min \leftarrow j$ ;
6     end
7   end
8   swap  $A[i]$  and  $A[min]$ ;
9 end
```

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort			
Selection sort			

Exercício 1.5. Mostre que $n = O(n^2)$.

Exercício 1.6. Mostre que $100n + 5 = O(n^2)$.

Exercício 1.7. Mostre que $\frac{n(n-1)}{2} = O(n^2)$.

Exercício 1.8. Mostre que $n^3 \neq O(n^2)$.

Exercício 1.9. *Sejam $f(n), g(n)$ e $h(n)$ funções dos inteiros não-negativos nos reais positivos. Mostre que se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$.*

Assim, como $O(g(n))$ estabelece uma cota superior para funções, o conjunto $\Omega(g(n))$ estabelece uma cota inferior para as funções:

Definição 1.10. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $\Omega(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c \cdot g(n) \leq f(n), \forall n \geq n_0$. Alternativamente, $\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$*

Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n , o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor caso. Por exemplo, no melhor caso, o algoritmo InsertionSort é $\Omega(n)$, e portanto, o tempo de execução do algoritmo InsertionSort está entre $\Omega(n)$ e $O(n^2)$. A definição alternativa para o conjunto $\Omega(g(n))$ em termos de limites é dada pelo lema a seguir:

Lema 1.11. *Uma função $f(n) = \Omega(g)$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$, incluindo o caso em que o limite é igual a ∞ .*

Demonstração. Exercício. □

A forma mais precisa de expressar o comportamento assintótico de um algoritmo é fornecendo cotas superiores e inferiores ao mesmo tempo. No parágrafo anterior, apresentamos uma cota superior e uma cota inferior para o algoritmo InsertionSort. No entanto, estas cotas são de classes diferentes, o conjunto $\Theta(g(n))$, definido a seguir, é utilizado quando ambas as cotas são da mesma classe.

Definição 1.12. *Seja g uma função dos inteiros não-negativos nos reais positivos. Então $\Theta(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem constantes reais positivas c_1 e c_2 , e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$. Alternativamente, $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.\}$*

Como qualquer constante pode ser vista como um polinômio de grau 0, podemos representar funções constantes como $\Theta(n^0)$, ou simplesmente, $\Theta(1)$. O lema a seguir apresenta uma caracterização do conjunto $\Theta(g(n))$ em termos de limite:

Lema 1.13. Uma função $f(n) = \Theta(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$, para alguma constante $0 < c < \infty$.

Demonstração. Exercício. □

Teorema 1.14. 1. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;

2. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $f(n) \neq \Theta(g(n))$;

3. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$ então $f(n) = \Omega(g(n))$, mas $f(n) \neq O(g(n))$.

Demonstração. Exercício. □

Exercício 1.15. Prove que $\sum_{i=1}^n i^k = \Theta(n^{k+1})$ para qualquer inteiro $k \geq 0$ fixado.

Teorema 1.16. Dadas funções $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Demonstração. Exercício. □

Lema 1.17. 1. $f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;

2. Se $f(n) = \Theta(g(n))$ então $g(n) = \Theta(f(n))$;

3. Θ define uma relação de equivalência sobre as funções. Cada conjunto $\Theta(f(n))$ é uma classe de equivalência que chamamos de classe de complexidade;

4. $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$;

5. $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$;

Definição 1.18. Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $o(g(n)) = \{f(n) : \text{para qualquer constante positiva } c, \text{ existe uma constante positiva } n_0 \text{ tal que } 0 \leq f(n) < c \cdot g(n), \forall n \geq n_0.\}$

Lema 1.19. Uma função $f(n) = o(g)$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Definição 1.20. Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $\omega(g(n)) = \{f(n) : \text{para qualquer constante positiva } c, \text{ existe uma constante positiva } n_0 \text{ tal que } 0 \leq c \cdot g(n) < f(n), \forall n \geq n_0.\}$

Lema 1.21. Uma função $f(n) = \omega(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, se este limite existir.

Lema 1.22. Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$, ou seja, a notação O é transitiva. Também são transitivos Ω, Θ, o e ω .

Teorema 1.23. $\lg n = o(n^\alpha), \forall \alpha > 0$. Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de n (incluindo potências fracionárias)

Teorema 1.24. $n^k = o(2^n), \forall k > 0$. Ou seja, potências de n crescem mais lentamente que a função exponencial 2^n . Mais ainda, potências de n crescem mais lentamente do que qualquer função exponencial $c^n, c > 1$.

Exercício 1.25. Mostre que $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 1.26. Mostre que $6n^3 \neq \Theta(n^2)$.

Exercício 1.27. Sejam $f(n), g(n)$ e $h(n)$ funções não-negativas tais que $f(n) = O(h(n))$ e $g(n) = O(h(n))$. Prove que $f(n) + g(n) = O(h(n))$.

Referências

- [1] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.
- [3] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.
- [4] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.