

Parte II

Algoritmos

Nesta seção vamos utilizar os temas abordados nos capítulos anteriores para analisar algoritmos. Os computadores e seus algoritmos estão presentes na maioria das atividades quotidianas, utilizamos computadores para fazer compras, transações bancárias, etc. Os carros, aviões e equipamentos hospitalares modernos também possuem diversos sistemas embarcados. A medida em que estes equipamentos de tornam mais comuns em nosso dia a dia, aumenta também o seu poder de processamento e de armazenamento. Diante desta realidade é natural perguntar: por que analisar algoritmos? Inicialmente porque precisamos garantir que são corretos. Mas o que significa dizer que um algoritmo é correto? Intuitivamente, significa que o algoritmo sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se AlgOrd é um algoritmo de ordenação de inteiros, então espera-se que para qualquer vetor de inteiros A dado, AlgOrd(A) retorne uma permutação de A que esteja ordenada. Isto significa que as respostas dadas pelo algoritmo são corretas para todas as entradas possíveis, e que estas respostas são geradas em tempo finito. Como veremos, em alguns casos a prova de correção é simples, mas em geral esta é uma tarefa complexa. Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática estudada no capítulo anterior.

Adicionalmente, precisamos analisar a eficiência destes algoritmos. Mas não poderíamos simplesmente migrar para um computador mais potente e com mais capacidade de armazenamento quando fosse necessário? A resposta é não. Ainda que tivéssemos uma capacidade infinita de processamento e/ou de armazenamento, a análise da eficiência seria necessária. De fato, não faz sentido ter que esperar horas para a finalização de um processamento se for possível fazê-lo de forma mais eficiente em apenas alguns segundos, ou utilizar uma quantidade gigantesca de memória sem necessidade. Estudaremos diversos problemas ao longo deste curso, e veremos situações em que uma abordagem ingênua (força bruta) requer um número exponencial de operações, enquanto que uma abordagem cuidadosa pode diminuir substancialmente o número de operações necessárias para resolver o mesmo problema. Ao analisarmos a eficiência dos algoritmos também faremos o uso de diversas ferramentas matemáticas (somatórios, conjuntos, funções, matrizes, etc). O apêndice VIII do livro [11] pode ser usado para revisar estes temas. Já a análise da complexidade de tempo e/ou complexidade de espaço de um algoritmo consiste no estudo sistemático do número de operações realizadas, ou espaço extra demandado, durante a execução do algoritmo.

Capítulo 5

Análise Assintótica

O ponto de partida do estudo que faremos consiste em determinar a quantidade de recursos demandados por um algoritmo em função do tamanho das instâncias, e neste contexto, é natural esperar que quanto maior for a instância a ser resolvida, maior também será a quantidade de recursos demandados. Os recursos que estamos interessados em investigar são basicamente o tempo de computação e o espaço de armazenamento (ou memória). Assim, considerando novamente o contexto de ordenação de listas (ou vetores), quanto maior for a lista a ser ordenada (instância), maior será o tempo (recurso) demandado. Isto sugere então que para o problema de ordenação, o tamanho das instâncias seja justamente o número de elementos da lista (ou vetor) a ser ordenado.

O parâmetro n que denota o tamanho da entrada também precisa ser fornecido a partir de uma medida adequada para que possamos fazer uma análise concisa. Para o caso de ordenação de uma lista (ou vetor), vimos que o número de elementos da lista representa uma medida adequada, e a tabela abaixo apresenta outros exemplos:

Problema	Tamanho da entrada
Busca em um vetor	Tamanho do vetor
Multiplicação de duas matrizes	Dimensão das matrizes
Busca em grafos	Tamanho da representação do grafo ¹

O modelo computacional que utilizaremos é o de uma máquina de acesso aleatório (*random-access machine* - RAM) com um processador, e devemos lembrar que nossos algoritmos serão implementados como programas neste modelo, onde as instruções são executadas sequencialmente, sem operações concorrentes[11, 22, 20, 5, 7]. As instruções no modelo RAM são as encontradas em computadores reais:

- aritmética (soma, subtração, multiplicação, divisão, resto, piso e teto);
- movimento de dados (*load*, *store*, *copy*);
- controle (ramos condicionais e não-condicionais, chamadas a subprocedimentos e retorno).

Assumiremos que cada uma destas instruções é executada em tempo constante. Além disto, os tipos abstratos de dados deste modelo são os números inteiros e números em ponto flutuante.

Vejam agora os fundamentos para a análise da eficiência de algoritmos. 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,

¹O tamanho da representação normalmente é determinado a partir do número de vértices e número de arestas do grafo.

enquanto que no segundo caso, queremos analisar a quantidade de espaço extra (memória) que é utilizado pelo algoritmo durante sua execução. 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. Uma ideia inicial é contar todas as operações realizadas pelo algoritmo, mas veremos que esta abordagem possui alguns problemas, além de ser muito trabalhosa.

5.1 Busca sequencial

Como primeiro exemplo, considere o problema de buscar um elemento em um vetor arbitrário. O pseudo-código 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 da primeira ocorrência encontrada. Caso contrário, o algoritmo retorna o valor -1.

```

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

```

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

Parece bastante intuitivo dizer que este algoritmo é correto, mas como **provar** isto? Inicialmente temos que expressar a noção de correção por meio de um teorema e em seguida, precisamos provar este teorema. Construiremos uma invariante para o laço **while** que nos permita concluir a correção ao final da execução do algoritmo. Uma invariante é uma propriedade que é verdadeira antes da execução do algoritmo (inicialização), permanece verdadeira durante a execução do algoritmo (manutenção) e tal que sua validade ao final da execução do algoritmo nos permite concluir sua correção (finalização). Para o algoritmo SequentialSearch, considere a seguinte invariante:

Antes da i -ésima iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x .

A prova de uma invariante é construída pelos 3 passos citados acima:

Demonstração. A prova é dividida em 3 passos:

- **Inicialização:** Precisamos mostrar que antes da primeira iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x . Este passo é trivial porque antes da primeira iteração, temos que $i = 0$, e portanto o subvetor $A[0..i-1]$ é vazio.

- **Manutenção:** Este é o passo que exige mais cuidado na prova. Observe que antes da primeira iteração o valor de i é 0. Considerando que as condições do laço sejam satisfeitas, i será incrementado, e portanto antes da segunda iteração o valor de i é 1, e assim sucessivamente. Logo, antes da k -ésima iteração $k > 1$, o valor de i é $k-1$ e podemos assumir por hipótese que o subvetor $A[0..k-2]$ não possui ocorrências de x . Para que a próxima iteração ocorra, precisamos que $k < n$

e $A[k-1] \neq x$. Nestas condições, temos que o subvetor $A[0..k-1]$ não possui ocorrências de x preservando assim, a invariante.

- **Finalização:** Ao final da execução do laço, a condição " $i < n$ and $A[i] = x$ ($0 \leq i < n$)" não é mais satisfeita, e portanto temos que $i \geq n$ ou $A[i] = x$ ($0 \leq i < n$). Se $i \geq n$ então o vetor A não possui ocorrências de x e o algoritmo retorna -1 de acordo com a linha 5. Se $A[i] = x$ ($0 \leq i < n$) então a posição i é retornada, uma vez que o elemento procurado está na posição i do vetor A . Isto finaliza a prova de correção do algoritmo SequentialSearch.

□

Agora vamos analisar a complexidade em tempo e espaço deste algoritmo. 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:

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 que constitui o melhor caso possível, e portanto será denotada como *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 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:

1. 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	não é executada
2	c_2	
3	0	
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.

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

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 $T_w(n) = c_1 + c_2 \cdot n + c_3 \cdot (n - 1) + c_5 + c_8$. 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.

5.2 O algoritmo de ordenação por inserção (*insertion sort*)

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 (*insertion sort*), cujo pseudocódigo é dado a seguir:

```

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

```

$$T(n) = \sum_{j=1}^{n-1} t_j$$

① Melhor caso: $(t_j = 1) : T_b(n) = \sum_{j=1}^{n-1} 1 = n - 1 = O(n)$

② Pior caso: $(t_j = j) : T_w(n) = \sum_{j=1}^{n-1} j = \frac{(n-1) \cdot n}{2} = O(n^2)$

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

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

A primeira pergunta que precisamos responder é: **este algoritmo é correto?** Isto é, ele satisfaz as especificações do problema que propõe resolver?

1. A correção do algoritmo de ordenação por inserção

Queremos ordenar os elementos de um vetor, assim esperamos que os elementos do vetor gerado após a execução do algoritmo coincidam com os elementos do vetor original, e que o vetor resultante esteja ordenado. Como vimos no exemplo anterior, em algoritmos iterativos utilizamos as invariantes de laço para estabelecer a correção do algoritmo. Dada a dinâmica do algoritmo InsertionSort, considere a seguinte invariante de laço:

Antes da j -ésima iteração do laço **for** (linhas 1-9), o subvetor $A[0..j-1]$ está ordenado e contém os mesmos elementos do vetor original $A[0..j-1]$.

Assim, se esta propriedade for válida ao final da execução do laço **for**, *i.e.* antes da $n+1$ -ésima iteração, teremos que o vetor gerado consiste dos elementos do vetor original $A[0..n-1]$ ordenado. Isto corresponde a dizer que InsertionSort é correto.

Como então provar esta invariante para InsertionSort? A prova é por indução no número de iterações do laço **for**:

- **Inicialização** (Base da indução):

Antes da primeira iteração do laço **for**, temos que $j = 1$ (condição necessária para iniciar o laço), e portanto a invariante é trivial porque o subvetor unitário $A[0]$ está ordenado por definição.

- **Manutenção** (Passo indutivo):

Considere a k -ésima iteração, isto é, $j = k$ ($1 < k < n$). Temos como hipótese que "Antes da k -ésima iteração do laço **for** o subvetor $A[0..k-1]$ é uma permutação que está ordenada do subvetor original $A[0..k-1]$." Assim, durante a k -ésima iteração, o laço **while** vai deslocar cada elemento maior do que $A[k]$, *i.e.* key , uma posição para a direita até encontrar a posição correta onde o elemento $A[k]$ deve ser inserido, de forma que neste momento o subvetor $A[0..k]$ está ordenado e possui os mesmos elementos do subvetor $A[0..k]$ original. A incrementarmos o valor de k para a próxima iteração, a invariante é reestabelecida. Informalmente estamos dizendo que o laço **while** encontra a posição correta para inserir $A[j]$ (que está armazenado na variável key). Provaremos este fato com a ajuda de uma invariante para o laço **while**:

Antes de cada iteração do laço **while**, o subvetor $A[i..j]$ possui elementos que são maiores ou iguais a key .

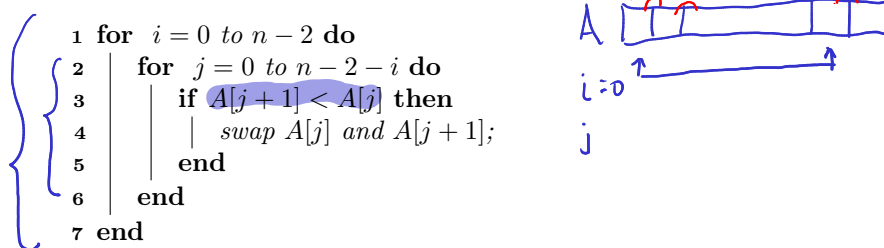
A prova é também por indução no número de iterações do laço **while**:

- (a) **Inicialização**: Antes da primeira iteração do **while**, estamos assumindo as condições para que o laço seja executado pela primeira vez, ou seja, temos que $i = j - 1 = k - 1$, $key = A[j]$ e $A[i] > key$, e portanto a invariante está satisfeita.
- (b) **Manutenção**: Por hipótese de indução temos que o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key . Durante uma iteração do laço, o elemento $A[i]$ é copiado na posição $i+1$ do vetor A , e portanto a invariante continua valendo.

(c) **Finalização:** Ao final da execução do laço, temos que i é, de fato, a posição correta para inserir o elemento $A[k]$ já que todos os elementos do subvetor $A[i+1..j]$ são maiores ou iguais a key . É importante observar que a inserção do elemento $A[k]$ na posição i não elimina nenhum elemento do vetor original porque o elemento que está na posição i foi copiado para a posição $i+1$, se o laço **while** foi executado pelo menos uma vez, ou ele é o próprio elemento armazenado em key , quando o laço não é executado.

• **Finalização:** Ao final da execução do laço **for**, temos $j = n$, e portanto a invariante corresponde a dizer que o vetor $A[0..n-1]$ obtido ao final da execução do algoritmo está ordenado, e é uma permutação do vetor original $A[0..n-1]$. Assim, concluímos a prova da correção do algoritmo InsertionSort.

Exercício 84. Prove que o algoritmo BubbleSort a seguir é correto.



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

Exercício 85. Prove que o algoritmo SelectionSort a seguir é correto.

```

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

```

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

2. A complexidade do algoritmo de ordenação por inserção

Agora 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:

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

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

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	$n - 1$	$c_8 \cdot (n - 1)$

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 $2 \leq j \leq n$:

$$T_b(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot (n - 1) + c_8 \cdot (n - 1) \\ = (c_1 + c_2 + c_3 + c_4 + c_8) \cdot n - (c_2 + c_3 + c_4 + c_8)$$

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 86. *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 encontrarmos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$. 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 apresenta uma definição alternativa para o conjunto $O(g(n))$ em termos de limites:

Lema 87. 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. $\forall \epsilon > 0, \exists n_0 : \left| \frac{f(n)}{g(n)} - c \right| < \epsilon, \forall n > n_0$ □

Teorema 88. (a) 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 $g(n) = O(f(n))$;

$-\epsilon < \frac{f(n)}{g(n)} - c < \epsilon, \forall n > n_0$
 $f(n) < (c + \epsilon) \cdot g(n), \forall n > n_0$

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

(c) 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

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

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 2 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*:

$$\Rightarrow \underbrace{O(f(n) + g(n))}_{\downarrow} = O(\max(f(n), g(n))) \quad (5.1)$$

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 exaustivamente 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.

Exercício 89. Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios 7 e 9.

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			$O(1)$
→ Selection sort			$O(1)$

Exercício 90. Mostre que $n = O(n^2)$.

Exercício 91. Mostre que $100n + 5 = O(n^2)$.

Exercício 92. Mostre que $\frac{n(n-1)}{2} = O(n^2)$.

↗ **Exercício 93.** Mostre que $n^3 \neq O(n^2)$.

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

Definição 94. *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 um 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 95. *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 96. *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 97. 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 98. (a) Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;

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

(c) 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 99. Prove que $\sum_{i=1}^n i^k = \Theta(n^{k+1})$ para qualquer inteiro $k \geq 0$ fixado.

Teorema 100. 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 101. (a) $f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;

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

(c) 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;

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

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

Definição 102. 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 103. Uma função $f(n) = o(g)$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Definição 104. 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 105. Uma função $f(n) = \omega(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, se este limite existir.

Lema 106. 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 107. $\lg n = o(n^\alpha), \forall \alpha > 0$. Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de n (incluindo potências fracionárias)

Teorema 108. $n^k = o(2^n), \forall k > 0$. 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 109. Mostre que $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 110. Mostre que $6n^3 \neq \Theta(n^2)$.