Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

1 Introdução e Motivação

O objetivo deste curso é o estudo de técnicas para analisar e projetar de algoritmos. As técnicas utilizadas para analisar algoritmos, nos permitirão estudar o tempo e o espaço demandado pelo algoritmo para processar uma determinada entrada (complexidade de tempo e de espaço). As técnicas de projeto de algoritmos estão relacionadas ao trabalho da construção de algoritmos a partir de um dado problema. Neste contexto, a técnica de divisão e conquista resolve um problema a partir da sua divisão em problemas menores seguida de uma combinação adequada das soluções dos subproblemas para assim construir uma solução do problema original. Outras técnicas que serão estudadas incluem a chamada *programação dinâmica* e os *algoritmos gulosos*. Uma vez construído, a primeira preocupação que precisamos ter em relação a um algoritmo é se ele funciona da forma esperada. Em outras palavras, antes de qualquer coisa precisamos saber se o algoritmo é correto, e só depois faremos a análise da sua complexidade. Intuitivamente, dizemos que um algoritmo A é correto quando sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se A é um algoritmo de ordenação de inteiros, então espera-se que para qualquer lista de inteiros l, A retorne uma permutação de l 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.

Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática. Ao analisarmos a eficiência (ou complexidade) dos algoritmos também faremos o uso de diversas ferramentas matemáticas (somatórios, conjuntos, funções, matrizes, etc). O apêndice VIII do livro [2, 3] pode ser usado para revisar estes temas.

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. À medida que estes equipamentos se 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. 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. Mais ainda, é a análise da complexidade quem vai nos mostrar como podemos aproveitar melhor a capacidade de processamento de um computador como veremos ao longo do curso.

2 A correção de algoritmos

Como primeiro exemplo, considere o problema de ordenar uma lista de inteiros. Existem diversos algoritmos que resolvem este problema, e uma abordagem simples consiste em inicialmente inserir elementos em uma lista ordenada. A função $insert\ x\ l$ definida a seguir, insere o elemento x na lista l:

$$insert\ x\ l := \begin{cases} x :: nil, & \text{se } l = nil \\ x :: l, & \text{se } x \leq h \text{ e } l = h :: tl \\ h :: (insert\ x\ tl), & \text{se } x > h \text{ e } l = h :: tl \end{cases}$$

*flaviomoura@unb.br

A inserção é feita de forma que se l está ordenada então a lista resultante também está ordenada. Podemos enunciar esta propriedade como um lema:

Lema 2.1. *Se l é uma lista ordenada e x é um inteiro, então $(insert\ x\ l)$ é uma lista ordenada.*

Exercício 2.2. *Prove o Lema 2.1.*

Na aula de *[2024-03-18 seg]*, iniciamos a prova do Lema 2.1 utilizando indução na estrutura da lista l. A base da indução corresponde ao caso em que a lista l é vazia. Neste caso, temos que provar que $insert\ x\ nil$ é uma lista ordenada, assumindo que a lista vazia está ordenada. Mas isto é trivial porque, pela definição de *insert*, $insert\ x\ nil$ é a lista unitária $x :: nil$ que está ordenada por definição.
O passo indutivo ocorre quando l tem a forma $h :: tl$. Neste caso, temos que provar que $insert\ x\ (h :: tl)$ é uma lista ordenada, assumindo que a lista $h :: tl$ está ordenada. A definição de *insert* nos dá dois subcasos:

1. $x \leq h$: Neste caso, $insert\ x(h :: tl)$ retorna $x :: h :: tl$ que está ordenada porque o elemento inserido x é menor ou igual ao primeiro elemento de uma lista ordenada.
2. $x > h$: Na aula de *[2024-03-20 qua]*, iniciamos com a escrita da hipótese de indução:

   > Se tl é uma lista ordenada e x é um inteiro, então $(insert\ x\ tl)$ é uma lista ordenada.

Temos como hipótese (do problema) que $h :: tl$ é uma lista ordenada, e queremos mostrar que a lista $insert\ x\ (h :: tl)$ é uma lista ordenada. Pela definição da função *insert*, temos que $insert\ x\ (h :: tl) = h :: (insert\ x\ tl)$. Agora observe que, por hipótese de indução, a lista $insert\ x\ tl$ está ordenada. Além disto, temos que $h < x$ e h é menor ou igual a todo elemento da lista tl já que $h :: tl$ está ordenada. Então h é menor ou igual a todo elemento de $(insert\ x\ tl)$, e portanto a lista $h :: (insert\ x\ tl)$ está ordenada. □

Agora vamos refazer esta prova mecanicamente utilizando o assistente de provas Coq. Primeiramente, vamos definir o predicado que caracteriza uma lista ordenada.

```
Inductive ordenada : list nat -> Prop :=
  | nil_ord : ordenada nil
  | one_ord : forall n:nat, ordenada (n::nil)
  | cons_ord : forall (n m:nat) (l:list nat), n <= m ->
               ordenada (m::l) -> ordenada (n::m::l).
```

A função *insert* pode ser definida em Coq como segue:

```
Fixpoint insert (x:nat) (l:list nat) : list nat :=
  match l with
  | nil => x::nil
  | h::tl => if x <=? h then x::l else h::(insert x tl)
  end.
```

Agora podemos enunciar e provar o lema 2.1 em Coq:

```
Lemma insert_ord : forall (x:nat) (l:list nat), ordenada l -> ordenada (insert x l).
```

Exercício 2.3. *Prove o lema* `insert_ord` *no Coq.*

A resolução do exercício anterior nos dá uma boa ideia do que consiste o processo de formalização (ou mecanização) de um algoritmo, ainda que bastante simples. A função principal do algoritmo de ordenação por inserção é dada a seguir:

$$insertion\_sort\ l := \begin{cases} l, & \text{se } l = nil \\ insert\ x\ (insertion\_sort\ tl), & \text{se } l = h :: tl \end{cases}$$

Exercício 2.4. *O algoritmo* `insertion_sort` *é correto? Como você responderia a esta pergunta com as devidas justificativas em papel e lápis? E no Coq?*

Tente resolver os exercícios apresentados, e traga as suas dúvidas para discutirmos na próxima aula. O arquivo Coq está disponível em http://flaviomoura.info/files/paa_2024_1_introducao.v

Exercício 2.5. *Considere a seguinte função:*

$$f\ n := \begin{cases} 0, & \text{se } n = 0 \\ 2.n - 1 + f\ (n-1), & \text{se } n > 0 \end{cases}$$

Mostre que a função f computa o quadrado do seu argumento, i.e. $f\ n = n^2$.

Exercício 2.6. *Considere a seguinte função:*

$$seq\_search\ x\ l := \begin{cases} FALSE, & \text{se } l = nil; \\ TRUE, & \text{se } l = h :: l' \text{ e } h = x; \\ seq\_search\ x\ l', & \text{se } l = h :: l' \text{ e } h \neq x; \end{cases}$$

Prove que se x for um elemento de l então seq_search x l retorna TRUE, e retorna FALSE, quando x não ocorre em l.

Exercício 2.7. *A busca binária é mais eficiente que a busca sequencial, mas este algoritmo assume que o vetor, onde a busca será realizada a busca, está ordenado. O pseudocódigo da busca binária em um vetor ordenado de inteiros com n elementos é dado a seguir:*

Algorithm 1: BinarySearch($A[1..n], low, high, key$)

```
1  if high < low then
2  |  return -1;
3  end
4  mid = ⌊(high + low)/2⌋;
5  if key > A[mid] then
6  |  return BinarySearch(A, mid + 1, high, key);
7  end
8  else
9  |  if key < A[mid] then
10 |  |  return BinarySearch(A, low, mid − 1, key);
11 |  end
12 |  else
13 |  |  return mid;
14 |  end
15 end
```

A correção deste algoritmo pode ser estabelecida em duas etapas. A primeira dela consiste em provar que se a chave key não ocorre no vetor $A[1..n]$, então BinarySearch($A[1..n], 1, n,$ key) retorna o valor -1, e a segunda consiste em provar que o algoritmo retorna a posição correta do elemento procurado.

Prove as seguintes propriedades:

1. *Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key não ocorre em $A[1..n]$, então BinarySearch($A[1..n], 1, n,$ key) retorna o valor -1.*

2. *Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key ocorre em $A[1..n]$ na posição $1 \leq j \leq n$, então BinarySearch($A[1..n], 1, n,$ key) retorna o valor j.*

2.1 A correção de algoritmos não-recursivos

Como primeiro exemplo, considere o problema de buscar um elemento em um vetor de números naturais. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n-1]$ contendo n números naturais e o natural 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 2: SequentialSearch($A[0..n-1], x$)

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

Parece bastante intuitivo dizer que este algoritmo é correto, mas como **provar** isto? Inicialmente temos que expressar a noção de correção como uma propriedade, e em seguida, precisamos provar esta propriedade. Em programas contendo laços utilizamos as *invariantes de laço*, que são propriedades satisfeitas durante toda a execução de um laço. A prova de uma invariante de laço consiste em três etapas, e é análoga a uma prova indutiva:

1. **Inicialização**: Nesta etapa mostramos que a propriedade é satisfeita antes da primeira execução do laço. Esta etapa é equivalente à base da indução em uma prova indutiva;
2. **Manutenção**: Esta é a etapa mais delicada da prova porque corresponde ao passo indutivo. Aqui assumimos por hipótese (de indução) que a invariante vale antes de uma iteração arbitrária do laço (depois da primeira) e mostramos que a invariante continua válida antes da próxima iteração. Ou seja, assumimos que a invariante vale antes da k-ésima iteração ($k > 0$) e mostramos que ao final desta iteração, isto é, antes da $(k+1)$-ésima iteração, a invariante continua satisfeita;
3. **Finalização**: Nesta etapa concluímos que a invariante é satisfeita durante toda a execução do laço, e esta informação é utilizada para estabelecer a correção do algoritmo.

Com estas informações, construiremos uma invariante para o laço **while** (linhas 2-4) que nos permita concluir a correção ao final da execução do algoritmo. A construção da invariante costuma ser a etapa mais difícil porque precisamos pensar em uma propriedade que seja verdadeira antes da execução do laço (inicialização), permaneça verdadeira durante toda a execução do laço (manutenção) e tal que sua validade ao final da execução do laço nos permita concluir sua correção (finalização).

No caso da busca sequencial queremos mostrar que ao final da execução, o algoritmo SequentialSearch retorna -1 se x não ocorre em A, e retorna $0 \leq i \leq n-1$ se a primeira ocorrência de x é na posição i do vetor A. Precisamos adaptar esta propriedade de forma que possa ser utilizada ao longo da execução do algoritmo, e não apenas ao final de sua execução. Sabemos que uma nova iteração do laço **while** só ocorrerá se o elemento x não foi ainda encontrado, pois caso contrário o algoritmo para. A partir destas observações, considere a seguinte invariante para o laço **while** do algoritmo SequentialSearch:

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 consideraremos novamente o algoritmo de ordenação por inserção, mas sua versão não-recursiva. Sabemos que o algoritmo *insertion sort* (ordenação por inserção)e tem como etapa principal inserir um elemento em um vetor ordenado. Assim, para ordenarmos $n > 0$ números naturais em ordem crescente vamos supor que estes números estejam armazenados no vetor $A[0..n-1]$. 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 $1 \leq i < n$. O algoritmo de ordenação por inserção (*insertion sort*) é dado pelo pseudocódigo a seguir([2]):

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

```
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
```

A prova da correção de algoritmos iterativos que possuem laços utilizam as *invariantes de laço*, *i.e.* uma propriedade que é preservada durante a execução do laço. 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 $A[i+1..j]$ possui elementos que são maiores ou iguais a key.

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

1. **Inicialização**: Antes da primeira iteração do **while** temos que $i + 1 = j = k$, e como $key = A[j]$ a invariante está satisfeita.
2. **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.
3. **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 2.8. *Prove que o algoritmo BubbleSort a seguir é correto.*

Algorithm 4: 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
```

Exercício 2.9. *Prove que o algoritmo BubbleSort2 [3] a seguir é correto.*

Algorithm 5: BubbleSort2($A[0..n - 1]$)

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

Exercício 2.10. *Prove que o algoritmo SelectionSort a seguir é correto.*

Algorithm 6: SelectionSort($A[0..n - 1]$)

```
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
```

3 A complexidade dos algoritmos

Agora vamos analisar a complexidade em tempo e espaço do algoritmo SequentialSearch. 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
$n.\log n$	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:

3.1 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	
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. $T_b(n) = O(1)$

3.2 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	$c_2.(n+1)$
3	$c_3.n$
5	c_5
6	0
8	c_8
Total	$c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$

$\rightarrow T_w(n) = O(n)$

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

3.3 A complexidade da ordenação por inserção

Agora faremos uma análise da complexidade do algoritmo de ordenação por inserção. 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 $j = 1, \ldots, n-1$. 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	$c_1.n$
2	c_2	$n-1$	$c_2.(n-1)$
3	c_3	$n-1$	$c_3.(n-1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4.\sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1}(t_j-1)$	$c_5.\sum_{j=1}^{n-1}(t_j-1)$
6	c_6	$\sum_{j=1}^{n-1}(t_j-1)$	$c_6.\sum_{j=1}^{n-1}(t_j-1)$
8	c_8	$n-1$	$c_8.(n-1)$

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

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

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 $2 \leq j \leq n$:

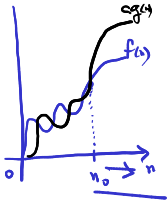
$$\begin{aligned} T_b(n) &= c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.(n-1) + c_8.(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8).n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

$\longrightarrow T_b(n) = O(n)$

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

$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\frac{(n-1).n}{2}) + (c_5 + c_6).(\frac{(n-2).(n-1)}{2})$ $\rightarrow T_w(n) = O(n^2)$.
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 $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8$ 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 $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8 \leq c_1.n + c_2.n + c_3.n - c_3 + c_6.n + c_8.n \leq (c_1 + c_2 + c_3 + c_6 + c_8).n \leq c.n$ 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 $g(n)$:

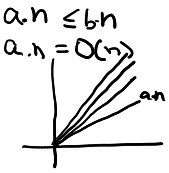


Definição 3.1. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.\}$*

$f(n) \leq c.g(n)$, $\forall n \geq n_0$.

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.

$a.n \leq b.n$
$a.n = O(n)$



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 $O(1) + O(2) + \ldots + O(n)$ (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 3.2. *Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, incluindo o caso em que $c = 0$, então $f(n) = O(g(n))$.*

Demonstração. Exercício. □

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lfloor \lg n \rfloor}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lg n} \leq 2^{\lfloor \lg n \rfloor + 1} = 2.2^{\lfloor \lg n \rfloor} = 2.g(n), \forall n$. No entanto, o limite $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 3.3. *1. Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, onde c é uma constante real positiva, então* $f(n) = O(g(n))$ *e* $g(n) = O(f(n))$*;*

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

3. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *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
$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\frac{(n-1).n}{2}) + (c_5 + c_6).(\frac{(n-2).(n-1)}{2})$ 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 da página 8, 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*:

$$O(f(n) + g(n)) = O(max(f(n), g(n))) \tag{1}$$

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.

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 3.4. *Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios da página 6.*

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 3.5. *Mostre que* $n = O(n^2)$.

Exercício 3.6. *Mostre que* $100n + 5 = O(n^2)$.

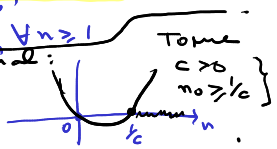
Exercício 3.7. *Mostre que* $\frac{n(n-1)}{2} = O(n^2)$.

Exercício 3.8. *Mostre que* $n^3 \neq O(n^2)$.

Temos que mostrar que existem constantes $c>0$ e $n_0>0$ tais que $n \leq c.n^2$, $\forall n \geq n_0$. Tome $c=1$ e $n_0=1$ $n \leq n^2$ $\forall n \geq 1$

De uma forma mais geral: $c.n^2 - n \geq 0$, $\forall n \geq n_0$

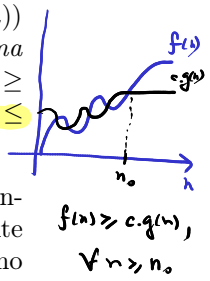
raízes: 0 e $1/c$ ($c>0$).



Exercício 3.9. *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 3.10. *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, $\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 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 3.11. *Uma função $f(n) = \Omega(g)$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, 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.

(3.7) $\frac{n(n-1)}{2} = O(n^2)$.

Precisamos mostrar que existem constantes positivas $c>0$ (real) e $n_0>0$ (inteira) tais que

$\frac{n(n-1)}{2} \leq c \cdot n^2$, $\forall n \geq n_0$.

$c=1$ e $n_0=1$ ✓

$n^2 + n \geq 0$, $\forall n \geq 1$

$n^2 - n \leq 2c \cdot n^2$, $\forall n \geq n_0$

$-n \leq (2c-1) \cdot n^2$, $\forall n \geq n_0 > 0$

$(2c-1) n^2 + n \geq 0$, $\forall n \geq n_0$.

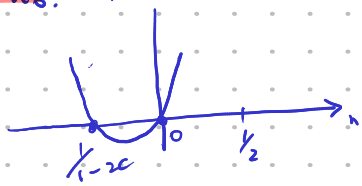
$\equiv n \cdot ((2c-1) \cdot n + 1) \geq 0$

$n = \frac{-1}{2c-1} = \frac{1}{1-2c}$.

$2c-1 > 0 \Rightarrow c > 1/2$:

raízes: 0 e $1/1-2c$

Tome $n_0 > 0$ e $c > 1/2$

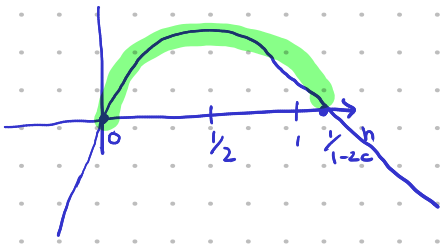


$2c-1 < 0 \Rightarrow c < \frac{1}{2}$:

raízes: 0 e $1/1-2c$

Tome $0 < c < 1/2$ e

$0 \leq n_0 \leq 1/1-2c$.



(3.8) $n^3 \neq O(n^2)$.

Def.

Suponha que $n^3 = O(n^2)$. Neste caso, existem constantes positivas c e n_0 tais que

$n^3 \leq c \cdot n^2$, $\forall n \geq n_0 > 0$ $\Longleftrightarrow$

$n \leq c$, $\forall n \geq n_0$ ↯

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 3.4. *Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios da página 6.*

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 3.5. *Mostre que* $n = O(n^2)$.

Exercício 3.6. *Mostre que* $100n + 5 = O(n^2)$.

Exercício 3.7. *Mostre que* $\frac{n(n-1)}{2} = O(n^2)$.

Exercício 3.8. *Mostre que* $n^3 \neq O(n^2)$.

Exercício 3.9. *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 3.10. *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,* $\Omega(g(n)) = \{f(n) :$ *existem constantes positivas* c *e* n_0 *tais que* $0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$

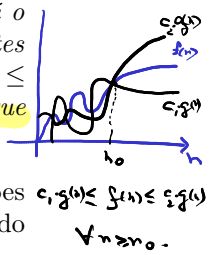
Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n, o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor 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 3.11. *Uma função* $f(n) = \Omega(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, *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 3.12. *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, $\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 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 3.13. *Uma função $f(n) = \Theta(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, para alguma constante $0 < c < \infty$.*

Demonstração. Exercício. □

Teorema 3.14.

1. *Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;*
2. *Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $f(n) \neq \Theta(g(n))$;*
3. *Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ então $f(n) = \Omega(g(n))$, mas $f(n) \neq O(g(n))$.*

Demonstração. Exercício. □

Exercício 3.15. *Prove que $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ para qualquer inteiro $k \geq 0$ fixado.*

Teorema 3.16. *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 3.17.

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

Lema 3.19. *Uma função $f(n) = o(g)$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.*

Definição 3.20. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $\omega(g(n)) = \{f(n) :$ para qualquer constante positiva c, existe uma constante positiva n_0 tal que $0 \leq c \cdot g(n) < f(n), \forall n \geq n_0.\}$*

Lema 3.21. *Uma função $f(n) = \omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, se este limite existir.*

Lema 3.22. *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 3.23. $\lg n = o(n^\alpha), \forall \alpha > 0$. *Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de n (incluindo potências fracionárias)*

Teorema 3.24. $n^k = o(2^n), \forall k > 0$. *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 3.25. *Mostre que* $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 3.26. *Mostre que* $6n^3 \neq \Theta(n^2)$.

Exercício 3.27. *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* $f(n) + g(n) = O(h(n))$.

4 Leitura complementar:

- [2] (Capítulos 1 e 2)
- [3] (Capítulos 1 e 2)
- [4] (Capítulos 1 e 3)
- [1] (Capítulos 1 e 2)

Referências

[1] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[4] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.