

# Projeto e Análise de Algoritmos

Flávio L. C. de Moura

20 de abril de 2023

Finalizando a discussão da aula anterior, vamos apresentar uma prova da correção da busca sequencial dada pelo pseudocódigo a seguir:

```
i ← 0;
while i < n and A[i] ≠ x do
  | i ← i + 1;
end
if i < n then
  | return i;
else
  | return -1
end
```

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

A correção de SequentialSearch será estabelecida por meio do seguinte teorema:

**Teorema 1.** *Se  $x$  ocorre em  $A[0..n - 1]$  então o algoritmo SequentialSearch( $A[0..n - 1], x$ ) retorna o índice  $i$  ( $0 \leq i \leq n - 1$ ) correspondente a um índice de  $A$  que contém o elemento  $x$ , e retorna -1 quando  $x$  não ocorre em  $A[0..n - 1]$ .*

*Prova.* A prova será feita por meio da seguinte invariante (proposta durante a aula):

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

**Inicialização.** Antes de 0-ésima iteração (primeira iteração), temos que  $i = 0$  e portanto a invariante é verdadeira uma vez que o vetor  $A[0..i - 1]$  é vazio neste caso.

**Manutenção.** Assuma que antes da  $k$ -ésima ( $0 < k \leq n - 1$ ) iteração, o subvetor  $A[0..k - 1]$  não possui ocorrências do elemento  $x$ . Durante a  $k$ -ésima iteração o elemento  $x$  será comparado com  $A[k]$ , e temos dois casos:

1.  $A[k] \neq x$ : Neste caso, concluímos que o subvetor  $A[0..k]$  não possui ocorrências do elemento  $x$ , e a invariante permanece verdadeira antes da  $k + 1$ -ésima iteração;
2.  $A[k] = x$ : Neste caso, o laço termina, e  $i$  não é mais incrementado, e portanto a invariante permanece verdadeira durante toda a execução do laço. Adicionalmente, neste caso o algoritmo retorna o índice  $k$ .

**Terminação.** Ao final da execução do laço temos que  $i \geq n$  ou  $A[i] = x$ , *i.e.* a negação da condição do laço **while**. Novamente, temos 2 casos a considerar:

1.  $i \not\leq n$ : Neste caso, temos que  $i = n$  e portanto a invariante nos garante que o (sub)vetor  $A[0..i - 1] = A[0..n - 1]$  não possui ocorrências do elemento  $x$  e o algoritmo retorna -1 (linha 8);
2.  $A[i] = x$ : Neste caso, o algoritmo retorna o índice  $i$  (linha 6) que corresponde a uma posição de  $A$  que contém o elemento  $x$ . Mais ainda, como a invariante nos garante que o subvetor  $A[0..i - 1]$  não possui ocorrências do elemento  $x$ , temos que  $i$  corresponde ao menor índice de  $A$  que contém  $x$ .

□

**Exercício 2.** Considere o seguinte pseudocódigo para a busca sequencial, onde  $A[0..n-1]$  é um vetor qualquer:

```

i ← -1;
for j = 0 to n - 1 do
  if A[j] = x then
    i ← j;
  end
end
return i

```

**Algoritmo 2:** SeqSearch( $A[0..n-1], x$ )

Responda:

1. Este algoritmo é correto? Em caso, afirmativo faça a prova, e em caso negativo, justifique sua resposta.
2. Faça a análise assintótica deste algoritmo.

Em vetores ordenados podemos utilizar a busca binária:

```

if high < low then
  return -1;
end
mid = ⌊(high + low)/2⌋;
if key > A[mid] then
  return BinarySearch(A, mid + 1, high, key);
end
else
  if key < A[mid] then
    return BinarySearch(A, low, mid - 1, key);
  end
  else
    return mid;
  end
end

```

**Algoritmo 3:** BinarySearch( $A, low, high, key$ )

1. Prove que BinarySearch é correto.
2. Faça a análise assintótica de BinarySearch.

Para a próxima aula, estude o material a seguir e o Capítulo 4 de [1].

O algoritmo de ordenação *mergesort* é um exemplo de algoritmo recursivo, que se caracteriza por dividir o problema original em subproblemas que, por sua vez, são resolvidos recursivamente. As soluções dos subproblemas são então combinadas para gerar uma solução para o problema original. Este paradigma de projeto de algoritmo é conhecido com *divisão e conquista*. Este algoritmo foi inventado por J. von Neumann em 1945.

```

if p < r then
  q = ⌊(p+r)/2⌋;
  mergesort(A, p, q);
  mergesort(A, q + 1, r);
  merge(A, p, q, r);
end

```

**Algoritmo 4:** mergesort( $A, p, r$ )

A etapa de combinar dois vetores ordenados (algoritmo *merge*) é a etapa principal do algoritmo *mergesort*. O procedimento  $merge(A, p, q, r)$  descrito a seguir recebe como argumentos o vetor  $A$ , e os índices  $p, q$  e  $r$  tais que  $p \leq q < r$ . O procedimento assume que os subvetores  $A[p..q]$  e  $A[q+1..r]$  estão ordenados.

```

 $n_1 = q - p + 1$  ; // Qtd. de elementos em  $A[p..q]$ 
 $n_2 = r - q$  ; // Qtd. de elementos em  $A[q+1..r]$ 
let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays;
for  $i = 1$  to  $n_1$  do
  |  $L[i] = A[p + i - 1]$ ;
end
for  $j = 1$  to  $n_2$  do
  |  $R[j] = A[q + j]$ ;
end
 $L[n_1 + 1] = \infty$ ;
 $R[n_2 + 1] = \infty$ ;
 $i = 1$ ;
 $j = 1$ ;
for  $k = p$  to  $r$  do
  | if  $L[i] \leq R[j]$  then
  | |  $A[k] = L[i]$ ;
  | |  $i = i + 1$ ;
  | end
  | else
  | |  $A[k] = R[j]$ ;
  | |  $j = j + 1$ ;
  | end
end

```

**Algoritmo 5:**  $merge(A, p, q, r)$

**Exercício 3.** Prove que o algoritmo *merge* é correto.

**Exercício 4.** Prove que o algoritmo *mergesort* é correto.

**Exercício 5.** Faça a análise assintótica do algoritmo *merge*.

**Exercício 6.** Faça a análise assintótica do algoritmo *mergesort*.

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [3]:

**Definição 7.** Seja  $f(n)$  uma função não-negativa definida no conjunto dos números naturais. Dizemos que  $f(n)$  é eventualmente não-decrescente se existir um número inteiro  $n_0$  tal que  $f(n)$  é não-decrescente no intervalo  $[n_0, \infty)$ , ou seja,

$$f(n_1) \leq f(n_2), \forall n_2 > n_1 \geq n_0.$$

**Definição 8.** Seja  $f(n)$  uma função não-negativa definida no conjunto dos números naturais. Dizemos que  $f(n)$  é suave se for eventualmente não-decrescente e

$$f(2.n) = \Theta(f(n))$$

**Teorema 9.** Sejam  $f(n)$  uma função suave, e  $c$  e  $n_0$  constantes positivas. Se  $f(2n) \leq c.f(n), \forall n \geq n_0$  então  $f(2^k n) \leq c^k . f(n), \forall n \geq n_0$  e  $k \geq 1$ .

**Teorema 10.** Seja  $f(n)$  uma função suave. Então para qualquer  $b \geq 2$  fixado,

$$f(b.n) = \Theta(f(n))$$

O teorema a seguir é conhecido como *regra da suavização*

**Teorema 11.** *Seja  $T(n)$  uma função eventualmente não-decrescente, e  $f(n)$  uma função suave. Se  $T(n) = \Theta(f(n))$  para valores de  $n$  que são potências de  $b$  ( $b \geq 2$ ), então*

$$T(n) = \Theta(f(n)), \forall n.$$

A regra da suavização nos permite expandir a informação sobre a ordem de crescimento estabelecida para  $T(n)$  de um subconjunto de valores (potências de  $b$ ) para o domínio inteiro. O teorema a seguir é um resultado muito útil nesta direção conhecido como *teorema mestre*:

**Teorema 12.** *Seja  $T(n)$  uma função eventualmente não-decrescente que satisfaz a recorrência*

$$T(n) = a.T(n/b) + f(n), \quad \text{para } n = b^k, k = 1, 2, 3, \dots$$

$$T(1) = c$$

onde  $a \geq 1, b \geq 2$  e  $c \geq 0$ . Se  $f(n) = \Theta(n^d)$ , onde  $d \geq 0$ , então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^d \\ \Theta(n^d \cdot \lg n), & \text{se } a = b^d \\ \Theta(n^d), & \text{se } a < b^d \end{cases}$$

*Prova.* Considere que  $f(n) = n^d$ . Aplicando o método da substituição para a recorrência do teorema, obtemos:

$$T(b^k) = a^k \cdot [T(1) + \sum_{j=1}^k f(b^j)/a^j]$$

Como  $a^k = a^{\log_b n} = n^{\log_b a}$ , podemos reescrever a equação acima como:

$$T(n) = n^{\log_b a} \cdot [T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j]$$

e para  $f(n) = n^d$ , temos:

$$T(n) = n^{\log_b a} \cdot [T(1) + \sum_{j=1}^{\log_b n} (b^j)^d/a^j] = n^{\log_b a} \cdot [T(1) + \sum_{j=1}^{\log_b n} (b^d/a)^j]$$

A soma acima forma uma série geométrica, e portanto:

$$\sum_{j=1}^{\log_b n} (b^d/a)^j = (b^d/a) \frac{(b^d/a)^{\log_b n} - 1}{(b^d/a) - 1}, \text{ se } b^d \neq a.$$

Quando  $b^d \neq a$ , temos que  $\sum_{j=1}^{\log_b n} (b^d/a)^j = \log_b n$ . Agora basta analisarmos cada um dos casos:  $a < b^d$ ,  $a > b^d$  e  $a = b^d$ . □

Apresentaremos agora uma versão um pouco mais geral do teorema mestre[2]. Consideraremos como anteriormente uma recorrência da forma:

$$T(n) = a.T(n/b) + f(n)$$

on  $a \geq 1$  e  $b > 1$  são constantes, e  $f(n)$  é uma função assintoticamente positiva.

**Teorema 13.** *Sejam  $a \geq 1$  e  $b \geq 2$  constantes,  $f(n)$  uma função assintoticamente positiva, e  $T(n)$  definida nos inteiros não-negativos pela recorrência  $T(n) = a.T(n/b) + f(n)$ , onde  $n/b$  deve ser interpretado como  $\lfloor n/b \rfloor$  ou  $\lceil n/b \rceil$ . Então  $T(n)$  tem as seguintes cotas assintóticas:*

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $T(n) = \Theta(n^{\log_b a})$ ;
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$ ;
3. Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para alguma constante  $\epsilon > 0$ , e se  $a \cdot f(n/b) \leq c \cdot f(n)$  para alguma constante  $c < 1$ , então para todo  $n$  suficientemente grande, temos que  $T(n) = \Theta(f(n))$ .

A prova será dividida em três lemas, onde inicialmente consideraremos que  $n$  é potência de  $b$ .

**Lema 14.** Sejam  $a \geq 1$  e  $b > 1$  constantes,  $f(n)$  uma função não-negativa definida para potências de  $b$ . Defina  $T(n)$  para potências de  $b$  pela recorrência:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ a \cdot T(n/b) + f(n), & \text{se } n = b^i \end{cases}$$

onde  $i$  é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j \cdot f(n/b^j).$$

*Prova.* Analise a árvore de recorrência da equação dada. □

Em termos da árvore de recorrência, os três casos do teorema mestre correspondem aos casos onde o custo total da árvore é:

1. dominado pelo custo das folhas;
2. uniformemente distribuído ao longo da árvore;
3. dominado pelo custo da raiz.

**Lema 15.** Sejam  $a \geq 1$  e  $b > 1$  constantes,  $f(n)$  uma função não-negativa definida para potências de  $b$ . A função  $g(n)$  definida para potências de  $b$  por:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j \cdot f(n/b^j).$$

tem as seguintes cotas assintóticas para potências de  $b$ :

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $g(n) = O(n^{\log_b a})$ ;
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $g(n) = \Theta(n^{\log_b a} \cdot \lg n)$ ;
3. Se  $a \cdot f(n/b) \leq c \cdot f(n)$  para alguma constante  $c < 1$  e para todo  $n$  suficientemente grande, então  $g(n) = \Theta(f(n))$ .

*Prova.* Exercício. □

**Exercício 16.** Resolva as seguintes relações de recorrência:

1.  $T(1) = 1, T(n) = 3T(n/2) + n^2, n \geq 2$
2.  $T(1) = 1, T(n) = 2T(n/2) + n, n \geq 2$
3.  $T(1) \in \Theta(1), T(n) = 3T(n/3 + 5) + n/2$
4.  $T(1) = 1, T(n) = 2T(n-1) + 1, n \geq 2$
5.  $T(1) \in \Theta(1), T(n) = 9T(n/3) + n$

6.  $T(1) \in \Theta(1), T(n) = T(2n/3) + 1$
7.  $T(1) \in \Theta(1), T(n) = 2T(n/4) + 1$
8.  $T(1) \in \Theta(1), T(n) = 2T(n/4) + \sqrt{n}$
9.  $T(1) \in \Theta(1), T(n) = 2T(n/4) + \sqrt{n} \lg^2 n$
10.  $T(1) \in \Theta(1), T(n) = 2T(n/4) + n$
11.  $T(1) \in \Theta(1), T(n) = 2T(n/4) + n^2$
12.  $T(1) \in \Theta(1), T(n) = 3T(n/2) + n \ln(n)$
13.  $T(1) \in \Theta(1), T(n) = 3T(n/4) + n \ln(n)$
14.  $T(1) \in \Theta(1), T(n) = 2T(n/2) + n \ln(n)$
15.  $T(1) \in \Theta(1), T(n) = 2T(n/2) + n/\ln(n)$
16.  $T(1) \in \Theta(1), T(n) = T(n-1) + 1/n$
17.  $T(1) \in \Theta(1), T(n) = T(n-1) + \ln(n)$
18.  $T(1) \in \Theta(1), T(n) = \sqrt{n}T(\sqrt{n}) + n$
19.  $T(n) = 8T(n/2) + \Theta(n^2)$
20.  $T(n) = 8T(n/2) + \Theta(1)$
21.  $T(n) = 7T(n/2) + \Theta(n^2)$

## Referências

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, fourth 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.