Projeto e Análise de Algoritmos (2025-2)

Flávio L. C. de Moura*

30 de setembro de 2025

Exercícios para a Prova 1

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

1. 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, utilizando as definições de notação assintótica, que $f(n) + g(n) = O(h(n))$.

*flaviomoura@unb.br

Solução

Sabemos que $f(n) = O(h(n))$, i.e. existem constantes positivas c_1 e n_1 tais que $f(n) \leq c_1.h(n)$, para todo $n \geq n_1$. Analogamente, de $g(n) = O(h(n))$, i.e. existem constantes positivas c_2 e n_2 tais que $g(n) \leq c_2.h(n)$, para todo $n \geq n_2$. Queremos mostrar que $f(n) + g(n) = O(h(n))$, i.e. precisamos encontrar constantes positivas c e n_0 tais que $f(n) + g(n) \leq c.h(n)$, para todo $n \geq n_0$. Somando as desigualdades acima, temos $f(n) + g(n) \leq (c_1 + c_2).h(n)$, para todo $n \geq \max\{n_1, n_2\}$. Logo, basta tomar $c = c_1 + c_2$ e $n_0 = \max\{n_1, n_2\}$. □

2. Sejam $f(n)$ e $g(n)$ funções não-negativas tais que $g(n) = O(f(n))$. Prove, utilizando as definições de notação assintótica, que $f(n) + g(n) = \Theta(f(n))$.

Solução

Por hipótese, temos que

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

e queremos mostrar que

$$f(n) + g(n) = \Theta(f(n)) \tag{2}$$

ou seja, precisamos encontrar constantes positivas c_1, c_2 e n_0 tais que

$$c_1, f(n) \leq f(n) + g(n) \leq c_2.f(n), \forall n \geq n_0$$

Da hipótese, (1) sabemos que existem constantes positivas c' e n' tais que

$$g(n) \leq c'.f(n), \forall n \geq n' \tag{3}$$

e portanto,

$$f(n) + g(n) \leq (1 + c').f(n), \forall n \geq n'.$$

Ou seja, $f(n) + g(n) = O(f(n))$. Adicionalmente, $f(n) + g(n) \geq f(n), \forall n$, e assim, podemos tomar $c_1 = 1$, $c_2 = 1 + c'$ e $n_0 = n'$. □

3. O pseudocódigo 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) Faça a análise da complexidade do melhor caso para este algoritmo.

Solução

No melhor caso, o elemento procurado está na posição central, e portanto não teremos chamadas recursivas na execução do algoritmo. Logo $T_b(n) = \Theta(1)$. □

4. Faça a análise da complexidade do pior caso para este algoritmo.

Solução

No pior caso, o elemento procurado não se encontra no vetor. Neste caso, a complexidade é dada pela recursão $T_w(n) = T_w(n/2) + \Theta(1)$ que tem solução $T_w(n) = O(\lg(n))$ pelo TM. □

5. A correção deste algoritmo pode ser estabelecida em duas etapas.

(a) 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. Prove a seguinte afirmação:

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.

Solução

Indução no tamanho do vetor A, ou seja, em $n = high - low + 1$.

Base ($n = 0$): Neste caso, $0 = high - low + 1$, e portanto $low > how$ e o algoritmo retorna -1 (linha 2).

Passo ($n > 0$): Temos 2 casos possíveis:

i. $key > A[mid]$. Neste caso, temos por hipótese de indução que BinarySearch$(A[1..n], mid + 1, n, key)$ retorna o valor -1, e portanto BinarySearch$(A[1..n], 1, n, key)$ retorna o valor -1.

ii. Quando $key < A[mid]$, a justificativa é análoga ao caso anterior. □

Como discutido na aula passada, indução é desnecessária neste caso. De fato, observe que a execução de BinarySearch sempre termina porque cada chamada recursiva se dá sobre um subvetor de tamanho estritamente menor do que o vetor inicial, e não é possível decrementar o tamanho de um vetor indefinidamente. Adicionalmente, qualquer execução do algoritmo termina necessariamente na linha 2 (retornando -1) ou linha 13 (retornando mid). A linha 13 só é executada quando as condições $key > A[mid]$ (linha 5) e $key < A[mid]$ (linha 9) falham, ou seja, pela lei da tricotomia, a linha 13 só é executada quando $key = A[i]$ para algum $1 \leq i \leq n$. Mas estamos assumindo que key não ocorre no vetor A, ou seja, estamos assumindo que $key \neq A[i]$, para todo $1 \leq i \leq n$. Logo, o algoritmo termina com a execução da linha 2 retornando -1, como queríamos provar. □

(b) A segunda etapa consiste em provar que se a chave key ocorre no vetor $A[1..n]$, então BinarySearch$(A[1..n], 1, n, key)$ retorna uma posição válida do vetor, digamos k, tal que $A[k] = key$. Prove a seguinte afirmação:

Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key ocorre no vetor $A[1..n]$, então BinarySearch$(A[1..n], 1, n, key)$ retorna o valor $1 \leq k \leq n$, tal que $A[k] = key$.

Solução

Indução no tamanho do vetor A, ou seja, em $n = high - low + 1$.

Base ($n = 1$): A base da indução é feita com um vetor unitário porque estamos assumindo que a chave key ocorre no vetor $A[1..n]$. Como $n = 1$, temos que $low = high$ e $mid = 1$. Como a chave key ocorre no vetor $A[1..n]$ então as condições das linhas 5 e 9 não são satisfeitas e a linha 13 retorna a posição 1 como esperado.

Passo ($n > 1$): Neste caso, se $key > A[mid]$ então key ocorre no subvetor $A[mid + 1..n]$ e por hipótese de indução BinarySearch$(A[1..n], mid + 1, n, key)$ retorna o valor $mid + 1 \leq k \leq n$, tal que $A[k] = key$ como desejado. Se $key < A[mid]$ então key ocorre no subvetor $A[1..mid - 1]$ e por hipótese de indução BinarySearch$(A[1..n], 1, mid - 1, key)$ retorna o valor $1 \leq k \leq mid - 1$, tal que $A[k] = key$ como desejado. Caso contrário, $key = A[mid]$ e a posição mid é retornada pela linha 13 como desejado. □

6. Considere o problema dos elementos únicos, que checa se todos os n elementos de um vetor de tamanho n são distintos:

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

```
1 for i = 0 to n − 2 do
2 |   for j = i + 1 to n − 1 do
3 |   |   if A[i] = A[j] then
4 |   |   |   return false
5 |   |   end
6 |   end
7 end
8 return true
```

Faça a análise assintótica deste algoritmo, e justifique sua resposta.

Solução

Seja $T(n)$ o número de comparações executadas por este algoritmo (linha 3). O número de comparações é interrompido assim que dois elementos iguais são encontrados (linha 4), e portanto, no melhor caso apenas uma comparação é feita: por exemplo, quando os dois primeiros elementos do vetor A são iguais. Neste caso, $T(n) = \Theta(1)$. No pior caso, temos $T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - 1 - i) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2}$. Logo $T(n) = \Theta(n^2)$, ou seja, o algoritmo é quadrático no tamanho da entrada. □

7. Considere um algoritmo recursivo hipotético que resolve uma classe de problemas da seguinte forma:

   (a) A instância do problema de tamanho n recebida como entrada é dividida em 4 subproblemas de tamanho $n/2$, que são por sua vez resolvidas recursivamente;

   (b) As soluções dos 4 subproblemas são combinadas em tempo $n^2 . \lg n$, para que seja possível formar uma solução do problema original.

   Responda os itens a seguir:

   (a) Escreva a recorrência que modela a complexidade deste algoritmo.

   Solução

   $$T(n) = 4.T(n/2) + n^2 . \lg n$$

   (b) Determine a complexidade assintótica deste algoritmo.

Solução

O teorema mestre não se aplica a esta recorrência. De fato, o caso 3 seria o candidato para resolver esta recorrência, mas para isto precisamos que $n^2.\lg n = \Omega(n^{\lg 4+\epsilon}) = \Omega(n^{2+\epsilon})$ para alguma constante $\epsilon > 0$, e isto não ocorre: de fato, para que $n^2.\lg n = \Omega(n^{2+\epsilon})$ precisam existir constantes positivas c_0 e n_0 tais que $n^2.\lg n \geq c_0.n^{2+\epsilon}, \forall n \geq n_0$ o que equivale a $\lg n \geq c_0.n^{\epsilon}, \forall n \geq n_0$, o que é um absurdo já que $\lg n = o(n^{\epsilon})$ para todo $\epsilon > 0$. Alternativamente, podemos simplesmente dizer que $n^2.\lg n$ não é *polinomialmente maior* do que n^2.

Ainda que desnecessário mostrar já que o caso 3 não aplica, mas considerando que a situação apresentada no parágrafo anterior não tenha sido observada, note que a condição de regularidade do caso 3 também não é satisfeita; de fato, para que $4.(n/2)^2.\lg(n/2) \leq c.n^2.\lg n$ para alguma constante $c < 1$ e n suficientemente grande, teríamos que

$$\begin{array}{ll} n^2.((\lg n) - 1) \leq c.n^2.\lg n & \Rightarrow \\ (\lg n) - 1 \leq c.\lg n & \Rightarrow \\ \lg n \leq \frac{1}{1-c} & \text{(para } n \text{ suficientemente grande)} \end{array}$$

o que é um absurdo, já que $\lg n$ é uma função crescente, e portanto não pode ser limitada por uma constante. □

(c) Qual é a a complexidade assintótica deste algoritmo hipotético? Justifique sua resposta.

Solução

Vamos resolver esta recorrência pelo método da substituição. Para isto, vamos assumir que $n = 2^k$ para algum k positivo, e que $T(1) = 0$. Assim,

$$\begin{aligned}
T(n) &= 4.T(n/2) + n^2.\lg n \\
&= 4^2.T(n/2^2) + n^2.(\lg(n/2) + \lg n) \\
&= 4^3.T(n/2^3) + n^2.(\lg(n/2^2) + \lg(n/2) + \lg n) \\
&= \ldots \\
&= 4^k.T(1) + n^2.(\sum_{i=0}^{k-1} \lg(n/2^i)) \\
&= n^2.(\sum_{i=0}^{k-1}(\lg(n) - i)) \\
&= n^2.(\sum_{i=0}^{k-1} \lg(n) - \sum_{i=0}^{k-1} i) \\
&= n^2.(k.\lg(n) - \tfrac{k.(k-1)}{2}) \\
&= n^2.(\lg^2(n) - \tfrac{\lg^2(n) - \lg n}{2}) \\
&= n^2.\tfrac{\lg^2(n) + \lg n}{2}.
\end{aligned}$$

Podemos utilizar indução (forte) para verificarmos que a expressão acima é, de fato, solução da recorrência original:

$$\begin{aligned}
T(n) &= 4.T(n/2) + n^2.\lg n \\
&\overset{h.i.}{=} 4.(\tfrac{n}{2})^2.\tfrac{\lg^2(\frac{n}{2}) + \lg(\frac{n}{2})}{2} + n^2.\lg n \\
&= n^2.\tfrac{((\lg n) - 1)^2 + (\lg n) - 1}{2} + n^2.\lg n \\
&= \tfrac{n^2.\lg^2 n - 2.n^2.\lg n + n^2 + n^2.\lg n - n^2 + 2.n^2.\lg n}{2} \\
&= \tfrac{n^2.(\lg^2 n + \lg n)}{2}.
\end{aligned}$$

Logo $T(n) = \Theta(n^2.\lg^2 n)$.

Alternativamente, podemos calcular cada uma das cotas separadamente utilizando indução (forte). Inicialmente, mostraremos que $T(n) \leq n^2.\lg^2 n$, para n suficientemente grande. Temos,

Solução

$$\begin{aligned} T(n) &= 4.T(n/2) + n^2.\lg n \\ &\overset{h.i.}{\leq} 4.((\tfrac{n}{2})^2.\lg^2(\tfrac{n}{2})) + n^2.\lg n \\ &= n^2.\lg^2(\tfrac{n}{2})) + n^2.\lg n \\ &= n^2.(\lg^2 n - 2.\lg n + 1) + n^2.\lg n \\ &= n^2.\lg^2 n - n^2.((\lg n) - 1) \\ &\leq n^2.\lg^2 n, \text{ para } n \geq 2. \end{aligned}$$

Portanto, $T(n) = O(n^2.\lg^2 n)$.
Para a cota inferior, mostraremos que $T(n) \geq \frac{n^2}{2}.\lg^2 n$, para n suficientemente grande. Temos,

$$\begin{aligned} T(n) &= 4.T(n/2) + n^2.\lg n \\ &\overset{h.i.}{\geq} 4.(\tfrac{(\frac{n}{2})^2}{2}.\lg^2(\tfrac{n}{2})) + n^2.\lg n \\ &= \tfrac{n^2}{2}.\lg^2(\tfrac{n}{2})) + n^2.\lg n \\ &= \tfrac{n^2}{2}.(\lg^2 n - 2.\lg n + 1) + n^2.\lg n \\ &= \tfrac{n^2}{2}.\lg^2 n + \tfrac{n^2}{2} \\ &\geq \tfrac{n^2}{2}.\lg^2 n \end{aligned}$$

Logo, $T(n) = \Omega(n^2.\lg^2 n)$, e portanto $T(n) = \Theta(n^2.\lg^2 n)$. □