Projeto e Análise de Algoritmos

Equações de Recorrência

Flávio L. C. de Moura*

1 A complexidade de algoritmos recursivos

1.1 Busca sequencial

Considere novamente o pseudocódigo da busca sequencial recursiva:

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

Denote por $T_{ss}(x, l)$ o número de comparações realizadas pelo algoritmo $seq\_search\ x\ l$, que pode ser definida como a seguir:

$$T_{ss}(x, l) := \begin{cases} 0, & \text{se } l = nil; \\ 1, & \text{se } l = h :: l' \text{ e } h = x; \\ 1 + T_{ss}(x, l'), & \text{se } l = h :: l' \text{ e } h \neq x. \end{cases}$$

Assim, se l é a lista vazia, nenhuma comparação é feita. Se l é uma lista não-vazia, digamos $h :: l'$, e x é igual a h (primeiro elemento da lista) então apenas 1 comparação é feita e o algoritmo para. Caso x não seja igual a h então recursivamente continuamos contando o número de comparações. Por exemplo, $T_{ss}(1, 1 :: 3 :: 5 :: nil) = 1$, $T_{ss}(2, 1 :: 3 :: 5 :: nil) = 3$, $T_{ss}(3, 1 :: 3 :: 5 :: nil) = 2$, etc. Ou seja, a função $T_{ss}(x, l)$ retorna o número exato de comparações realizadas pelo algoritmo seq_search durante a busca do elemento x na lista l.

Para fazermos a análise assintótica do algoritmo seq_search no pior caso, construiremos uma recorrência análoga à função T_{ss} que utiliza o tamanho da lista l como parâmetro.

$$T^w_{ss}(|l|) := \begin{cases} 0, & \text{se } l = nil; \\ 1 + T^w_{ss}(|l'|), & \text{se } l = h :: l'. \end{cases}$$

onde $|l|$ denota o número de elementos da lista l.

Assim, a função $T^w_{ss}(n)$ vai retornar o número de comparações necessárias para realizar a busca em uma lista com n elementos no pior caso:

$$T^w_{ss}(n) := \begin{cases} 0, & \text{se } n = 0; \\ 1 + T^w_{ss}(n-1), & \text{se } n > 0. \end{cases}$$

Esta recorrência pode ser resolvida utilizando o *método da substituição* que consiste na aplicação sucessiva da definição da recorrência até que sejamos capazes de inferir uma solução. A verificação da correção da solução pode ser feita por indução, como veremos a seguir. Assumindo que $n > 0$, temos

$$\begin{aligned} T^w_{ss}(n) &= T^w_{ss}(n-1) + 1 \\ &= T^w_{ss}(n-2) + 2 \\ &= T^w_{ss}(n-3) + 3 \\ &= \ldots \\ &= T^w_{ss}(n-n) + n = 0 + n = n. \end{aligned}$$

Logo, $T^w_{ss}(n) = n$. Agora, podemos utilizar indução para verificar que esta é, de fato, uma solução da recorrência. A base da indução é trivial porque quando $n = 0$ temos $T^w_{ss}(0) = 0$. Quando $n > 0$, temos que $T^w_{ss}(n) \stackrel{def.}{=} 1 + T^w_{ss}(n-1) \stackrel{h.i.}{=} 1 + (n-1) = n$ como queríamos mostrar. Em notação assintótica, acabamos de mostrar que $T^w_{ss}(n) = \Theta(n)$.

*flaviomoura@unb.br

1.2 Ordenação por inserção recursivo

Considere o pseudocódigo do algoritmo de ordenação por inserção recursivo:

$is\ nil = nil$

$is\ (h :: tl) = insert\ h\ (is\ tl)$

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

onde

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

Qual é o número de comparações realizadas pelo algoritmo de ordenação por inserção, isto é, pela função is, para ordenar uma lista l? Vamos denotar por $T_{is}()$ a função que faz esta contagem. Se l for a lista vazia então nenhuma comparação é feita, ou seja, $T_{is}(nil) = 0$. Se $l = h :: tl$ então é feita uma chamada à função ins, além da chamada recursiva à função is:

$$T_{is}(l) = \begin{cases} 0, & \text{se } l = nil \\ T_{is}(tl) + T_{ins}\ h\ (is\ tl), & \text{se } l = h :: tl \end{cases}$$

Observe que, $T_{is}(1 :: 2 :: 3 :: nil) = 2$, $T_{is}(3 :: 2 :: 1 :: nil) = 3$, $T_{is}(1 :: 2 :: 3 :: 4 :: nil) = 3$ e $T_{is}(4 :: 3 :: 2 :: 1 :: nil) = 6$, etc. Portanto o número de comparações pode ser diferente para listas de mesmo tamanho, o que é esperado pelas chamadas feitas à função ins. Como então definir a função $T^w_{is}(n)$ que nos dá um limite superior para o número de comparações feitas pelo algoritmo de ordenação por inserção para uma lista qualquer de tamanho n? Em outras palavras, qual a complexidade do pior caso para o algoritmo de ordenação por inserção? Sabemos que quando $n = 0$, nenhuma comparação é feita. Quando $n > 0$, o algoritmo é aplicado recursivamente na cauda da lista, isto é, em uma lista de tamanho $n - 1$, e é feita uma chamada à função ins cuja complexidade já conhecemos. Isto nos permite escrever a função $T^w_{is}(n)$ como a seguir:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + T^w_{ins}(n-1), & \text{se } n > 0 \end{cases}$$ que pode ser simplificada como a seguir, já que $T^w_{ins}(n) = n$:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + (n-1), & \text{se } n > 0 \end{cases}$$

Podemos usar o método da substituição para encontrarmos uma solução para esta recorrência, e em seguida utilizar indução para verificarmos se a solução está correta. Pelo método da substituição, podemos ir aplicando a definição da recorrência, assumindo que $n > 0$:

$$\begin{aligned} T^w_{is}(n) &= T^w_{is}(n-1) + (n-1) \\ &= T^w_{is}(n-2) + (n-2) + (n-1) \\ &= T^w_{is}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \\ &= T^w_{is}(n-n) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= \textstyle\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \end{aligned}$$

Para finalizar, precisamos utilizar indução em n para provar que $T^w_{is}(n) = \frac{n(n-1)}{2}$. Se $n = 0$, o resultado é trivial. Se $n > 0$ então, por definição, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1)$. A hipótese de indução, nos dá que $T^w_{is}(n-1) = \frac{(n-1)(n-2)}{2}$, e portanto, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1) \stackrel{h.i.}{=} \frac{(n-1)(n-2)}{2} + (n-1) = \frac{n(n-1)}{2}$.

Nossa conclusão, portanto, é que o algoritmo de ordenação por inserção recursivo é correto, e sua complexidade no pior caso é quadrática, assim como na versão não-recursiva.

Exercício 1.1. *Resolva as seguintes relações de recorrência:*

1. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$
2. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + 1/n$
3. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + ln(n)$

1.3 Mergesort

Algoritmos recursivos desempenham um papel fundamental em Computação. 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 subproblema 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 Newmann em 1945.

O algoritmo *mergesort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, que consiste das seguintes etapas:

1. **Divisão**: O algoritmo divide a lista (ou vetor) l recebida como argumento ao meio, obtendo as listas l_1 e l_2;

2. **Conquista**: O algoritmo é aplicado recursivamente às listas l_1 e l_2 gerando, respectivamente, as listas ordenadas l'_1 e l'_2;

3. **Combinação**: O algoritmo combina as listas l'_1 e l'_2 através da função *merge* que então gera a saída do algoritmo.

Por exemplo, ao receber a lista $(4 :: 2 :: 1 :: 3 :: nil)$, este algoritmo inicialmente divide esta lista em duas sublistas, a saber $(4 :: 2 :: nil)$ e $(1 :: 3 :: nil)$. O algoritmo é aplicado recursivamente às duas sublistas para ordená-las, e ao final deste processo, teremos duas listas ordenadas $(2 :: 4 :: nil)$ e $(1 :: 3 :: nil)$. Estas listas são, então, combinadas para gerar a lista de saída $(1 :: 2 :: 3 :: 4 :: nil)$.

Algorithm 1: mergesort(A, p, r)

if $p < r$ **then**
    $q = \lfloor \frac{p+r}{2} \rfloor$;
    mergesort(A, p, q);
    mergesort$(A, q + 1, r)$;
    merge(A, p, q, r);
end

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.

Algorithm 2: merge(A, p, q, r)

$n_1 = q - p + 1$; `// 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

Exercício 1.2. *Prove que o algoritmo merge é correto.*

Exercício 1.3. *Prove que o algoritmo mergesort é correto.*

Exercício 1.4. *Faça a análise assintótica do algoritmo merge.*

Exercício 1.5. *Faça a análise assintótica do algoritmo mergesort.*

2 O Teorema Mestre

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [2]:

Definição 2.1. *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 2.2. *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*

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

Exercício 2.3. *Mostre que $f(n) = n$ é suave.*

Exercício 2.4. *Mostre que $f(n) = \lg n$ é suave.*

Exercício 2.5. *Mostre que $f(n) = n.\lg n$ é suave.*

Exercício 2.6. *Mostre que $f(n) = 3^n$ não é suave.*

Lema 2.7. *Sejam c e n_0 constantes positivas, e $f(n)$ uma função tal que $f(2n) \leq c.f(n), \forall n \geq n_0$. Então $f(2^k n) \leq c^k.f(n), \forall n \geq n_0$ e $k \geq 1$.*

Teorema 2.8. *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 2.9. *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 2.10. *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*

$T(n) = a.T(n/b) + f(n)$, *para* $n = b^k, k = 1, 2, 3, \ldots$

$T(1) = c$

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

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

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

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

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

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

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

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

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[1]. 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 2.11. *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 $T(n) = \Theta(f(n))$.*

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

Lema 2.12. *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:*

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

onde i é um inteiro positivo. Então

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

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 2.13. *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:*

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

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 $g(n) = \Theta(n^{\log_b a}. \lg n)$;*

3. *Se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $g(n) = \Theta(f(n))$.*

Prova. Exercício. □

Exercício 2.14. *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. $T(1) \in \Theta(1)$, $T(n) = 3T(n/2) + nln(n)$
13. $T(1) \in \Theta(1)$, $T(n) = 3T(n/4) + nln(n)$
14. $T(1) \in \Theta(1)$, $T(n) = 2T(n/2) + nln(n)$
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] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

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