Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

16 de outubro de 2023

1 Equações de Recorrência

Iniciaremos o estudo das equações de recorrência fazendo a análise da complexidade do algoritmo de ordenação por inserção recursivo. A função auxiliar $ins\ x\ l$ é utilizada para inserir o elemento x na lista l:

Definição 1.1. *Sejam x um número natural, e l uma lista de números naturais. A função $(ins\ x\ l)$, que insere o natural x na lista l, é definida recursivamente como a seguir:*

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

Definiremos o algoritmo de ordenação por inserção pela função recursiva is (*insertion sort*) que recebe uma lista l como argumento:

Definição 1.2. *Seja l uma lista de números naturais. A função is é definida recursivamente como a seguir:*

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

Se l for a lista vazia não há nada a fazer, e caso contrário, recursivamente ordenamos a cauda da lista para então inserir o novo elemento. Ou seja, dada uma lista não vazia $h :: tl$, a função is insere o primeiro elemento h na versão ordenada da cauda tl.

Vamos iniciar nossa análise pela função $ins\ x\ l$ (Definição 1.1). A operação básica no caso do algoritmo de ordenação por inserção é a comparação entre chaves. Note que quando l é a lista vazia, a lista unitária $x :: nil$ é retornada, e nenhuma comparação é feita. Quando l é uma lista da forma $h :: tl$ então comparamos x com h, e quando $x \leq h$, ou seja, após uma comparação, a lista $x :: h :: tl$ é retornada e o algoritmo termina. Por outro lado, se $x > h$, ou seja, após uma comparação, o algoritmo continua buscando recursivamente a posição correta para inserir x. Denotaremos por $T_{ins}\ x\ l$ a função que computa o número exato de operações básicas realizadas pela função ins para inserir o elemento x na lista l:

$$T_{ins}\ x\ l = \begin{cases} 0, & \text{se } l = nil \\ 1, & \text{se } l = h :: tl \text{ e } x \leq h \\ 1 + (T_{ins}\ x\ tl), & \text{se } l = h :: tl \text{ e } x > h \end{cases}$$

Observe que esta função pode retornar valores distintos para listas de mesmo tamanho. De fato, temos que $T_{ins}\ 1\ (2 :: 3 :: 4 :: nil) = 1$, enquanto que $T_{ins}\ 4\ (1 :: 2 :: 3 :: nil) = 3$. Normalmente estamos interessados na análise do pior caso, porque ela nos fornece uma cota superior para o custo do algoritmo, isto é, para o número de comparações realizadas durante a sua execução. Para calcularmos o número de comparações no pior caso, definiremos a função $T^w_{ins}(n)$ que retorna o número máximo de comparações realizadas pelo algoritmo is ao receber como entrada uma lista contendo $n \geq 0$ elementos:

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

Assim, se $n = 0$ (lista vazia) então nenhuma comparação é feita, e se $n > 0$ então o elemento a ser inserido é comparado com o primeiro elemento da lista, e recursivamente, computamos o custo $T^w_{ins}(n-1)$ para fazer a inserção na cauda da lista original.

*flavio@flaviomoura.info

Exercício 1.3. *Prove que $T^w_{ins}(n) = n$, para todo $n \geq 0$.*

Assim, a relação entre as funções T_{ins} e T^w_{ins} é dada pelo lema a seguir:

Exercício 1.4. *Sejam x um número natural, e l uma lista de números naturais. Prove que $T_{ins}\ x\ l \leq T^w_{ins}(|l|)$, onde $|l|$ denota o tamanho da lista l.*

Os dois últimos exercícios nos permitem concluir que $T_{ins}\ x\ l \leq |l|$, ou seja, que a função ins tem complexidade linear no pior caso: $T^w_{ins}(n) = O(n)$

Agora faremos uma análise semelhante para a função is. A função recursiva $T_{is}(l)$ a seguir, retorna o número de comparações realizadas pelo algoritmo is para ordenar a lista l:

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

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 $n > 0$:

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

Podemos continuar este processo de substituição até chegarmos em $T^w_{is}(1)$ que é igual a 0:

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

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, $T^w_{is}(n) = T^w_{is}(n-1)+(n-1) \overset{h.i.}{=} \frac{(n-1)(n-2)}{2}+(n-1) = \frac{n(n-1)}{2}$.

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: $T^w_{is}(n) = O(n^2)$.

Exercício 1.5. *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. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + ln(n)$

Vejamos um outro exemplo, o algoritmo *mergesort*:

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.

Algorithm 1: mergesort(A, p, r)

```
1  if p < r then
2  |  q = ⌊(p+r)/2⌋;
3  |  mergesort(A, p, q);
4  |  mergesort(A, q + 1, r);
5  |  merge(A, p, q, r);
6  end
```

Denotaremos por $T_{ms}(n)$ a função que retorna o número de comparações realizadas pelo algoritmo *mergesort* ao receber como entrada um vetor contendo n elementos, ou seja, $r - p + 1 = n$ que pode ser definida recursivamente como a seguir:

$$T_{ms}(n) = \begin{cases} 2.T_{ms}(n/2) + T_{merge}(n), & \text{se } n > 1 \\ 0, & \text{caso contrário.} \end{cases}$$

Observe que esta recorrência depende do custo da função auxiliar merge que é responsável por combinar os subvetores ordenados. De fato, o algoritmo merge consiste na 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)

```
1  n1 = q − p + 1 ;                          // Qtd. de elementos em A[p..q]
2  n2 = r − q ;                              // Qtd. de elementos em A[q + 1..r]
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays;
4  for i = 1 to n1 do
5  |  L[i] = A[p + i − 1];
6  end
7  for j = 1 to n2 do
8  |  R[j] = A[q + j];
9  end
10 L[n1 + 1] = ∞;
11 R[n2 + 1] = ∞;
12 i = 1;
13 j = 1;
14 for k = p to r do
15 |  if L[i] ≤ R[j] then
16 |  |  A[k] = L[i];
17 |  |  i = i + 1;
18 |  end
19 |  else
20 |  |  A[k] = R[j];
21 |  |  j = j + 1;
22 |  end
23 end
```

As comparações realizadas pelo algoritmo merge ocorrem na linha 15 do pseudocódigo acima. O número de comparações é definido pelo laço **for** da linha 14, ou seja, são realizadas $r - p + 1$ comparações na linha 15. Se $n = r - p + 1$ então $T_{merge}(n) = n$. Podemos então reescrever a recorrência anterior da seguinte forma:

$$T_{ms}(n) = \begin{cases} 2.T_{ms}(n/2) + n, & \text{se } n > 1 \\ 0, & \text{caso contrário.} \end{cases}$$

Aqui podemos utilizar novamente o método da substituição para resolver esta recorrência. As contas podem ser simplificadas ao resolvermos para o caso em que n é uma potência de 2, ou seja, $n = 2^k$ $(k \geq 0)$.

Exercício 1.6. *Prove que* $T_{ms}(n) = \Theta(n. \lg n)$*, quando* $n = 2^k$ $(k \geq 0)$.

A seguir, mostraremos que a solução apresentada no exercício anterior continua verdadeira para qualquer valor de n, e não apenas para potências de 2.

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

Definição 1.7. *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 1.8. *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 1.9. *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 1.10. *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 1.11. *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 1.12. *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*
$T(n) = a.T(n/b) + f(n), \quad$ *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

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

Demonstração. 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 1.13. *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 $T(n) = \Theta(n^{\log_b a}.\lg n)$;*
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 1.14. *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:*

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

onde i é um inteiro positivo. Então

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

Demonstração. 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 1.15. *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:*

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

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* $g(n) = \Theta(f(n))$.

Demonstração. Exercício. □

Exercício 1.16. *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. $T(1) \in \Theta(1)$, $T(n) = 3T(n/2) + nln(n)$
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.