Complexidade de Algoritmos

Prof. Dr. Leandro Balby Marinho

UFCG CEEI Departamento de Sistemas e Computação

Análise e Técnicas de Algoritmos

Roteiro

Introdução

- Como analizar o custo de um algoritmo?
  - Mensurar os recursos necessários em termos de:
    - tempo (foco desse curso)
    - espaço
- Como o tempo de execução escala com o tamanho da entrada?
- Considere as seguintes funções representando o custo de dois algoritmos para uma dada entrada de tamanho n
  - Algoritmo 1: $100n^2 + 17n + 4$
  - Algoritmo 2: n^3
- Qual o algoritmo mais eficiente?

Calculando Tempo de Execução

- Como calcular o tempo de execução de um algoritmo ?
- Contamos o número de vezes que cada operação é realizada, mas
  - difícil e desnecessário.
- Identificar a **operação básica**, i.e., a operação mais custosa. Por exemplo,
  - Algoritmos de ordenação: comparação.
  - Multiplicação de matrizes: adição e multiplicação.
- Agora contamos quantas vezes que a operação básica - que executa em tempo c_{op} - é realizada em uma entrada de tamanho n, i.e.,

$$T(n) \approx c_{op} C(n) \tag{1}$$

onde $C(n)$ é o número de vezes que a operação é realizada.

Calculando Tempo de Execução

- Agora podemos responder perguntas do tipo:
  1. Em quanto tempo esse algoritmo executa em uma máquina que é duas vezes mais lenta que a minha?
  2. Quanto tempo a mais roda um algoritmo com o dobro do tamanho da entrada original?

Exercício 1: Responda a segunda pergunta assumindo

$$C(n) := \frac{1}{2}n(n-1)$$

Roteiro

Ordem de Grandeza

- Considere as funções $f(x) = x$ e $g(x) = x^2$
  - Para valores cada vez maiores de x, a diferença entre os valores de f e g é cada vez maior.
  - Essa diferença não vai sumir apenas multiplicando f por uma constante, não importa quão grande.
  - Isso indica que f e g se comportam fundamentalmente diferentes em relação às suas taxas de crescimento.
- Funções que apresentam taxas de crescimento similares, possuem mesma ordem de grandeza.
- Algoritmos são comparados em termos das ordens de grandeza de suas funções de custo e não na forma exata da função.

Intuição

- Podemos pensar em funções como meios de transporte, tal que
  - funções de mesma ordem de grandeza representam o mesmo meio de transporte.
  - uma ordem de grandeza (ou classe) representa andar a pé, outra andar de carro, e outra viajar de avião.
  - velocidades dentro de uma mesma modalidade são aproximadamente iguais, e.g., andar e correr, Jipe e Vectra.
- Note que andar (a qualquer velocidade) é muito diferente de dirigir, que é muito diferente de voar.

Tamanho de Entrada vs. Função de Custo

n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^3	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Algoritmos com custo exponencial são inviáveis para problemas com entradas médias/grandes.

Pior, Melhor e Caso Médio

- A complexidade de um algoritmo depende do *tamanho* e do *formato* da entrada.
- No **pior caso**, um algoritmo realiza o maior número de operações possíveis para resolver um problema.
  - Mais útil para análise pois não faz nenhum tipo de suposição sobre o formato de entrada.
- O **melhor caso** é o inverso do pior caso.
- O **caso médio** corresponde ao número médio de operações usadas para resolver o problema sob todas as instâncias do problema para um determinado tamanho de entrada.

Exemplo 1

Considere o algoritmo BUSCALINEAR abaixo que busca por um dado item k numa lista de n elementos.

```
BUSCALINEAR(A, k)
    // Input: Um array A[1..n] e uma chave de busca k.
    // Output: O indice do primeiro elemento em A que casa com k, ou 0
    i = 1
    while i ≤ n and A[i] ≠ k
        i = i + 1
    if i ≤ n
        return i
    else return 0
```

Qual o melhor, pior e caso médio desse algoritmo?

Exemplo 1 Cont.

O melhor caso se dá quando $A[1] = k$ e o pior quando k não está em A.

Para o caso médio, note que, para uma dada entrada de tamanho n, há x casos possíveis quando k está no array. Se $A[1] = k$, precisamos de 3 comparações, se $A[2] = k$, adicionamos mais duas, e assim por diante. Portanto, o número médio de comparações usadas é igual a:

$$\frac{3 + 5 + 7 + \ldots + (2x + 1)}{x} = \frac{2(1 + 2 + 3 + \ldots + x)x}{x}$$

Como $1 + 2 + 3 + \ldots + x = \dfrac{x(x+1)}{2}$, o custo médio é

$$\frac{2[x(x + 1/2)]}{x} + 1 = x + 2$$

Roteiro

Big-O: Intuição

A **análise assintótica** estuda o comportamento de funções para valores aribitrariamente grandes de suas variáveis.

- Seja $g(n)$ uma função não negativa definida em $\mathbb{R}^+$.
- $O(g(n))$ representa o conjunto de funções de menor ou mesma ordem de grandeza que $g(n)$ (vezes uma constante).
- As seguintes afirmações são verdadeiras:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad n(n-1) \in O(n^2)$$

- Por outro lado, note que :

$$n^2 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2)$$

Big-Ω: Intuição

- $\Omega(g(n))$ representa o conjunto de funções de maior ou mesma ordem de grandeza que $g(n)$ (vezes uma constante).
- As seguintes afirmações são verdadeiras:

$$n^3 \in \Omega(n^2), \quad n(n-1) \in \Omega(n^2), \quad 200n + 10 \notin \Omega(n^2)$$

Big-Θ: Intuição

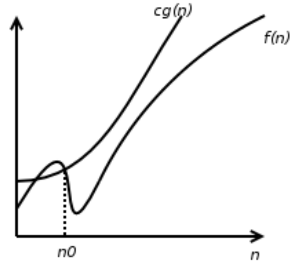
- $\Theta(g(n))$ representa o conjunto de funções que tem a mesma ordem de grandeza que $g(n)$ (vezes uma constante).
- As seguintes afirmações são verdadeiras:

$$n^2 + 3 \in \Theta(n^2), \quad n^2 + \log n \in \Theta(n^2), \quad 200n + 10 \notin \Theta(n^2)$$

Big-O: Definição

Big-O

Sejam f e g funções de $\mathbb{R} \to \mathbb{R}^+$. Então $f \in O(g)$, se existem constantes positivas n_0 e c tais que, para $n > n_0$, $f(n) \leq cg(n)$.

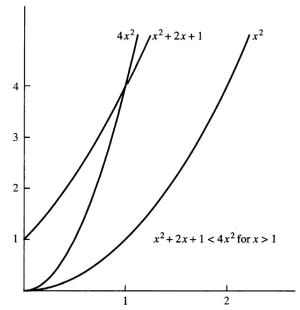


Exemplo 2

Mostre que $f(x) = x^2 + 2x + 1$ é $O(x^2)$ usando a definição.

- Ideia: escolha n_0 para o qual seja fácil estimar $f(x)$ para $x > n_0$.
- Para $n_0 = 1$, por exemplo, sabemos que $x < x^2$ e $1 < x^2$ quando $x > 1$. Portanto,

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$



Exercícios

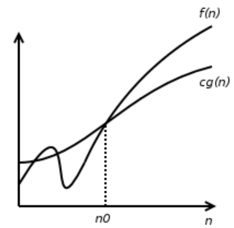
Exercício 2: Determine outro par de constante n_0 e c para o exemplo 2 do slide anterior

Exercício 3: Mostre que $100n + 5 \in O(n^2)$ usando a definição

Big-Ω: Definição

Big-Ω

Sejam f e g funções de $\mathbb{R}^+ \to \mathbb{R}^+$. Então $f \in \Omega(g)$, se existem constantes positivas n_0 e c tais que, para $n > n_0$, $f(n) \geq cg(n)$.



Exemplo 3

Mostre que a função $f(x) = 8x^3 + 5x^2 + 7$ é $\Omega(x^3)$.

Note que $f(x) \geq g(x)$ para todos os reais positivos x. Portanto o par $(n_0, c) := (1, 1)$, por exemplo, satisfaz a confição.

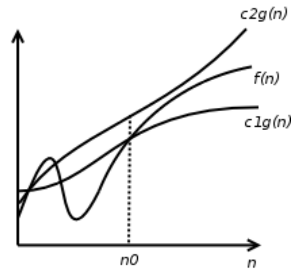
Exercício 4: Mostre que $n^3 \in \Omega(n^3)$.

Big-Θ: Definição

Big-Θ

Sejam f e g funções de $\mathbb{R}^+ \to \mathbb{R}^+$. Então $f \in \Theta(g)$, se existirem constantes positivas k, c_1 e c_2 tais que, se $n > n_0$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$



Exemplo 4

Seja $f(n) := \frac{1}{2}n(n-1)$, mostre que $f(n) \in \Theta(n^2)$.

Primeiro provamos que $f(n) \in O(n^2)$

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad (\text{para } n \geq 0)$$

Em seguida, mostramos que $f(n) \in \Omega(n^2)$

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \geq \frac{1}{4}n^2 \quad (\text{para } n \geq 2)$$

Portanto podemos selecionar $c_1 := \frac{1}{2}$, $c_2 := \frac{1}{4}$ e $n_0 := 2$.

Ordem de Grandeza de Polinômios

Teorema 1

Seja $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, onde $a_0, a_1, \ldots, a_n$ são números reais com $a_n \neq 0$. Então $f(x)$ é de ordem x^n.

Exemplo 5: Os polinômios

$$3x^8 + 10x^7 + 221x^2 + 1444, \; x^{19} - 18x^4 - 10,112$$

estão na ordem de $\Theta(x^8)$ e $\Theta(x^{19})$ respectivamente.

Limites para Comparar Ordens de Grandeza

O limite da razão das funções em questão pode levar a três casos:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} := \begin{cases} 0, & f(n) \text{ tem ordem de grandeza menor que } g(n) \\ c > 0, & f(n) \text{ tem mesma ordem de grandeza que } g(n) \\ \infty, & f(n) \text{ tem ordem de grandeza maior que } g(n) \end{cases}$$

Caso tenhamos uma indeterminação da forma $\frac{\infty}{\infty}$, usamos a regra de L'Hôpital:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} := \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Exemplo 6

Compare as ordens de grandeza de $\frac{1}{2}n(n-1)$ e n^2.

$$\begin{aligned} \lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2} \lim_{n\to\infty} \frac{n^2 - n}{n^2} \\ &= \frac{1}{2} \lim_{n\to\infty} \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{2} \end{aligned}$$

Como o limite é igual a uma constante positiva,

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

Soma de Ordens de Grandeza

A seguinte propriedade é útil para analizar algoritmos que realizam duas partes consecutivas de execução.

Teorema 2

Se $f_1(n) \in O(g_1(n))$ e $f_2(n) \in O(g_2(n))$, então

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

O mesmo vale para Ω e Θ.

Exemplo 7

Assuma um algoritmo que verifica se dois array são idênticos (possuem os mesmos elementos). Uma abordagem seria:

1. ordene o array aplicando algum algoritmo de ordenação conhecido;
2. percorra os arrays sequencialmente comparando os elementos um a um.

Assumindo que o custo de (1) é $O(n^2)$ e custo de (2) $O(n)$, o custo do algoritmo é dado por

$$O(\max\{n^2, n\}) = O(n^2)$$

Roteiro

1. Introdução
2. Ordem de Grandeza de Funções
3. Notação Assintótica
4. Complexidade em Algoritmos Iterativos
5. Análise de Algoritmos Recursivos

Análise de Complexidade

1. Escolha uma unidade para medir o tamanho da entrada.
2. Identifique a operação básica do algoritmo.
3. Expresse o número de execuções da operação básica por um somatório.
4. Ache uma forma fechada para o somatório.
5. De (4) derive a ordem de grandeza do algoritmo.

Identidades Úteis Envolvendo Somatórios

$$\sum_{i=l}^{u} cx_i = c\sum_{i}^{u} x_i$$

$$\sum_{i=l}^{u}(x_i \pm y_i) = \sum_{i=l}^{u} x_i \pm \sum_{i=l}^{u} y_i$$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

$$\sum_{i=1}^{u} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

Exemplo 8

Determine a complexidade do algoritmo abaixo que retorna o maior elemento em um vetor de inteiros.

```
MAXIMO(A, n)
max = A[1]
for i = 2 to n
    if A[i] > max
        max = A[i]
return max
```

Exemplo 8 Cont.

- Tamanho da entrada = número de elementos do array A.
- Operação básica = comparação (linha 3).
- Quantidade de operações

$$C(n) := \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Exercício 5

Determine a complexidade do algoritmo abaixo que verifica se um array possui elementos distintos entre si.

```
ElementoDistinto(A, n)
for i = 1 to n − 1
    for j = i + 1 to n
        if A[i] == A[j]
            return false
return true
```


Exercício 6

Determine a complexidade do algoritmo de ordenação Bolha abaixo. O algoritmo recebe um array A de números reais com n elementos, onde $n \geq 2$.

```
Bolha(A, n)
for i = 1 to n − 1
    for j = 1 to n − i
        if A[j] > A[j + 1]
            temp = A[j]
            A[j] = A[j + 1]
            A[j + 1] = temp
```

Qual o custo no melhor caso desse algoritmo?

Roteiro

Análise de Algoritmos Recursivos

- Algoritmos iterativos
  - complexidade expressa através de somatórios.
- Algoritmos recursivos
  - complexidade expressa através de recorrências.
- Veremos três formas de resolver recorrências:
  - Método da Substituição
  - Árvore de Recorrência
  - Método Mestre

Análise de Complexidade

1. Escolha uma unidade para medir o tamanho da entrada.
2. Identifique a operação básica do algoritmo.
3. Expresse o número de execuções da operação básica por uma recorrência.
4. Ache uma forma fechada para a recorrência.
5. De (4) derive a ordem de grandeza do algoritmo.

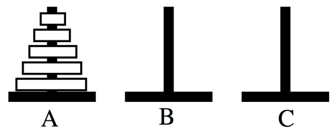
Método da Substituição

- O método da substituição envolve dois passos:
  1. Pressupor a solução da recorrência.
  2. Provar que a suposição é correta por indução.
- É um método eficiente quando é fácil pressupor a solução.
- Veremos duas formas de pressupor a solução:
  - Expansão
  - Árvores de Recorrência

Exemplo 9

Considere um procedimento recursivo para resolver o problema da Torre de Hanoi (movendo os pinos de A a C).

1. Resolva recursivamente o problema de mover os top $n-1$ discos do pino A para o pino B.
2. Mova o disco n ao pino C.
3. Resolva recursivamente o problema de mover os $n-1$ discos do pino B para o pino C.

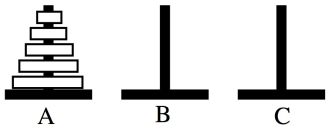


Quantos movimentos são realizados para n discos?

Exemplo 9

Considere um procedimento recursivo para resolver o problema da Torre de Hanoi (movendo os pinos de A a C).

1. Resolva recursivamente o problema de mover os top $n-1$ discos do pino A para o pino B. (**$\mathbf{T(n-1)}$ movimentos**)
2. Mova o disco n ao pino C. (**1 movimento**)
3. Resolva recursivamente o problema de mover os $n-1$ discos do pino B para o pino C. (**$\mathbf{T(n-1)}$ movimentos**)



Quantos movimentos são realizados?

Exemplo 9 Cont.

Como para 0 discos realizamos 0 movimentos, o número total de movimentos pode ser dado pela seguinte relação de recorrência:

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

Como achar a forma fechada da recorrência?

Exemplo 9 Cont.

Usamos a abordagem expandir, conjecturar e verificar. Então, expandindo aplicando a definição para n, $n-1$, $n-2$, etc.:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2[2T(n-2)+1]+1 = 2^2T(n-2)+3 \\ &= 2^2[2T(n-3)+1]+3 = 2^3T(n-3)+7 \\ &= 2^3[2T(n-4)+1]+7 = 2^4T(n-4)+15 \end{aligned}$$

Note que após k expansões, a equação tem a forma

$$T(n) = 2^kT(n-k)+2^k-1$$

Exemplo 9 Cont.

A expansão tem que parar quando $n-k=0$, ou seja, quando $k=n$. Nesse ponto temos:

$$\begin{aligned} T(n) &= 2^nT(n-n)+2^n-1 \\ &= 2^nT(0)+2^n-1 \\ &= 2^n0+2^n-1 \\ &= 2^n-1 \end{aligned}$$

Agora precisamos provar que a conjectura é verdadeira.

Exemplo 9 cont.

Podemos verificar isso por indução.

- Base: $T(0) = 2^0 - 1 = 0$, que é verdade pela recorrência.
- Hipótese: $T(k) = 2^k - 1$
- Mostrar: $T(k+1) = 2^{k+1} - 1$

$$T(k+1) = 2T(k) + 1$$
$$T(k+1) = 2(2^k - 1) + 1$$
$$T(k+1) = 2^{k+1} - 1$$

E portanto fica provada a relação de recorrência.

Exemplo 10

Considere o algoritmo fatorial recursivo abaixo.

```
F(n)
if n == 0
    return 1
else
    return F(n − 1) · n  para n > 0
```

- A operação básica é o número de multiplicações indicado por $M(n)$.
- O número de multiplicações é definido por:

$$M(n) = \underbrace{M(n-1)}_{\text{para calcular } \mathrm{F}(n-1)} + \underbrace{1}_{\text{para calcular } \mathrm{F}(n-1)\cdot n}$$

Exemplo 10 cont.

Incluindo a condição inicial da sequencia, ou o caso base do algoritmo, temos a seguinte recorrência:

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

Expandindo, $M(n)$ temos:

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3 \end{aligned}$$

Note que após k expansões a equação tem a forma

$$M(n) := M(n-k) + k$$

Exemplo 10 cont.

A expansão para quando $n - k = 0$, ou seja, quando $k = n$. Nesse ponto temos:

$$\begin{aligned} M(n) &= M(n-n) + n \\ &= 0 + n \\ &= n \end{aligned}$$

Agora falta provar por indução. O caso base $M(0) = 0$ é verdadeiro pois quando $n = 0$ o algoritmo não realiza nenhuma multiplicação. Agora supomos que $M(k) = k$ e tentamos provar que $M(k+1) = k+1$.

$$\begin{aligned} M(k+1) &= M(k) + 1 \\ &= k + 1 \end{aligned}$$

E portanto o algoritmo executa em $M(n) = n$.

Exercício 7

Nem sempre é fácil pressupor a solução da recorrência através da expansão. Nesses casos, pode-se

- Tentar uma abordagem exaustiva de tentativa e erro.
- Utilizar soluções conhecidas para recorrências similares.

Resolva a relação de recorrência abaixo pressupondo

$$T(n) := n \lg n + n$$

$$T(n) := \begin{cases} 1 & \text{se } n = 1, \\ 2T(n/2) + n & \text{se não.} \end{cases}$$

Dividir-para-Conquistar

- Muitos algoritmos recursivos seguem uma abordagem **dividir-para-conquistar**:
  - **Divida** o problema em vários subproblemas mais simples.
  - **Conquiste** os subproblemas recursivamente.
  - **Combine** as soluções intermediárias.

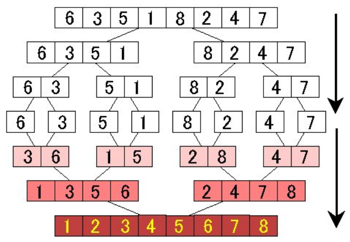
Análise de Algoritmos Dividir-para-Conquistar

- Se o tamanho do problema é $n \leq c$ para uma constante c suficientemente pequena, a solução mais simples executa em $\Theta(1)$.
- Suponha que a divisão do problema gere a subproblemas, cada um tendo $1/b$ do tamanho original.
- Se levarmos o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções intermediárias, teremos a recorrência:

$$T(n) := \begin{cases} \Theta(1), & \text{se } n \leq c, \\ aT(n/b) + D(n) + C(n), & \text{senão.} \end{cases}$$

Exemplo 11

Considere o algoritmo MergeSort.



- **Dividir:** Divide a lista de n elementos em duas listas de $n/2$ elementos cada.
- **Conquistar:** Ordena cada subsequência recursivamente.
- **Combinar:** Combina as subsequências ordenadas.

O Algoritmo Merge

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
cria arrays L[1 .. n1 + 1] e R[1 .. n2 + 1]
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Note que cada passo de ordenação executa em $\Theta(1)$, portanto o algoritmo executa em $\Theta(n)$ para n passos de ordenação.

O Algoritmo Merge-Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Se $p = r$ o array tem apenas um elemento e portanto já está ordenado.
- Senão, o passo de divisão calcula um índice q que particiona $A[p\,..\,r]$ nos subarrays $A[p\,..\,q]$, contentdo $\lceil n/2 \rceil$ elementos, e $A[q+1\,..\,r]$, contendo $\lfloor n/2 \rfloor$ elementos.

Análise do Merge Sort

- **Dividir:** Esse passo apenas calcula o índice representando o meio do subarray. Portanto $D(n) = \Theta(1)$.
- **Conquistar:** Dois problemas são resolvidos recursivamente, cada um de tamanho $n/2$, que contribuem $2T(n/2)$ ao tempo de execução.
- **Combinar:** Já vimos que o algoritmo MERGE executa em $\Theta(n)$ para um subarray de n elementos. Portanto $C(n) = \Theta(n)$.
- Isso nos dá a seguinte relação de recorrência[1]:

$$T(n) := \begin{cases} \Theta(1), & \text{se } n = 1, \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases}$$

[1] Note que $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$.

Notação Assinótica

- Até agora tínhamos uma recorrência como uma função exata.
- Normalmente usamos notação assintótica para descrever recorrências.
- Exemplo: $T(n) := 2T(n/2) + \Theta(n)$, com solução $T(n) := \Theta(n \lg n)$
- Normalmente não nos preocupamos com casos base.

Notação Assinótica

- Para resolver recorrências com notação assintótica Θ pelo método da substituição:
  - substitua a notação assintótica por uma constante descrevendo o limite superior (ou inferior) que se quer provar.
  - mostre os limites superior (O) e inferior (Ω) separadamente.

Exercício 8: Resolva a recorrência $T(n) = 2T(n/2) + \Theta(n)$ pelo método da substituição.

Árvore de Recorrência

- Traçar uma árvore de recorrência pode ajudar a pressupor a solução da recorrência.
- Cada nó representa o custo de um único subproblema entre as chamadas recursivas.
- Somamos os custos dentro de cada nível da árvore para obter um conjunto de custos por nível.
- Para resolver a recorrência, somamos os custos de cada nível da árvore.
- Uma árvore de recorrência é normalmente usada em conjunto com o método da substituição.

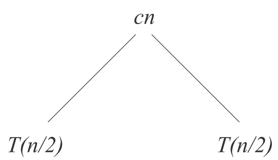
Árvore de Recorrência do Merge Sort

Podemos reescrever a recorrência como

$$T(n) := \begin{cases} c, & \text{se } n = 1, \\ 2T(n/2) + cn, & \text{se } n > 1. \end{cases}$$

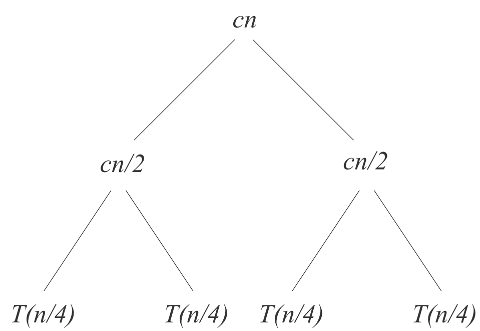
onde c representa o tempo de execução do caso base e da divisão e combinação por elemento do array.

O problema original tem um custo de cn, mais dois subproblemas, cada um com custo $T(n/2)$:



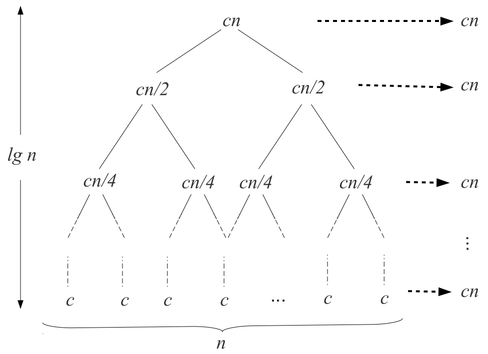
Árvore de Recorrência do Merge Sort

Para cada um dos subproblemas de tamanho $n/2$, temos um custo de $cn/2$, mais dois subproblemas, cada um com custo $T(n/4)$:



Árvore de Recorrência do Merge Sort

Continue expandindo até que o tamanho do problema seja igual a 1:



Ao descer cada nível, o nr. de subproblemas dobra, mas o custo por subproblema cai pela metada ⇒ o custo por nível permanece o mesmo.

Árvore de Recorrência do Merge Sort

- A árvore tem $\lg n + 1$ níveis (Mostre isso).
- Portanto, se cada nível tem um custo cn teremos custo total $cn(\lg +1) = cn \lg n + cn$.
- Ignorando os termos de menor ordem e a constante c chegamos finalmente a $\Theta(n \lg n)$.

O Método Mestre

- O método mestre fornece uma "receita de bolo" para resolver recorrências do tipo
$$T(n) = aT(n/b) + f(n)$$
onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ uma função assintótica positiva.[2]
- O método mestre depende do teorema a seguir.

[2] Representa o custo de dividir e combinar, ou seja, $f(n) = D(n) + C(n)$

O Método Mestre

Teorema Mestre

Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função, e $T(n)$ definida nos inteiros não negativos pela recorrência
$$T(n) = aT(n/b) + f(n)$$
onde interpretamos n/b como $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então $T(n)$ pode ser limitada assintoticamente como segue

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

O Método Mestre

- Nos três casos estamos comparando a função $f(n)$ com a função $n^{\lg_b a}$.
- A solução da recorrência é dada pela maior das duas funções:
  - No caso 1 do teorema a função $n^{\lg_b a}$ é maior, então $T(n) = \Theta(n^{\lg_b a})$.
  - No caso 3 a função $f(n)$ é maior, então $T(n) = \Theta(f(n))$.
  - No caso 2 as funções tem a mesma dimensão, multiplicamos por um fator logarítmico e teremos

$$T(n) = \Theta(n^{\lg_b a} \lg n) = \Theta(f(n) \lg n)$$

Exemplos

Exemplo 12: Resolva $T(n) := 9T(n/3) + n$ em termos de Θ.

- Como $a = 9$, $b = 3$, $f(n) = n$, comparamos n com $n^{\lg_3 9} = n^2$.
- Como $f(n) \in O(n^{2-\epsilon})$ para $\epsilon = 1$,

$$f(n) \in \Theta(n^2)$$

Exercício

Exercício 9: Use o método mestre para resolver as seguintes recorrências

a) $T(n) = 4T(n/2) + n$

b) $T(n) = 4T(n/2) + n^2$

Referências

Anany Levitin. Introduction to the Design and Analysis of Algorithms. Segunda Edição. Pearson International Edition, 2007.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. The MIT Press, 2a edição, 2001.