Projeto e Análise de Algoritmos II
Revisão de tópicos de Análise

Antonio Luiz Basile

Faculdade de Computação e Informática
Universidade Presbiteriana Mackenzie

March 5, 2018

Pequena revisão de Análise de Algoritmos

- Suponha que os computadores são infinitamente rápidos e de custo nenhum para memória.
- Será que ainda haveria algum motivo para estudar análise de algoritmos?
- A resposta é sim !
- Para demonstrar que sua solução termina e com a resposta certa.

Eficiência

- Algoritmos distintos para resolver o mesmo problema frequentemente diferem muito em eficiência.
- Estas diferenças podem ser muito mais significantes que diferenças de software e hardware.
- Por exemplo, para o problema de ordenação podemos comparar o insertion-sort com o mergesort.
- O primeiro demora tempo proporcional a n^2.
- O segundo demora tempo proporcional a $n \log n$.

Exemplo Prático

- Suponha o computador A mais rápido rodando Insertion Sort e computador B mais lento rodando Mergesort.
- Suponha que estejam rodando sobre um vetor de 10 milhões de elementos.
- Suponha que o computador A execute 10 bilhões de operações por segundo e que o computador B execute 10 milhões de operações por segundo. Portanto o computador A é 1000 vezes mais rápido que o computador B.
- Suponha que o Insertion Sort escrito em A é o mais otimizado possível e que requeira apenas $2n^2$ para ordenar n elementos. Por outro lado, suponha que um péssimo Mergesort esteja rodando em B e que demore $50n \lg n$ instruções.
- O computador A demorará $5,5$ horas e o computador B menos de 20 minutos (~ 17 vezes mais rápido).

Insertion Sort - Análise

Em geral, o tempo que leva um algoritmo cresce com o tamanho da entrada, logo é comum descrever o tempo do programa em função do tamanho de sua entrada.

ORDENAÇÃO-POR-INSERÇÃO (A, n)

para j crescendo de 2 até n faça	n
$x \leftarrow A[j]$	$n-1$
$i \leftarrow j-1$	$n-1$
enquanto $i > 0$ e $A[i] > x$ faça	$2+3+\ldots+n$
$A[i+1] \leftarrow A[i]$	$1+2+3+\ldots+n-1$
$i \leftarrow i-1$	$1+2+3+\ldots+n-1$
$A[i+1] \leftarrow x$	$n-1$

$$T(n) = (3/2)n^2 + (7/2)n - 4.$$

Figure: Insertion Sort (Feofiloff / CLR)

Insertion Sort - Análise

$$T(n) = (3/2)n^2 + (7/2)n - 4.$$

- A expressão de $T(n)$ é da forma $an^2 + bn + c$.
- O coeficiente $3/2$ de n^2 não é importante, pois não depende do algoritmo, mas de nossa hipótese *1 unidade de tempo por linha*.
- Já o n^2 é fundamental, pois caracteriza o algoritmo em si e não depende nem do computador nem dos detalhes de implementação do algoritmo.
- Dizemos que o algoritmo é quadrático.

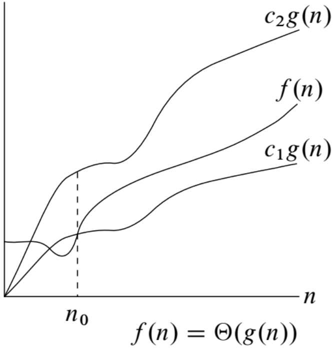
Crescimento de Funções (CLR)

- Quando olhamos para entradas com tamanhos suficientemente grandes para tornar apenas a ordem de crescimento do tempo de execução relevante, estamos estudando a eficiência **assintótica** dos algoritmos.
- As notações utilizadas para descrever o tempo de execução assintótico de um algoritmo são definidas em termos das funções cujos domínios são o conjunto dos **números naturais** $\mathbb{N} = \{0, 1, 2, \dots\}$.
- Em geral usaremos a notação assintótica para caracterizar o **tempo** de execução dos algoritmos. A notação assintótica, no entanto, pode ser aplicada a funções que caracterizam outros aspectos dos algoritmos, como por exemplo, o **espaço** utilizado.

Notação Θ

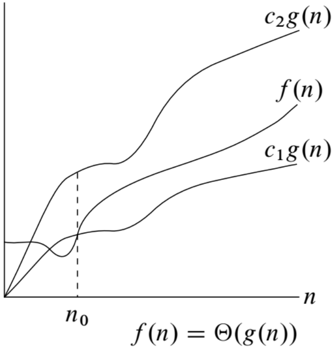
Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o conjunto de funções

$$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1,\ c_2 \text{ e } n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}.$$



Notação Θ

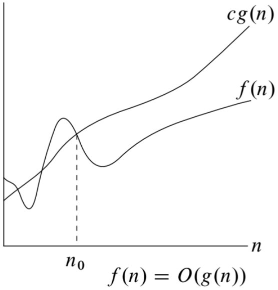
Uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existem constantes positivas c_1 e c_2 tal que ela possa ser sanduichada entre $c_1 g(n)$ e $c_2 g(n)$ para n suficientemente grande. Diz-se que $g(n)$ é um limite assintoticamente justo para $f(n)$.



Notação O

Para uma dada função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}.$$

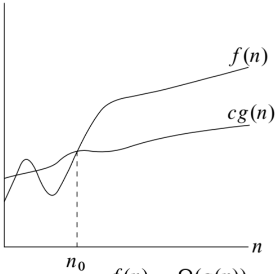


Usamos a notação O (lê-se "ó grande") como um limite superior para uma função, dentro de um fator constante.

Notação Ω

Para uma dada função $g(n)$, denotamos por $\Omega(g(n))$ o conjunto de funções

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tal que } 0 \le cg(n) \le f(n) \text{ para todo } n \ge n_0\}.$$



$f(n) = \Omega(g(n))$

Usamos a notação Ω (lê-se "ômega grande") como um limite assintótico inferior para uma função, dentro de um fator constante.

Comparando Funções

$$f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) \text{ implica } f(n) = \Theta(h(n)) \tag{1}$$

$$f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) \text{ implica } f(n) = O(h(n)) \tag{2}$$

$$f(n) = \Omega(g(n)) \text{ e } g(n) = \Omega(h(n)) \text{ implica } f(n) = \Omega(h(n)) \tag{3}$$

$$f(n) = \Theta(f(n)) \text{ e } f(n) = O(f(n)) \text{ e } f(n) = \Omega(f(n)) \tag{4}$$

$$f(n) = \Theta(g(n)) \text{ se e somente se } g(n) = \Theta(f(n)) \tag{5}$$

$$f(n) = O(g(n)) \text{ se e somente se } g(n) = \Omega(f(n)) \tag{6}$$

Exemplos (Prof. Moacir Ponte)

1. $2n + 10$ é $O(n)$?
2. n^2 é $O(n)$?
3. $3n^3 + 20n^2 + 5$ é $O(n^3)$?
4. $3 \log n + 5$ é $O(\log n)$?
5. 2^{n+2} é $O(2^n)$?

Exemplo 1: $2n + 10$ é $O(n)$?

Podemos realizar uma manipulação para encontrar c e n_0:

$$2n + 10 \leq c.n$$

$$c.n - 2n \geq 10$$

$$(c - 2)n \geq 10$$

$$n \geq \frac{10}{c - 2}$$

A afirmação é válida para $c = 3$ e $n_0 = 10$.

Exemplo 2: n^2 é $O(n)$?

É preciso encontrar c que seja maior ou igual a n para todo valor de n_0:

$$n^2 \leq c.n$$

$$n \leq c$$

É impossível, pois c deve ser constante.

Exemplo 3: $3n^3 + 20n^2 + 5$ é $O(n^3)$?

É preciso encontrar $c > 0$ e $n_0 \geq 1$, tais que $3n^3 + 20n^2 + 5 \leq c.n^3$, para $n \geq n_0$. Como

$$3n^3 + 20n^2 + 5 \leq (3 + 20 + 5).n^3$$

podemos tomar $c = 28$ e qualquer $n_0 > 1$

Exemplo 4: $3 \log n + 5$ é $O(\log n)$?

É preciso encontrar $c > 0$ e $n_0 \geq 1$, tais que $3 \log n + 5 \leq c. \log n$, para $n \geq n_0$. Note que

$$3 \log n + 5 \leq (3+5). \log n \text{ se } n > 1 \; (\log 1 = 0)$$

basta tomar, por exemplo, $c = 8$ e qualquer $n_0 > 2$

Exemplo 5: 2^{n+2} é $O(2^n)$?

É preciso $c > 0$ e $n_0 \geq 1$, tais que $2^{n+2} \leq c.2^n$, para todo $n \geq n_0$. Note que

$$2^{n+2} = 2^2.2^n = 4.2^n$$

Assim, basta tomar, por exemplo, $c = 4$ e qualquer n_0

Paradigma Divisão e Conquista

Muitos algoritmos importantes são estruturalmente recursivos. Tais algoritmos seguem tipicamente o paradigma da divisão e conquista (CLR):

- quebram o problema original em pedaços menores,
- resolvem os subproblemas recursivamente, e
- combinam estas soluções para criar uma solução para o problema original.

Revisão Paradigma Divisão e Conquista

O paradigma da divisão e conquista envolve 3 passos a cada nível recursivo (CLR):

- **Dividir** o problema em um número de subproblemas que são instâncias menores do mesmo problema.
- **Conquistar** os subproblemas resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem suficientemente pequenos, no entanto, resolva-os de modo direto.
- **Combinar** as soluções dos subproblemas na solução para o problema original.

Mergesort

O algoritmo **mergesort** é um ótimo exemplo para o paradigma da divisão e conquista. Intuitivamente opera como segue:

- **Dividir:** Divida a sequência de n elementos em 2 subsequências de $n/2$ elementos cada.
- **Conquistar:** Ordene as 2 subsequências recursivamente usando o mergesort.
- **Combinar:** Intercale (merge) as 2 subsequências para produzir a resposta ordenada.

A recursão para quando a subsequência a ser ordenada tem tamanho 1. Neste caso não há trabalho a ser realizado, dado que uma sequência de tamanho 1 já está ordenada.

Mergesort

```
void mergesort(Item a[], int l, int r)
  { int m = (r+l)/2;
    if (r <= l) return;
    mergesort(a, l, m);
    mergesort(a, m+1, r);
    merge(a, l, m, r);
  }
```

Figure: Mergesort (Sedgewick)

Análise do Mergesort

- Vamos supor, por simplicidade, um vetor $A[n]$, onde $n = 2^i$.
- Quanto tempo leva o algoritmo mergesort para executar sobre A ?

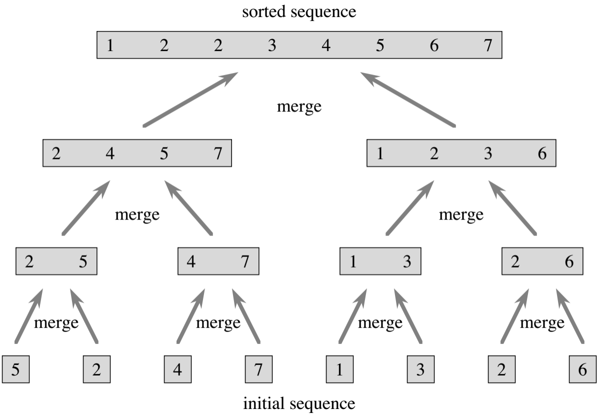


Figure: Simulação do Mergesort (CLR)

Análise do Mergesort

- **Dividir:** O passo da divisão apenas calcula o meio do subvetor, o que leva tempo constante, ou seja, tempo $\Theta(1)$.
- **Conquistar:** Recursivamente resolvemos 2 subproblemas, cada um com tamanho $n/2$, o que contribui $2T(n/2)$ para o tempo de execução.
- **Combinar:** Já observamos que a função Merge para um vetor de tamanho n leva tempo $\Theta(n)$.

```
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)
```

Figure: Mergesort (CLR)

Recorrência do Mergesort

A recorrência (ou fórmula aberta) do Mergesort pode ser obtida a partir do programa abaixo.

```
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)
```

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1. \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{se } n > 1. \end{cases} \tag{7}$$

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1. \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases} \tag{8}$$

Recorrência do Mergesort

A fórmula aberta abaixo é de pouca valia. O que realmente queremos é encontrar a fórmula fechada para esta recorrência. Em outras palavras, precisamos calcular a recorrência.

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

Como calcular a recorrência acima?