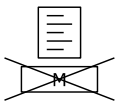
Construção e Análise de Algoritmos

discussão 06: A matemática da notação assintótica

1 Introdução

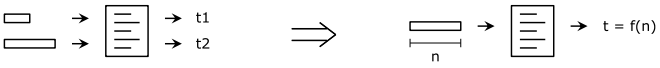
Para obter uma teoria elegante para a análise da eficiência do algoritmo, é preciso adotar uma série de medidas que eliminam ou isolam a influência de fatores que não dizem respeito à eficiência intrínseca de um algoritmo. A seguir, nós vamos discutir brevemente alguma dessas medidas.

- **número de passos da computação**



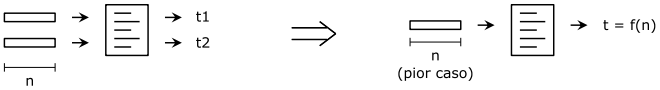
Não faz sentido imaginar que um algoritmo fica mais eficiente só porque ele é executado em uma máquina mais rápida. Para eliminar esse tipo de influência da máquina sobre a análise do algoritmo, nós estimamos a eficiência do algoritmo em termos de número de passos da computação, que ele realiza, ao invés do tempo de execução.

- **tamanho da entrada**



Em geral, entradas maiores levam a uma computação mais longa. Para isolar esse elemento na análise da eficiência de um algoritmo, nós expressamos o número de passos da computação como uma função da entrada, medida em número de bits.

- **análise do pior caso**



Não é incomum encontrar algoritmos que realizam computações de tamanho diferentes para entradas do mesmo tamanho. Para lidar com esse tipo de situação, a estimativa da eficiência do algoritmo em geral é feita em termos do seu desempenho no pior caso. (Em certos casos, no entanto, também pode ser útil fazer uma análise de caso médio.)

- **ignorar constantes aditivas e multiplicativas**

A primeira medida que nós mencionamos ainda não é o suficiente para eliminar toda a influência na análise da eficiência de um algoritmo. Máquinas diferentes permitem definir uma computação em termos de operações primitivas diferentes, e isso certamente tem impacto no comprimento da computação que um algoritmo realiza. Por outro lado, tipicamente, qualquer operação primitiva de uma máquina pode ser simulada ou implementada por meio de um número constante de operações primitivas de uma outra máquina. Isso significa que a influência da máquina corresponde no máximo a um fator aditivo ou multiplicativo na estimativa do comprimento da computação. Assim, ao ignorar as constantes aditivas e

multiplicativas da nossa estimativa, nós eliminamos mais essa influência da máquina sobre a análise.

2 Notação assintótica

A seguir, nós vamos apresentar uma pequena teoria que oferece as ferramentas matemáticas necessárias para realizar a análise de algoritmos de maneira formal.

O primeiro passo consiste em descrever o tempo de execução de um algoritmo por meio de funções da forma

$$f: \mathbb{N} \to \mathbb{N}$$

A ideia aqui é que, para cada $n \in \mathbb{N}$, o valor $f(n)$ indica o tempo de execução do algoritmo para entradas de tamanho n, no pior caso. (Note que esse passo captura as 3 primeiras observações acima.)

Por exemplo, abaixo nós temos algumas funções típicas que descrevem o tempo de execução de algoritmos:

$$f(n) = 10n \qquad f(n) = n^2 \qquad f(n) = \lfloor n \log n \rfloor$$

A seguir, para capturar a última observação, nós introduzimos uma relação de equivalência entre funções, por meio da seguinte definição:

Definição 1: (Notação Θ)

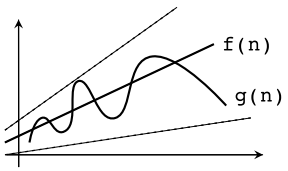
Dada uma função $f : \mathbb{N} \to \mathbb{N}$, nós denotamos por $\Theta(f)$ o conjunto das funções $g : \mathbb{N} \to \mathbb{N}$ que satisfazem a condição

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n), \qquad \forall n \geq n_0$$

para um par de constantes $c_1, c_2 > 0$, e para algum $n_0 > 0$.

Por exemplo, $10n^2 \in \Theta(n^2)$.

Intuitivamente, nós temos que $g \in \Theta(f)$ quando a função g tem aproximadamente a mesma *taxa de crescimento* que a função f (veja a figura abaixo).



Lema 1: A notação Θ define uma relação de equivalência entre funções, isto é

a) $f \in \Theta(f)$

b) $f \in \Theta(g) \Rightarrow g \in \Theta(f)$

c) $f \in \Theta(g)$ e $g \in Theta(h) \Rightarrow f \in \Theta(h)$

Prova: (propriedade (b))

Assumindo que $f \in \Theta(g)$, nós sabemos que existem constantes $c_1, c_2 > 0$ tais que

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

Da primeira desigualdade, segue que

$$g(n) \;\leq\; \frac{1}{c_1} \cdot f(n)$$

E, da segunda desigualdade, segue que

$$\frac{1}{c_2} \cdot f(n) \;\leq\; g(n)$$

Daí,

$$\frac{1}{c_2} \cdot f(n) \;\leq\; g(n) \;\leq\; \frac{1}{c_1} \cdot f(n)$$

e portanto $g \in \Theta(f)$.

□

Além disso, assumindo que $\mathbb{N} = \{1, 2, 3, \ldots\}$, não é difícil provar que

Lema 2: Se $a, b > 0$ são constantes, então $af + b \in \Theta(f)$.

Esses resultados mostram que a notação Θ oferece uma ferramenta matemática adequada para comparar a eficiência de algoritmos. Mais especificamente, considere dois algoritmos A e B que executam em tempo (i.e., número de passos) $f(n)$ e $g(n)$, respectivamente. Nós vamos dizer que A e B possuem a mesma eficiência computacional quando $f \in \Theta(g)$.

Notação O

Na maioria dos casos, não é realmente necessário obter uma estimativa precisa do tempo de execução de um algoritmo, mas um bom limite superior já é suficiente. Nesses casos, nós podemos utilizar a seguinte variante simplificada da notação Θ:

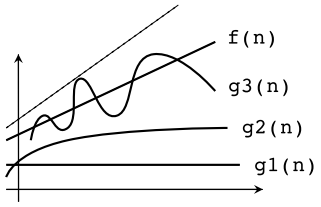
Definição 2: (Notação O)

Dada uma função $g(n)$, nós denotamos por $O(g)$ o conjunto das funções $f(n)$ que satisfazem a condição

$$f(n) \;\leq\; c \cdot g(n), \qquad \forall n \geq n_0$$

para alguma constante $c > 0$, e algum $n_0 \in \mathbb{N}$.

Intuitivamente, se $g \in O(f)$ então a função g cresce no máximo tão rápido quanta a função f (aproximadamente). Na figura abaixo, as funções g_1, g_2 e g_3 estão todas em $O(f)$.



A seguir, nós temos um resultado importante que relaciona as notações Θ e O:

Lema 3: Se $f \in \Theta(g)$ e $h = O(f)$, então $f + h \in \Theta(g)$.

Prova: Como $f \in \Theta(g)$, existem constantes $c_1, c_2 > 0$ tais que

$$c_1 \cdot g(n) \;\leq\; f(n) \;\leq\; c_2 \cdot g(n) \tag{1}$$

Analogamente, como $h \in O(f)$, existe constante $c_3 > 0$ tal que

$$h(n) \ \leq \ c_3 \cdot f(n) \tag{2}$$

A seguir, como $h(n) \geq 0$ para todo n, segue da primeira desigualdade em (**??**) que

$$c_1 \cdot g(n) \ \leq \ f(n) \ + \ h(n) \tag{3}$$

Já a segunda desigualdade em (**??**) nos dá que

$$(c_3 + 1) \cdot f(n) \ \leq \ c_2(c_3 + 1)g(n) \tag{4}$$

O resultado desejado segue diretamente de (**??**) e (**??**).

□

A consequência imediata desse teorema é que, ao analisar a eficiência de um algoritmo, nós podemos desconsiderar não apenas as constantes aditivas e multiplicativas, mas também todos os termos de ordem inferior.

Por exemplo, se um algoritmo realiza $3n^2 + 10n + 50$ passos quando recebe uma entrada de tamanho n, nós dizemos simplesmente que ele executa em "tempo" $\Theta(n^2)$.

Esse fato é muito conveniente pois ele permite agrupas os algoritmos em "classes" associadas a funções simples como $n, n^2, n \log n$, etc.

3 Análise de complexidade de algoritmos

A seguir, nós vamos ver alguns exemplos concretos de análise de eficiência ou complexidade de algoritmos.

a) Busca linear em um vetor não ordenado

```
Busca (x, V[1...n]){
    Para i := 1 Até n Faça {
        Se (x = V[i])
            Retorna (i)
    }
    Retorna(0);
}
```

O número de passos desse algoritmo depende da posição em que x é encontrado (se é que isso acontece). No pior caso, onde x não está no vetor V, o algoritmo realiza a comparação "$x = V[i]$" n vezes. Detalhes como a inicialização e incremento do índice i, e o teste a cada iteração do laço, podem ser ignorados pois eles afetam o tempo de execução apenas em termos de constantes aditivas e multiplicativas. Logo, a complexidade do algoritmo é $\Theta(n)$.

b) Busca binária em um vetor ordenado

```
Busca-binaria (x, V[1...n]){
    i ← 1; j ← n
    Enquanto (i ≤ j) {
        k ← ⌊(i + j)/2⌋
        Se (x = V[k]) Retorna (k)
        Se (x < V[k]) j ← k − 1
```

```
            Se (x > V[k]) i ← k + 1
        }
        Retorna(0)
    }
```

Para analisar a complexidade desse algoritmo, nós observamos que no pior caso, em que x não se encontra no vetor V, o laço é executado no máximo $\log_2 n$ vezes. Essa observação decorre do fato que o valor de $|i - j|$, que inicialmente é igual a n, é reduzido à metade a cada iteração do laço. Portanto, a complexidade do algoritmo é $\Theta(\log n)$.

(Note que a análise não é cuidadosa (e pode até cometer pequenos erros) em detalhes que afetam apenas constantes aditivas e multiplicativas. Isso quase nunca é um problema, pois elas não alteram a complexidade do algoritmo.)

Exercícios

1. Apresente uma prova para os Lemas 1 e 2 da Seção 2.

2. Prove que $f \in O(g)$ e $g \in O(h)$ então $f \in O(h)$.

3. Prove que $f \in O(g)$ e $g \in O(f)$ então $f \in \Theta(h)$.

4. Indique se cada uma das proposições abaixo é verdadeira ou falsa, e justifique a sua resposta.

   a) $5n + 8n^2 + 100n^3 \in O(n^4)$

   b) $5n + 8n^2 + 100n^3 \in O(nlog(n))$

5. Analise a complexidade dos algoritmos abaixo.

   a) **Algoritmo de ordenação da bolha**

```
Bubble-sort (V[1..n]) {
    Para i := n Até 2 {
        Para j := 1 Até i − 1 {
            Se (V[j] > V[j + 1]) {
                aux ← V[j + 1]
                V[j + 1] ← V[j]
                V[j + 1] ← aux
}   }   }   }
```

   b) **Algoritmo de ordenação por inserção**

```
Insertion-sort (V[1..n]) {
    Para i := 1 Até n − 1 {
        aux ← i
        Para j := i + 1 Até n
            Se (V[aux] > V[j])
                aux ← j
        aux2 ← V[i]
        V[i] ← V[aux]
        V[aux] ← aux2
}   }
```