Notação Assintótica

Danielle B. Colturato

Introdução

- O que é analisar um algoritmo?
  - Predizer a quantidade de recursos utilizados (memória, tempo de execução, número de processadores, acessos a disco, etc.);
  - Na maioria dos casos estaremos interessados em avaliar o tempo de execução (computação) gasto pelo algoritmo;
  - Contaremos o número de operações efetuadas.

Medidas de Complexidade

- Análise de Algoritmo é a medição da complexidade do algoritmo
  - Quantidade de "trabalho" necessário para sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo e em função do volume de dados a ser processado;
  - A quantidade de trabalho requerido por um algoritmo não pode ser descrito por um número, pois o número de operações básicas efetuadas não é a mesma para qualquer entrada (depende do tamanho da entrada).

Medidas de Complexidade

- Por que estudar Complexidade?
  - Performance
    - Escolher entre vários algoritmos o mais eficiente;
    - Desenvolver novos algoritmos para problemas que já tem solução;
    - Desenvolver algoritmos mais eficientes, devido ao aumento constante do tamanho dos problemas a serem resolvidos.
  - Complexidade Computacional
    - Permite determinar se a implementação de determinado algoritmo é viável.

Medidas de Complexidade

- Um algoritmo possui duas medidas de complexidade:
  - Espacial
    - Quantidade de memória que utiliza durante sua execução;
  - Temporal
    - Número aproximado de instruções que ele executa
  - Ambas as complexidades são em função do tamanho da entrada.

Medidas de Complexidade

- Exemplos de Complexidade Temporal
  - Limite do tamanho do problema através da taxa de crescimento

Algoritmo	Complexidade Temporal	Tamanho máximo do problema		
		1 seg	1 min	1 hora
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Notação O

- Sejam *f(n)* e *g(n)* funções cujo domínio são os *Naturais* e a imagem os *Reais*.
- Uma função *f(n)* é ***O(g(n))*** quando existe uma constante real *c>0* e uma constante inteira $n_0 \geq 1$, tais que $f(n) \leq c.g(n)$ para todo inteiro $n \geq n_0$. Quando isso acontece dizemos que *f(n)* é de ordem *g(n)*, ou seja, é ***O de g(n)***.

Notação O

- Como vimos na aula anterior, através da contagem das instruções primitivas o algoritmo do MaiorElemento, que encontra o maior de um vetor de comprimento *n*, é definida pela função $f(n) = 7n-2$.
- O algoritmo é ***O(n)***, porque se seguirmos a definição $f(n) \leq c.g(n)$, existe uma constante *c*, tal como *7*, e uma constante $n_0 = 1$, que satisfaz a relação, ou seja, $7n-2 \leq 7n$ para todo $n \geq 1$.

Notação O

- Outro exemplo:

```
Algoritmo Somatorio(int n) : int
{
    int soma = 0
    para int i = 1 até n faça{
        para int j = 1 até n faça{
        soma = soma + i + j
        }
    }
    retorna soma
}
```

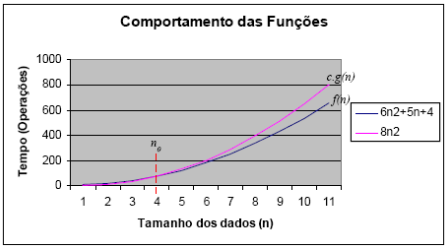
Notação O

- Contagem de passos para determinar a função *f(n)* que representa o algoritmo do exemplo:
  - **5n operações**: As atribuições soma = soma + i + j e a do incremento do j totalizam 5 operações executadas dentro de um *loop* que ocorre *n* vezes.
  - **n+2 operações**: considerando a inicialização do *j* e o teste no *loop* do *j* que ocorre *n+1* vezes.
  - ***6n+2* operações**: totalizando, esta é a quantidade de operações executada pelo *loop* do *j*.
  - ***(6n+2 + 2)n = $6n^2+4n$ operações:*** visto que o *loop i* executa o *loop j* e o incremento do *i n* vezes.
  - **n+2 operações**: considerando a inicialização do *i* e o teste no *loop* do *I* que ocorre *n+1* vezes.
  - ***$6n^2+5n+4$ operações:*** totalizando as operações executadas pelo loop i, a inicialização da variável soma e a operação de retorno.
- Este algoritmo é ***O(n^2)***, porque a relação pode ser satisfeita da seguinte forma: $6n^2+5n+4 \leq 8n^2$ para todo $n \geq 4$.

Notação O

- A notação O caracteriza o tempo de execução e o consumo de memória em função da quantidade de dados (*n*), no entanto de uma forma mais intuitiva e bem menos trabalhosa.
- Permite afirmamos que a função *f* que representa o algoritmo cresce de forma assintótica à uma outra função *g,* que é "maior que f".
- Isso significa que à medida que *n* tende ao infinito as funções têm um comportamento semelhante.
- A constante *c* representa "o quanto g é maior que f" e a constante *n0*, o ponto inicial em as funções terão comportamento parecido.

Notação O



Notação O

- Ao invés de usarmos a definição da notação O diretamente, podemos usar as seguintes regras, considerando que f(n), g(n), d(n) e e(n) são funções cujo domínio são os Naturais e as imagens, os Reais não negativos:

1. Se d(n) é O(f(n)), então a.d(n) é O(f(n)), para uma constante a>0.
2. Se *d(n)* é *O(f(n))* e *e(n)* é *O(g(n))* então *d(n)+e(n)* é *O(f(n)+g(n))*.
3. Se *d(n)* é *O(f(n))* e *e(n)* é *O(g(n))* então *d(n).e(n)* é *O(f(n).g(n))*.
4. Se *d(n)* é *O(f(n))* e *f(n)* é *O(g(n))* então *d(n)* é *O(g(n))*.
5. Se *f(n)* é um polinônimo de grau *d*, ou seja, $a_0 + a_1n + a_2n^2 + ...a_dn^d$, então *f(n)* é $O(n^d)$.
6. n^x é $O(a^n)$ para quais constantes x>0 e a>1.
7. $log(n^x)$ é *O(log(n))* para qualquer constante x>0.
8. $(log(n))^x$ é $O(n^y)$ para quaisquer constantes x>0 e y>0.
9. Se *d(n)* é *c (sempre constante, c>0)*, então *d(n)* é *O(1)*.
10. *O(O(f(n))) = O(f(n))*
11. *O(f(n).g(n)) = f(n). O(g(n))*

Notação O

- *Exemplo:*

f(n) = $5n^3 + 10$ corresponde a *$O(n^3)$*
$5n^3$ é *$O(n^3)$* ***[regra 1]***
10 é O(1) ***[regra 9]***
$O(n^3) + O(1)$ é $O(n^3 + 1)$ ***[regra 2]***
$O(n^3 + 1)$ é $O(n^3)$ ***[regra 5]***

Notação Ω (Ômega)

- A notação ômega é outro tipo de notação assintótica. Dizemos que f(n) é Ω g(n), se existe uma constante real c>0 e uma constante inteira $n_0 \geq 1$, tais que $f(n) \geq c.g(n)$ para $n \geq n_0$.
- Por exemplo, a função *f(n) = 7n-2,* que define o comportamento do algoritmo maior, *é Ω(n),* pois *7n-2 ≥ 6n para n ≥ 2.*

Notação Ω (Ômega)

- Exemplos:
  - *f(n) = 5n.log(n) corresponde a Ω(n),* considerando que *5n.log(n) ≥ 5n para* ***n*** *≥ 2.*
  - *f(n) = 3.log(n) + log(log(n)) corresponde a Ω(log(n)), considerando que:* *3.log(n) + log(log(n)) ≥ 3.log(n) para n ≥ 2.*

Notação Θ(Teta)

- A notação teta pode ser vista como uma junção das duas notações anteriores.
- Dizemos que f(n) é Θ g(n), se existe duas constantes reais c1 e c2 ambas maiores que 0 e uma constante inteira $n_0 \geq 1$, tais que $c_1.g(n) \leq f(n) \leq c_2.g(n)$ para $n \geq n0$..
- Considerando novamente, a função *f(n) = 7n-2 é Θ(n),* pois *6n ≤ 7n-2 ≤ 7n para n ≥ 2.*

Notação Θ(Teta)

- *Exemplos:*
  - *f(n) = 5n.log(n) corresponde a $\Theta(n^2)$,* considerando que ***1/3*** *$n^2 \leq$ 5n.log(n) ≤* ***3*** *n^2 para* ***n*** *≥ 2.*
  - *f(n) = 3.log(n) + log(log(n))* corresponde a *Θ(log(n)),* considerando que ***1.****log(n) ≤ 3.log(n) + log(log(n)) ≤* ***4.****log(n) para* ***n*** *≥ 2.*

Complexidade de Algoritmos

- É comum usarmos os nomes das funções para definir a complexidade de um algoritmo.
- Por exemplo, se o algoritmo é O(n) dizemos que sua complexidade é linear.
- Cada função tem um nome conhecido:
  - O(log(n)) : logarítmica
  - $O(n^2)$: quadrática
  - $O(n^3)$: cúbica
  - $O(n^k)$: polinomial
  - $O(a^n)$: exponencial, para a>1.

Exercícios

- Determine a notação assintótica O para as funções a seguir:
  a) $(n^2-n)/2$
  b) $n + 2\ n^{1/2}$
  c) $n^2 + 3n + 4$
  d) $2(\log(n))^2$

- Prove que:
  a) $5n = O(n \log(n))$
  b) $40n.\log(n) = O(n^2)$
  c) $7n.\log(n) = \Omega(n)$
  d) $12n^2 = \Omega(n.\log(n))$