DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Complexidade Assintótica

Professores: Luiz Chaimowicz e Raquel Prates

Função de complexidade

- Varia com o tamanho de n
  - Para n suficientemente pequeno, qualquer algoritmo custa pouco

- Estudamos o comportamento do custo quando n cresce (valores grandes)

comportamento assintótico das funções de custo

Complexidade Assintótica

- Seja ***T(n)*** uma medida de tempo de execução de um algoritmo para um problema de tamanho ***n***
- *Se* ***T(n) = 2n⁴ + 3n³ + 4n² + 5n + 6****, então*

$$T(n) = n^4\left(2 + \frac{3}{n} + \frac{4}{n^2} + \frac{5}{n^3} + \frac{6}{n^4}\right)$$

E para valores grandes de ***n*** *teremos:*

$$T(n) \cong 2n^4$$

Teóricos não se interessam muito por constantes ☺

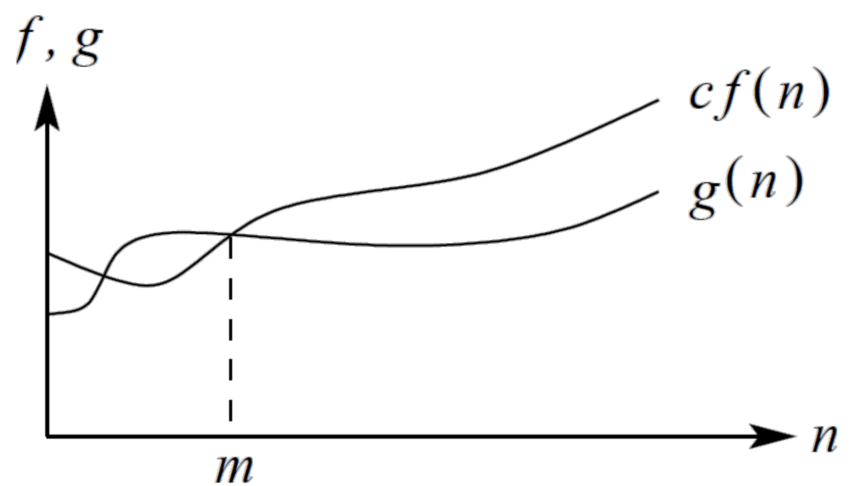
- *Dizemos queT(n) tem ordem de crescimento* ***n^4: O(n^4)***
- *Dizemos que um algoritmo é* ***mais eficiente*** *do que um outro se seu tempo de execução* ***no pior caso*** *tiver* ***ordem de cresimento*** *menor.*

Complexidade Assintótica

- Na análise de algoritmos, na maioria das vezes usa-se o estudo da complexidade assintótica, ou seja, **analisa-se o algoritmo quando o valor de *n* tente a infinito**
- Nesse caso, não é necessário se preocupar com as constantes e termos de menor crescimento
- Usa-se notações especiais para representar a complexidade assintótica

Dominação Assintótica

- **Definição**: Uma função f(n) ***domina assintoticamente*** outra função *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c|f(n)|$.

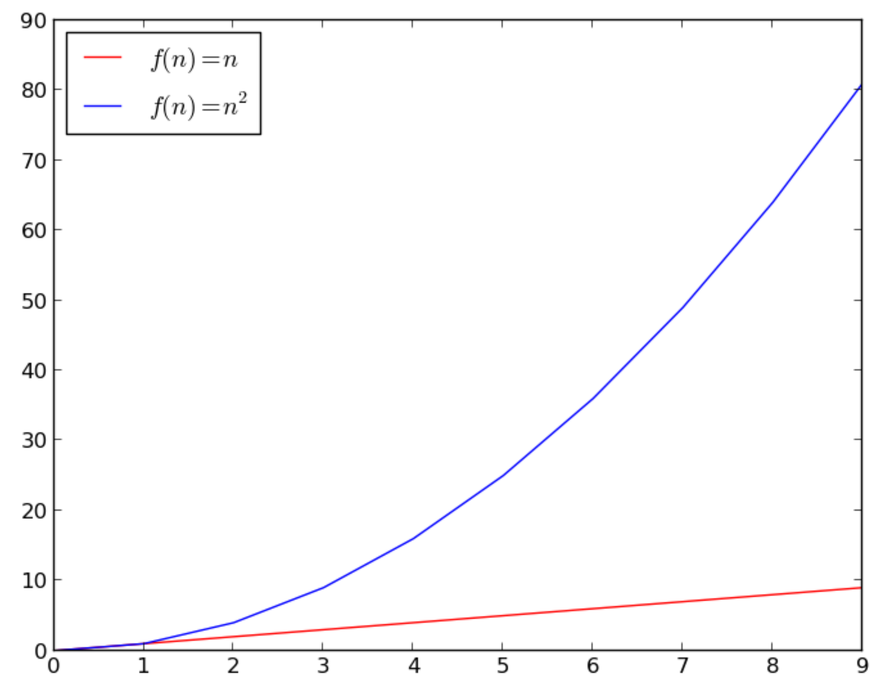


Dominação Assintótica

- Sejam $f(n) = n^2$ e $g(n) = n$.
  - $f(n)$ domina assintoticamente $g(n)$?
  - Se $|g(n)| \leq c|f(n)|$ para $n \geq m$ e $c > 0$

$c = 1$ e $m = 0$

$n \leq 1 \cdot n^2$ p/ $n \geq 0$



Dominação Assintótica

- Sejam $f(n) = n^2$ e $g(n) = (n+1)^2$
- $f(n)$ domina assintoticamente $g(n)$?
  - $g(n) \leq cf(n)$, $c = 4$ e $n \geq 1$

$$(n+1)^2 \leq 4n^2$$

$$n^2 + 2n + 1 \leq 4n^2$$

$$n + 2 + 1/n \leq 4n$$

Dominação Assintótica

- g(n) domina assintoticamente f(n)?
  - c = 1 e n $\geq$ 1

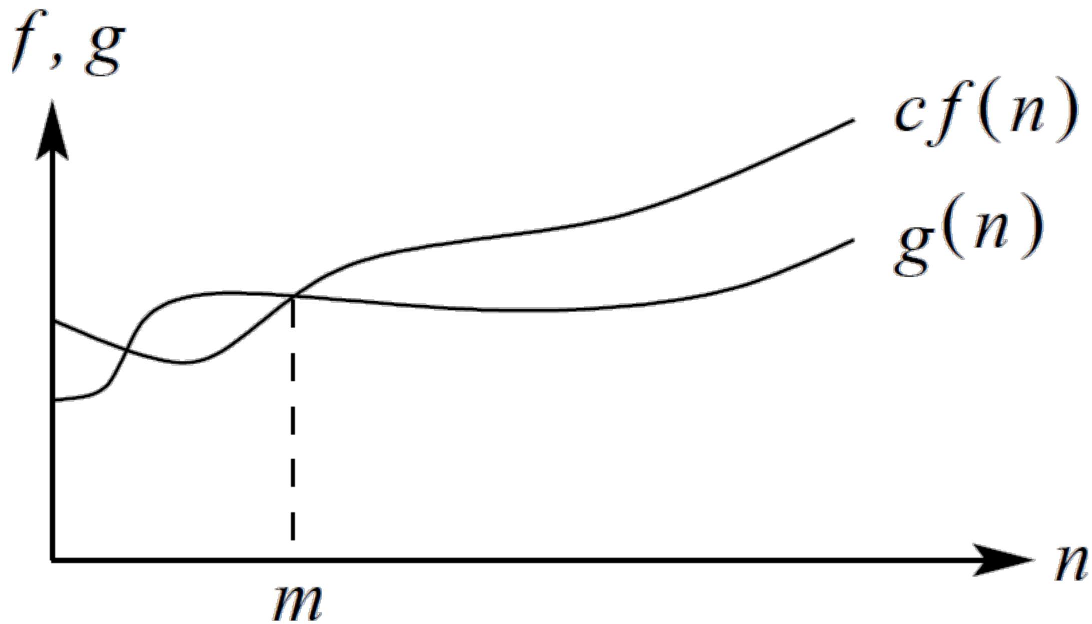
    $n^2 \leq (n+1)^2$

    $n^2 \leq n^2 + 2n + 1$

    $1 \leq 1 + 2/n + 1/n^2$
  - f(n) e g(n) dominam assintoticamente uma a outra

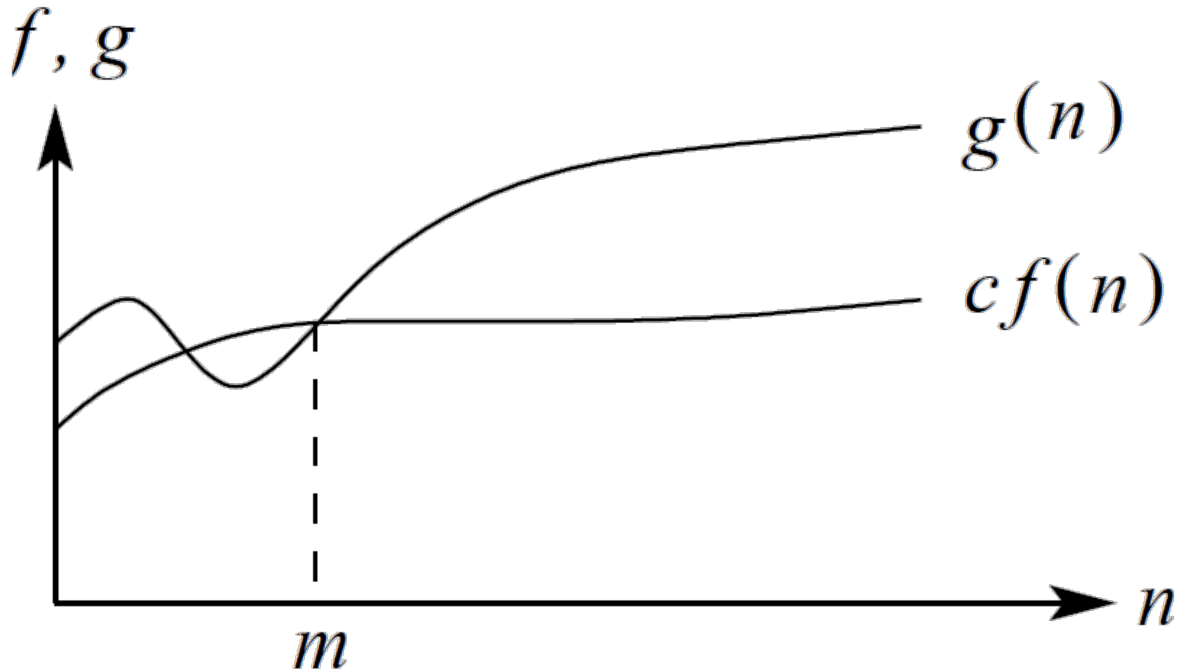
Notação Assintótica

- Notação **O**
  - especifica um **limite superior** para g(n)
  - **g(n) = O(f(n))**



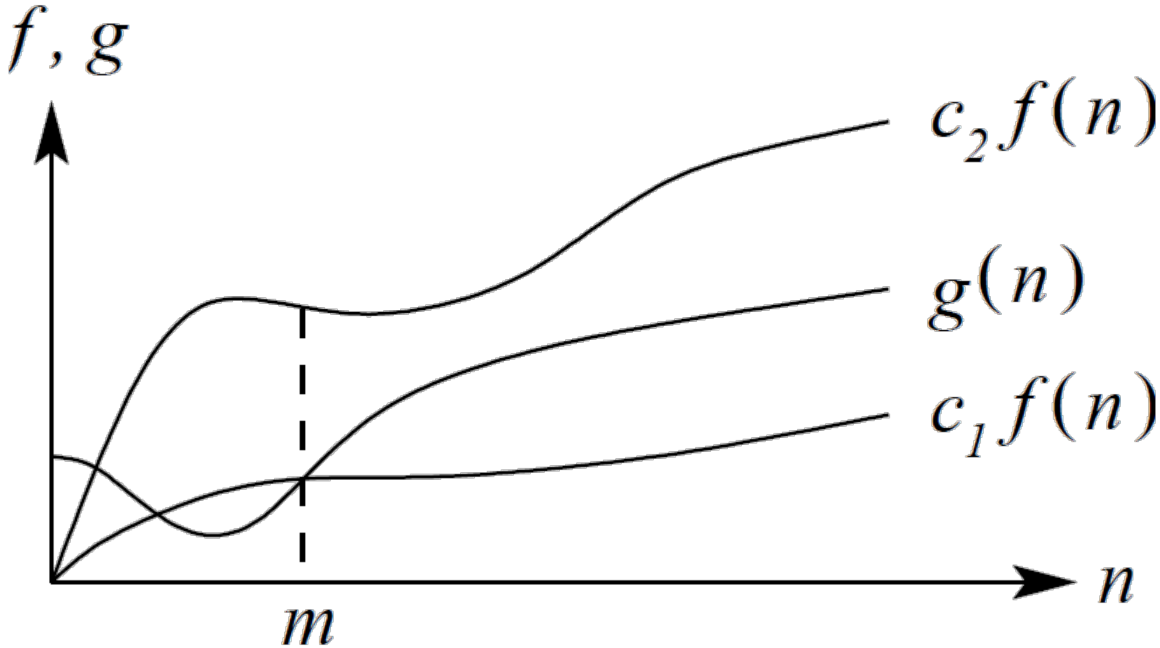
Notação Assintótica

- Notação **Ω**
  - especifica um **limite inferior** para g(n)
  - **g(n) = Ω (f(n))**



Notação Assintótica

- **Notação Θ**
  - especifica um **limite firme** para g(n)
  - **g(n) = Θ(f(n))**



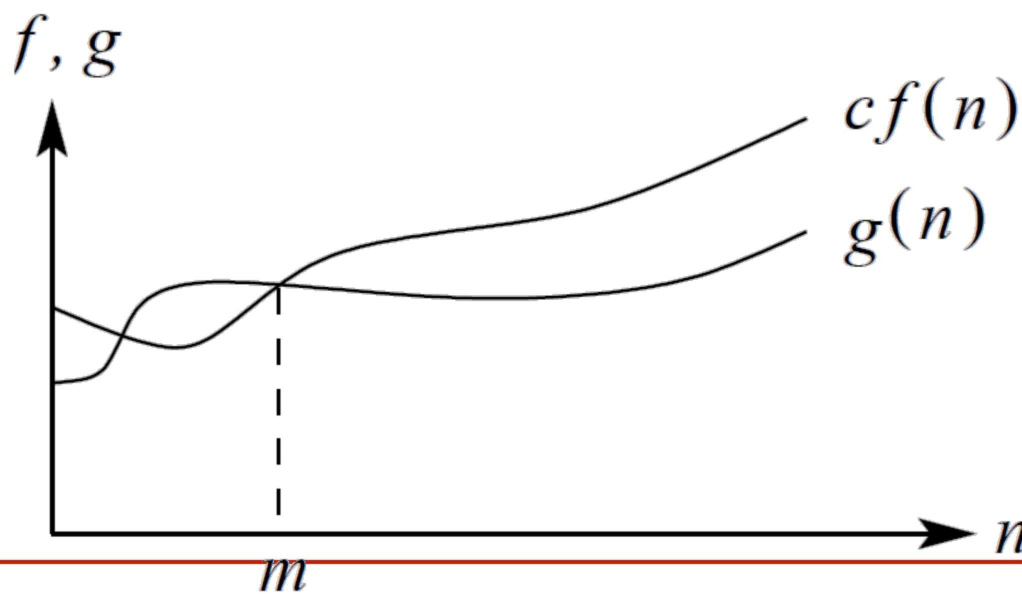
Notação O

- Escrevemos **g(**n**) = O(f(**n**))** se
  - **f(**n**)** domina assintoticamente **g(**n**)**.
  - Cuidado com o abuso de notação: g=O(f) e h=O(f) não implica g=h.
- Lê-se **g(**n**)** é da ordem no máximo **f(**n**)**.

- Exemplo: quando dizemos que o tempo de execução **f(**n**)** de um programa é **O(n²)**, significa que existem constantes **c** e **m** tais que, para valores de $n \geq m$, $f(n) \leq cn^2$.

Notação O

- **Definição**: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que
  - $g(n) \leq cf(n)$, para todo $n \geq m$.



Notação O

- **Exemplo 1**: Seja a função $g(n) = (n + 1)^2$.

  Ela é $O(n^2)$?

> *g*(*n*) é *O*(*f*(*n*)) se *g*(*n*) ≤ *cf*(*n*), para todo *n* ≥ *m*, *onde m e c são positivas*

$$(n+1)^2 \leq cn^2$$

$$n^2 + 2n + 1 \leq cn^2$$

$$1 + 2/n + 1/n^2 \leq c$$

$$m = 1 \text{ e } c = 4$$

Notação O

- **Exemplo 2**: $g(n) = n$ e $f(n) = n^2$

  $g(n) = O(n^2)$?
  - Sim, pois para $m \geq 0$ e $c = 1$, $n \leq n^2$.

- $f(n) = O(n)$?
  - Suponha que existam constantes *c* e *m* tais que para todo $n \geq m$, $n^2 \leq cn$, $n \leq c$
  - Logo $c \geq n$ para qualquer $n \geq m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

Notação O

- **Exemplo 3**: $g(n) = 3n^3 + 2n^2 + n$, $g(n) = O(n^3)$?
  - Seja m = 1 e c = 6,
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$

    $3 + 2/n + 1/n^2 \leq 6$
- A função $g(n) = 3n^3 + 2n^2 + n$ também é $O(n^4)$?
  - Sim, mas essa afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.

    $3n^3 + 2n^2 + n \leq cn^4$

    $3 + 2/n + 1/n^2 \leq cn$, $c=6$, $m=1$

Operações com a Notação O

- Qual é a função de complexidade da função func (considere atribuição a operação relevante)?

```
void func(int *v, int n) {
    int i, j, sum = 0;

     for (i = 0; i < n; i++)
         sum += v[i];

     for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
             sum += v[j];
}
```

Operações com a Notação O

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \qquad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

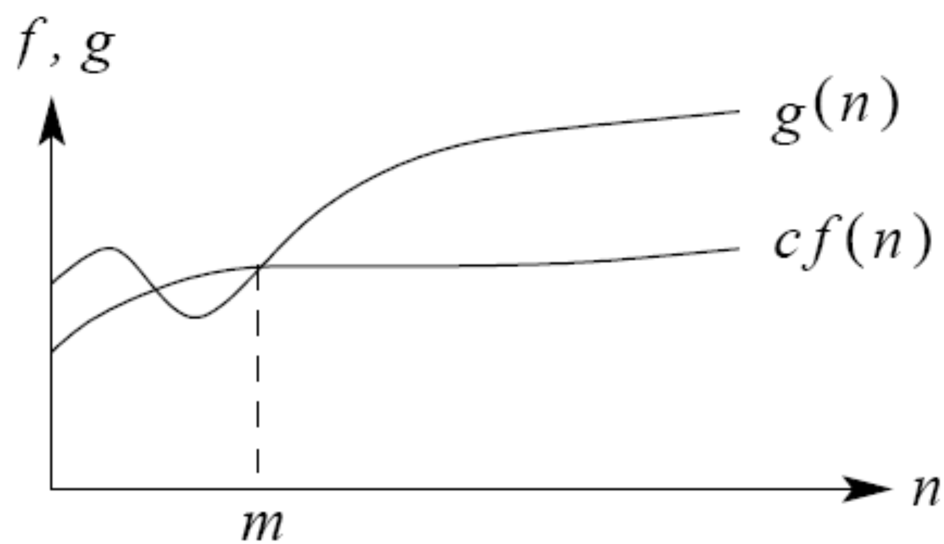
Operações com a Notação O

- **Exemplo 1**: regra da soma $O(f(n)) + O(g(n))$.
  - Suponha três trechos cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n^3)$.
  - O tempo de execução dos dois primeiros trechos é $O(\max(n, n^2))$, que é $O(n^2)$.
  - O tempo de execução de todos os três trechos é então: $O(\max(n^2, n^3))$, que é $O(n^3)$.

DCC

Notação Ω

- Especifica um **limite inferior** para g(n).
- **Definição**: Uma função g(n) é Ω(f(n)) se existirem **duas constantes c e m** tais que
  - $g(n) \geq cf(n)$, para todo $n \geq m$.



Notação Ω

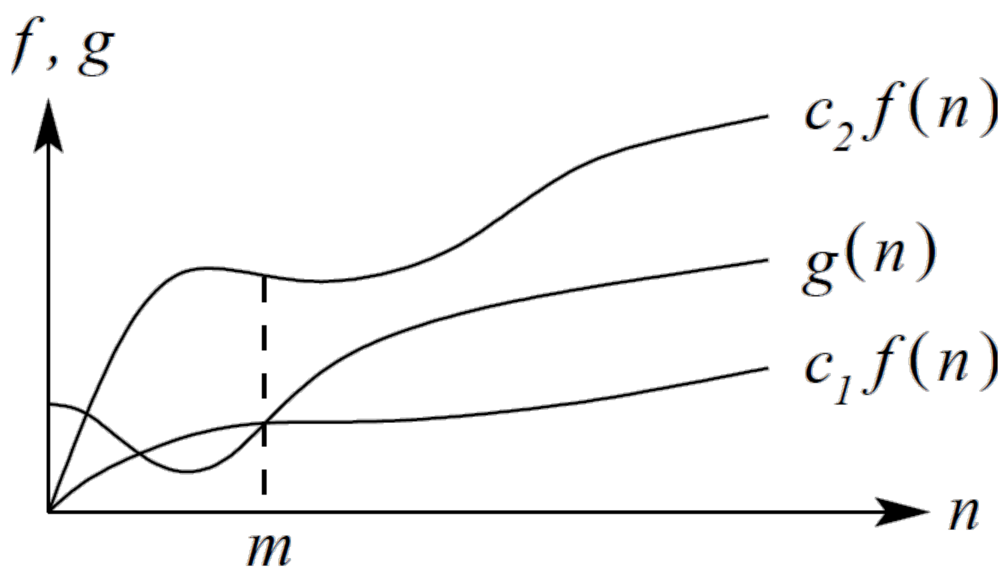
- **Exemplo**: Mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$

> $g(n)$ é $\Omega(f(n))$ se $g(n) \geq cf(n)$,
> para todo $n \geq m$, onde m e c são positivas

- Faça $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $m = 1$.
- $3 + 2/n \geq 1$ para $n \geq 0$

Notação Θ

- **Definição**: Uma função *g(n)* é Θ*(f(n))* se existirem constantes positivas c_1, c_2 e m tais que
  - $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$



Notação Θ

- **Exemplo**:
  Mostrar que $g(n) = n^2/3 - 2n$ é $\Theta(n^2)$

> $g(n)$ é $\Theta(f(n))$, se $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$, *onde* m, c_1 *e* c_2 *são positivas*

- $0 \leq c_1 n^2 \leq n^2/3 - 2n \leq c_2 n^2 \quad (\div n^2)$
  $0 \leq c_1 \leq 1/3 - 2/n \leq c_2$
- Faça $c_1 = 1/21$ *e* $c_2 = 1/3$, e $m = 7$ então

  $0 \leq 1/21 \leq 1/3 - 2/n \leq 1/3$ *para* $n >= 7$

Notação Θ

- Para todo $n \geq m$, a função $g(n)$ é igual a $f(n)$ a menos de uma constante.
- Para o caso da função $g(n)$ ser $\Theta(f(n))$, $f(n)$ é um **limite assintótico firme**.

- **Teorema:**

  Para quaisquer duas funções $f(n)$ e $g(n)$, teremos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Notação *O x* Notação Θ

- Apesar de muito usada na literatura como limite firme, a notação *O* é mais fraca que a notação Θ
  - **f(n) = Θ(g(n)) implica em f(n) = *O*(g(n))**, mas não o contrário
  - $\Theta(g(n)) \subset O(g(n))$
- Como é um limite superior, é muito usado para o pior caso.
  - Ex.
  - O inserção é $O(n^2)$, para qualquer entrada.

Notação *O* x Notação Θ

- Apesar de muito usada na literatura como limite firme, a notação *O* é mais fraca que a notação Θ
  - **f(n) = Θ(g(n)) implica em f(n) = *O*(g(n))**, mas não o contrário
  - $\Theta(g(n)) \subset O(g(n))$
- Como a notação O é um limite superior, é muito usado para o pior caso.
  - Exemplo: Método de ordenação por inserção é $O(n^2)$, para qualquer entrada.

Classes de Comportamento Assintótico

- Em geral, é interessante agrupar os algoritmos / problemas em **Classes de Comportamento Assintótico**, que vão determinar a complexidade inerente do algoritmo
- Como explicado, o comportamento assintótico é medido quando o tamanho da entrada (n) tende a infinito, com isso, as constantes são ignoradas e apenas o componente mais significativo da função de complexidade é considerado

Classes de Comportamento Assintótico

- **Quando dois algoritmos fazem parte da mesma classe de comportamento assintótico, eles são ditos equivalentes *assintoticamente*.**
  - Nesse caso, para escolher um deles deve-se analisar mais cuidadosamente a função de complexidade ou o seu desempenho em sistemas reais

Principais Classes de Problemas

- **f(n) = O(1):** Complexidade constante.
  - Uso do algoritmo independe de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.

```
void algoritmo1(int *v, int n){
int i, j, aux;

  for(i = 0; i < 10; i++){
     for(j = 0; j < 9; j++){
         if(v[j]>v[j+1]) {
          aux=v[j]; v[j]=v[j+1]; v[j+1]=aux;
         }
      }
   }
}
```

Principais Classes de Problemas

- **f(n) = O(n):** Complexidade linear.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
  - Cada vez que *n* dobra de tamanho, o tempo de execução dobra.

```
int algoritmo2(int *v, int n, int k){
int i;

    for(i = 0; i < n; i++){
        if(v[i] == k)
            return i;
    }
    return -1;
}
```

Principais Classes de Problemas

- **f(n) = O(log n):** Complexidade logarítmica.
  - Tempo menor do que o tamanho da entrada
  - A cada passo, uma fração const. de dados de entrada é descartada
  - Quando
    - n é mil, $\log_2 n \sim 10$
    - n é 1 milhão, $\log_2 n \sim 20$.
  - Exemplo:
    - Busca binária em um vetor previamente ordenado

Principais Classes de Problemas

- **f(n) = O(n log n)**: complexidade quase linear
  - Típico em algoritmos que quebram um problema em dois menores, resolvem cada um deles independentemente e ajuntando as soluções depois.
  - Exemplos:
    - Algoritmos de ordenação (mergesort, heapsort)

Principais Classes de Problemas

- **$f(n) = O(n^2)$:** Complexidade quadrática
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.

```
void algoritmo(int *v, int n){
int i, j, aux;

   for(i = 0; i < n; i++){
      for(j = 0; j < n-1; j++){
          if(v[j]>v[j+1]) {
             aux=v[j]; v[j]=v[j+1]; v[j+1]=aux;
          }
      }
   }
}
```

Principais Classes de Problemas

- **$f(n) = O(n^3)$:** Complexidade cúbica
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão.
  - Exemplo: multiplicação de matrizes (algoritmo simples)
- **$f(n)=O(n^k)$, p/ const. $k>0$:** Complexidade polinomial
  - Tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
  - $f(2n) = c\ f(n)$, $c=2^k$

Principais Classes de Problemas

- **$f(n) = O(2^n)$:** Complexidade exponencial
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
  - $f(2n) = f^{2n} = f(n)f(n)$

Principais Classes de Problemas

- **$f(n) = O(n!)$**
  - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa **força bruta** na solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40 \rightarrow$ um número com 48 dígitos!

Principais Classes de Problemas

- É possível piorar?
  - Bogosort O(n.n!)

```
void algoritmo(int *v, int n){
  while not inOrder(v){
      shuffle(v);
   }
}
```

Algoritmos Exponenciais x Polinomiais

- **Algoritmo exponencial:**
  - Tempo de execução tem função de complexidade $O(c^n)$; $c > 1$, para ALGUMA entrada de tamanho n.
- **Algoritmo polinomial:**
  - Tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio, para TODAS as entradas de tamanho n.
  - $T(n)=O(n^k)$, para uma constante $k>0$.

Algoritmos Exponenciais x Polinomiais

- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.

- Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais.

Algoritmos Exponenciais x Polinomiais

- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.

- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema.

Algoritmos Exponenciais x Polinomiais

- Um problema é considerado:
  - **intratável**: se não existe um algoritmo polinomial para resolvê-lo (para elo menos algumas entradas).
  - **bem resolvido**: quando existe um algoritmo polinomial para resolvê-lo (para todas as entradas).

Algoritmos Exponenciais x Polinomiais

- É importante lembrar que a distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- **Exemplo**: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.

Comparação de Funções de Complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Comparação de Funções de Complexidade

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Influência do aumento de velocidade dos computadores no tamanho do problema

Referências

- Ziviani, N., **Projeto de Algoritmos com Implementações em Pascal e C**, 3ª Edição, Cengage Learning, 2011.
  - Capítulo 1 - Seção 1.3.1

Referências

- Ziviani, N., **Projeto de Algoritmos com Implementações em Pascal e C**, 3ª Edição, Cengage Learning, 2011.
  - Capítulo 1 - Seção 1.3.1
- Cormen , T., Leiserson, C, Rivest R., Stein, C. Algoritmos – Teoria e Prática, 3a. Edição, Elsevier, 2012.
  - Capítulo 3 - Seção 3.1