Medida do Tempo de Execução de um Programa

Livro "Projeto de Algoritmos" – Nívio Ziviani

Capítulo 1 – Seção 1.3.1

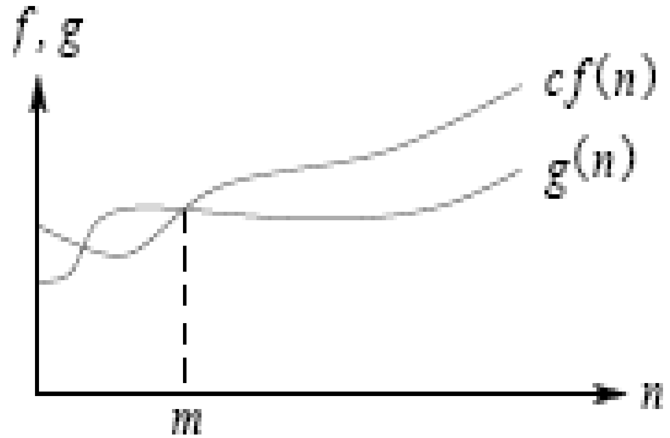
http://www2.dcc.ufmg.br/livros/algoritmos/

Comportamento Assintótico de Funções

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de n.
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de n)
- O comportamento assintótico de f(n) representa o limite do comportamento do custo quando n cresce.

Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **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 $|g(n)| \leq c \times |f(n)|$.



Dominação assintótica

Exemplo:

- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$.

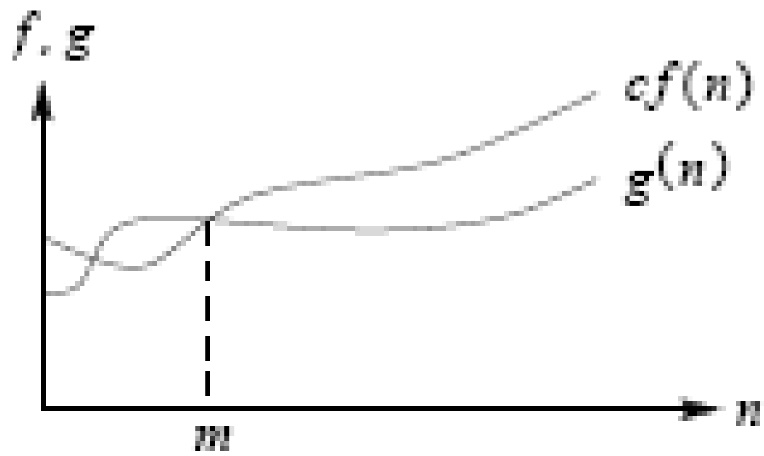
- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, desde que

  $|(n + 1)^2| \leq 4|n^2|$ para $n \geq 1$ e

  $|n^2| \leq |(n + 1)^2|$ para $n \geq 0$.

Notação O

- Escrevemos **g(n) = O(f(n))** para expressar que **f(n)** domina assintoticamente **g(n)**. Lê-se **g(n)** é da ordem no máximo **f(n)**.
- Exemplo: quando dizemos que o tempo de execução **T(n)** de um programa é **O(n²)**, significa que existem constantes **c** e **m** tais que, para valores de **n≥m**, **T(n)≤cn²**.
- Exemplo gráfico de dominação assintótica que ilustra a notação *O*.



Notação O

- O valor da constante m mostrado é o menor valor possível, mas qualquer valor maior também é válido.

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

Exemplos de Notação O

- **Exemplo**: $g(n) = (n + 1)^2$.
  - Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$.
  - Isto porque $(n + 1)^2 \leq 4n^2$ para $n \geq 1$.

- **Exemplo**: $g(n) = n$ e $f(n) = n^2$.
  - Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
  - Entretanto $f(n)$ não é $O(n)$.
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq cn$.
  - 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.

Exemplos de Notação O

- **Exemplo**: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.

  – Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.

  – A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.

- **Exemplo**: $g(n) = \log_5 n$ é $O(\log n)$.

  – O $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$.

  – Como $n = c^{\log_c n}$, tomando o logaritmo base b em ambos os lados da igualdade, temos que

$$\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c.$$

Operações com a Notação O

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

Operações com a Notação O

Exemplo: regra da soma $O(f(n)) + O(g(n))$.

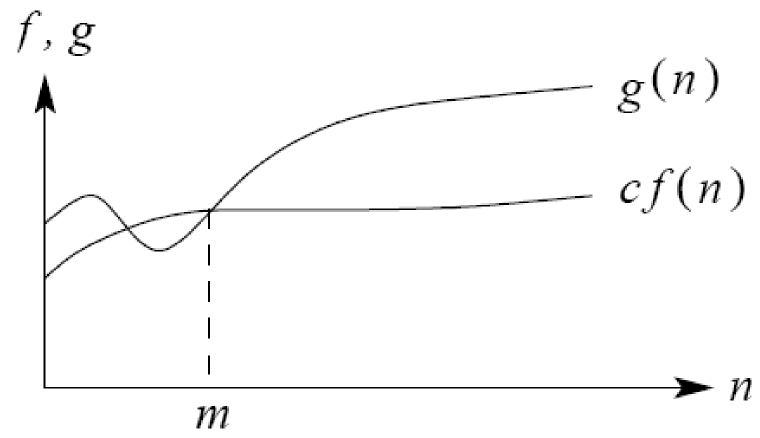
- Suponha três trechos cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$.

- 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 $O(\max(n^2, n \log n))$, que é $O(n^2)$.

Notação Ω

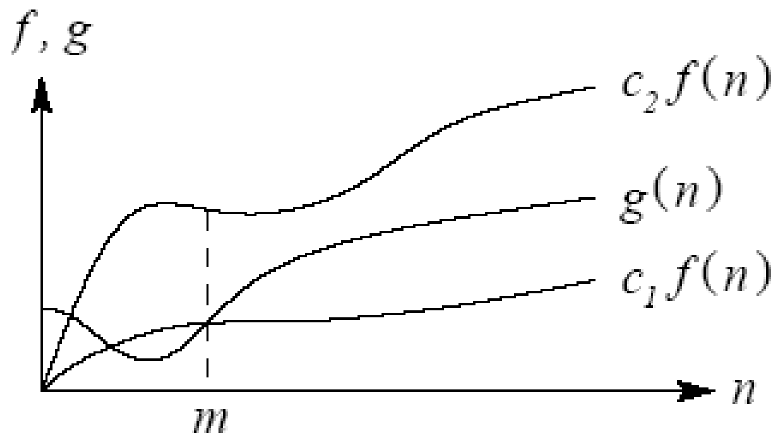
- Especifica um limite inferior para g(n).
- **Definição**: Uma função g(n) é (f(n)) se existirem duas constantes c e m tais queg(n) ≥ cf(n), para todo n ≥ m.
- **Exemplo**: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer c = 1, e então
- $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.



- Para todos os valores à direita de m, o valor de g(n) está sobre ou acima do valor de cf(n).

Notação Θ

- **Definição**: Uma função *g(n)* é *Θ(f(n))* se existirem constantes positivas c_1, c_2 e m tais que $0 \le c_1 f(n) \le g(n) \le c_2 f(n)$, para todo $n \ge m$.
- Exemplo gráfico para a notação:



- Dizemos que *g(n)* = *Θ(f(n))* se existirem constantes c_1, c_2 e m tais que, para todo $n \ge m$, o valor de g(n) está sobre ou acima de $c_1 f(n)$ e sobre ou abaixo de $c_2 f(n)$.
- Isto é, para todo $n \ge m$, a função *g(n)* é igual a *f(n)* a menos de uma constante.
- Neste caso, *f(n)* é um **limite assintótico firme**.

Classes de Comportamento Assintótico

- Se f é uma **função de complexidade** para um algoritmo F, então $O(f)$ é considerada a **complexidade assintótica** ou o comportamento assintótico do algoritmo F.
- A relação de dominação assintótica permite comparar funções de complexidade.
- Entretanto, se as funções f e g dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
- Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.
- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$.
- Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante.

Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.

- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.

- Porém, as constantes de proporcionalidade podem alterar esta consideração.

Comparação de Programas

- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$.

Qual dos dois programas é melhor?

- depende do tamanho do problema.
- Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo $100n$.
- Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
- Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Principais Classes de Problemas

f(n) = O(1)

– Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.

– Uso do algoritmo independe de n.

– As instruções do algoritmo são executadas um número fixo de vezes.

Principais Classes de Problemas

$f(n) = O(\log n)$.

– Um algoritmo de complexidade $O(\log n)$ é dito ter **complexidade logarítmica**.

– Típico em algoritmos que transformam um problema em outros menores.

– Pode-se considerar o tempo de execução como menor que uma constante grande.

– Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.

– Para dobrar o valor de log n temos de considerar o quadrado de n.

– A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Principais Classes de Problemas

f(n) = O(n)

– Um algoritmo de complexidade $O(n)$ é dito ter **complexidade linear**.

– Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.

– É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.

– Cada vez que n dobra de tamanho, o tempo de execução dobra.

Principais Classes de Problemas

$f(n) = O(n \log n)$

– Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.

– Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões.

– Quando n é 2 milhões, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Principais Classes de Problemas

$f(n) = O(n^2)$

– Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**.

– Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.

– Quando n é mil, o número de operações é da ordem de 1 milhão.

– Sempre que n dobra, o tempo de execução é multiplicado por 4.

– Úteis para resolver problemas de tamanhos relativamente pequenos.

Principais Classes de Problemas

f(n) = O(n³)

– Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**.

– Úteis apenas para resolver pequenos problemas.

– Quando n é 100, o número de operações é da ordem de 1 milhão.

– Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Principais Classes de Problemas

f(n) = O(2^n)

– Um algoritmo de complexidade O(2^n) é dito ter **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.

– Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.

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** para na solução do problema.

– $n = 20 \Rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.

– $n = 40 \Rightarrow$ um número com 48 dígitos.

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	$t_4 + 6,6$	$t_4 + 10$

Algoritmos Polinomiais

- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n)$; $c > 1$.
- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- 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 são geralmente simples variações de pesquisa exaustiva.
- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema.
- Um problema é considerado:

– intratável: se não existe um algoritmo polinomial para resolvê-lo.

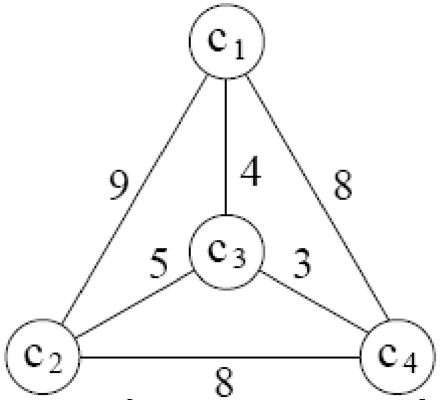
– bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.

Algoritmos Polinomiais x Algoritmos Exponenciais

- 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.
- Também existem algoritmos exponenciais que são muito úteis na prática.
- Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com freqüência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Exemplo de Algoritmo Exponencial

- Um **caixeiro viajante** deseja visitar n cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.
- A figura ilustra o exemplo para quatro cidades c_1, c_2, c_3, c_4, em que os números nos arcos indicam a distância entre duas cidades.



- O percurso < c_1, c_3, c_4, c_2, c_1> é uma solução para o problema, cujo percurso total tem distância 24.

Exemplo de Algoritmo Exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há $(n - 1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é $n!$.
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria $50! \approx 10^{64}$.
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.
- O problema do caixeiro viajante aparece com freqüência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.