Universidade Federal de Viçosa
Campus de Florestal

Algoritmos e Estruturas de Dados I (CCF 211)

Comportamento Assintótico
(Cap 01 – Seção 1.3 Ziviani)

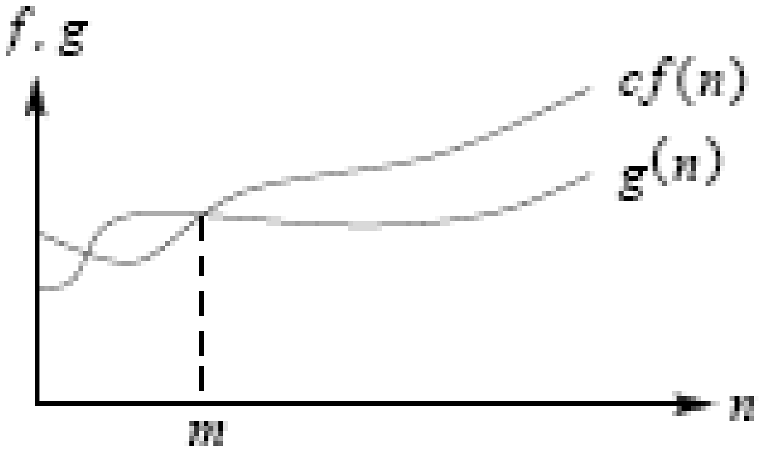
Profa.Thais R. M. Braga Silva
<thais.braga@ufv.br>

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.

Notação O

- Escrevemos **g(n) = O(f(n))** para expressar que **f(n)** domina assintoticamente **g(n)**. Lê-se **g(n)** é da ordem de no máximo **f(n)**.
  - Domínio assintótico: f(n) é sempre maior ou igual a g(n), a partir de certo ponto, quanto n tende a infinito
  - Existem constantes c e m tais que |g(n)| <= c|f(n)|, n >=m
  - A notação O expressa o domínio assintótico. Abaixo, exemplo grafico de dominação assintótica que ilustra a notação O



Notação O

- Quando g(n) é O(f(n)), f(n) nos ajuda a expressar apenas o comportamento da curva de g(n)

- Exemplo: quando dizemos que o tempo de execução **T(n)** de um programa é **O(n²)**, significa o programa é da ordem de no máximo n^2 ($|T(n)| <= c|n^2|$, n >=m)

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$. Para isso basta mostrar que $3n^3 + 2n^2 + n <= 6n^3$ para n >= 0
  - g(n) também é $O(n^4)$. Entretanto essa afirmaçào é mais fraca do que dizer que g(n) é $O(n^3)$

Notação O

- Dizer que g(n) é O(f(n)) significa que f(n) é um limite superior para a taxa de crescimento de g(n)
- Para encontrar f(n) dada uma g(n):
  - Deve-se desprezar as constantes
    - 3n é aproximado para n
  - Termos de menor grau devem ser desprezados
    - $n^2 + n$ é aproximado para n^2
    - $6n^3 + 4n – 9$ é aproximado para n^3

Notação O

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

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.

Classes de Comportamento Assintótico

- 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

- 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.
- **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.

```
IF (Condição) {
   //realiza alguma operação em tempo constante
} ELSE {
   //realiza alguma operação em tempo constante
}
```

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.
- Quando n é mil, $\log_2 n = 10$, quando n é 1 milhão, $\log_2 n = 20$.
- Para dobrar o valor de log n temos de considerar o quadrado de n.

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

$f(n) = O(n)$

```
FOR( int i =0; i < N; i++) {
    //realiza alguma operação em tempo constante
}
```

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 juntam 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

$f(n) = O(n^3)$

- 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 a solução do problema.
- n = 20 → 20! = 2432902008176640000, um número com 19 dígitos.
- n = 40 → 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 (tamanho)	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	$t_4 + 6,6$	$t_4 + 10$

* t = tamanho do problema

Algoritmo Polinomial

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 x Polinomiais

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 Exponenciais x Polinomiais - Exceções

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.

Algoritmos Exponenciais x Polinomiais - Exceções

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.

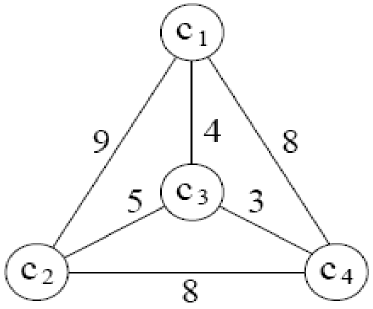
Tais exemplos não ocorrem com freqüência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

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 *c1*, *c2*, *c3*, *c4*, em que os números nos arcos indicam a distância entre duas cidades.



O percurso < *c1*, *c3*, *c4*, *c2*, *c1*> 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! ≈ 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.

Exercícios

Para os próximos algoritmos, identifique a função de complexidade na notação O

Exercício 0

```
void exemplo1 (int n){
    int i, a;
    a=0;
    for (i=0; i<n; i++)
      a+=i;
}
```

Exercício 1

```
// Considere A, B e C vetores globais

void p1 (int n){
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++){
      C[i][j]=0;
      for (k=n-1; k>=0; k--)
          C[i][j]=C[i][j]+A[i][k]*B[k][j];
   }
}
```

Exercício 2

```
void p2 (int n) {
  int i, j, x, y;

  x = y = 0;
  for (i=1; i<=n; i++){
    for (j=i; j<=n; j++)
      x = x + 1;
    for (j=1; j<i; j++)
      y = y + 1;
  }
}
```

Exercício 3

```
void p3 (int n)
{
  int i, j, x, y;

  x = y = 0;
  for (i=1; i<=n; i++)
  {
    for (j=i; j<=n; j++)
      x = x + 1;
  }
}
```