ESTRUTURAS DE DADOS

prof. Alexandre César Muniz de Oliveira

1. Introdução
2. Pilhas
3. Filas
4. Listas
5. Árvores
6. Grafos
7. Complexidade
8. Ordenação
9. Busca

Sugestão bibliográfica:

- **ESTRUTURAS DE DADOS USANDO C**
Aaron M. Tenenbaum, et alli

- **DATA STRUCTURES, AN ADVANCED APPROACH USING C**
Jeffrey Esakov & Tom Weiss

- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED)**
Michael Godrich & Roberto Tamassia

COMPLEXIDADE

1. Eficiência da técnica de ordenação

- Tempo de programação (elaboração) → - Análise do Algoritmo; - Projeto do Programa; - Escrita; - Validação
- Tempo de execução
- Espaço necessário para execução

2. Tempo de execução

Varia conforme máquinas, programas, e dados de entrada.

3. Análise matemática (complexidade do algoritmo)

- Melhor caso, pior caso, caso médio
- Exemplo (1):

Código	Operações primitivas
tmpMax := A[0]	2
for i:=1 to n-1 do	1+n*(1)+(n-1)*2
if tmpMax < A[i] then	(n-1)*2
tmpMax := A[i]	(n-1)*2
return tmpMax	1

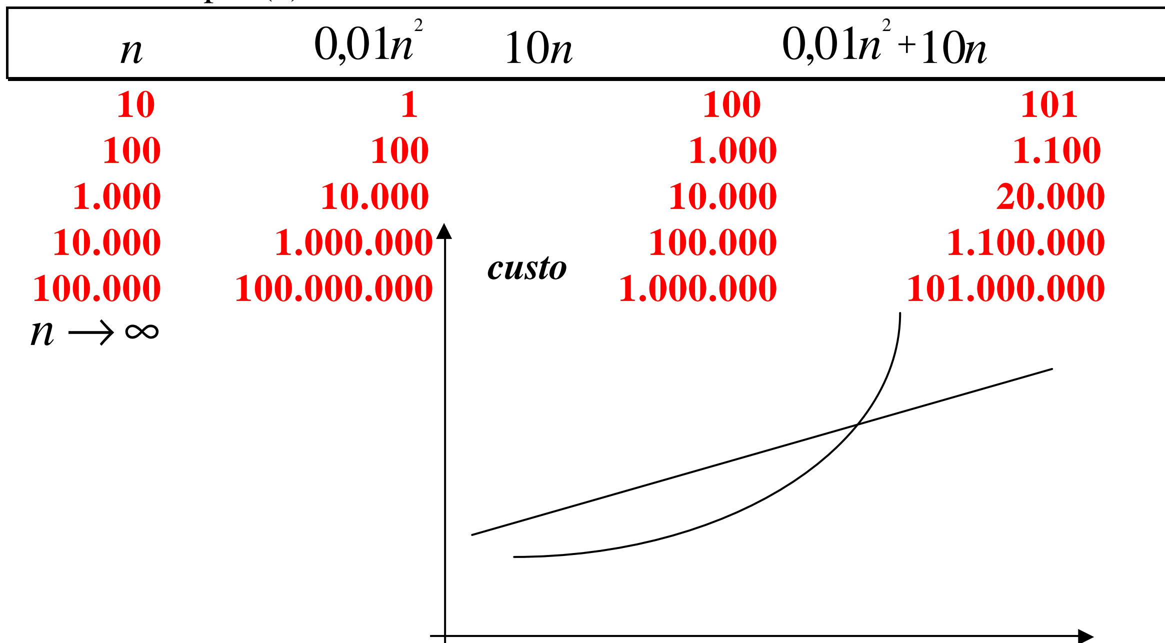
Número de operações primitivas executadas:

T(n) = 2+1+n+(n-1)*2+(n-1)*2+1 = 5n (melhor caso)

T(n) = 2+1+n+(n-1)*2+(n-1)*2+(n-1)*2+1 = 7n-2 (pior caso)

- Exemplo (2):

n	$0{,}01n^2$	$10n$	$0{,}01n^2+10n$
10	**1**	**100**	**101**
100	**100**	**1.000**	**1.100**
1.000	**10.000**	**10.000**	**20.000**
10.000	**1.000.000**	**100.000**	**1.100.000**
100.000	**100.000.000**	**1.000.000**	**101.000.000**
$n \to \infty$			



Possibilidades: $n < n\log n < n^2$

COMPLEXIDADE

4. Noções de complexidade de algoritmos

Na análise da complexidade de algoritmos é importante concentrar-se na taxa de crescimento do tempo de execução como uma função do tamanho de entrada n, obtendo-se um quadro geral do comportamento. Assim para o exemplo basta saber que o tempo de execução de algoritmo cresce proporcionalmente a n. (O tempo real seria n*factor constante, que depende de software e hardware).

4.1 Comportamento Assintótico de Funções

Comportamento a ser observado em uma função f(n), quando n tende ao infinito. O custo assintótico de uma função f(n) representa o limite do comportamento de custo quando n cresce. Em geral, o custo aumenta com o tamanho n do problema. Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado.

f(n) domina assintoticamente g(n) se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos $|g(n)| \leq c\,|f(n)|$

Seja $g(n) = n$ e $f(n) = -n^2$

Temos que $|n| \leq |-n^2|$ para todo $n \in N$. Fazendo $c = 1$ e $n_0 = 0$ a definição é satisfeita. Logo, f(n) domina assintoticamente g(n).

4.1 Notação O (*big O* ou *"O" grande*)

Notação trazida da matemática por Knuth (1968):

g(n) = O(f(n)), Lê-se:

- g(n) é de ordem no máximo f(n)
- f(n) domina assintoticamente g(n)
- f(n) é um limite assintótico superior para g(n)

O(f(n)) representa o conjunto de todas as funções que são assintoticamente dominadas por uma dada função f(n).

NOTAÇÃO O

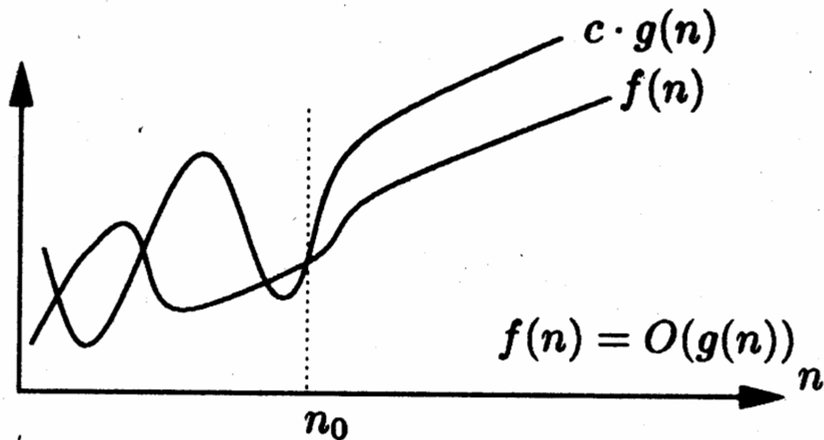
Se $g(n) \in O(f(n))$ então $g(n) = O(f(n))$

Formalmente:

$$g(n) = O(f(n)), \quad \exists\, c > 0 \text{ e } n_0 \;/\; 0 \leq g(n) \leq c\, f(n), \forall n \geq n_0$$

COMPLEXIDADE

Graficamente:



A notação O é usada para expressar o limite superior do tempo de execução de cada algoritmo para resolver um problema específico (limite superior de cada algoritmo para um problema).

OPERAÇÕES

- **f(n) = O(f(n))**
- **cO(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))**

Exemplos:

a) Suponha 3 trechos de programas cujos tempos de execução sejam O(n), O(n^2) e O(nlogn). O tempo de execução dos 2 primeiros trechos é O(max(n,n2)) = O(n^2). O tempo de execução de todos os 3 trechos é, então, O(max(n^2, nlogn)), que é O(n2).

b) O produto de O(logn) por O(n) é O(n logn)

c) g(n) = $(n+1)^2$ provar que g(n) = O(n^2) ?
Se existirem duas constantes positivas c e n_0 tais que, para $n \geq n_0$:
$|(n+1)^2| \leq c\,|n^2| \setminus |(n^2+2n+1)| \leq c\,|n^2| \setminus |(1+2/n+1/n^2)| \leq c$
Para n=1 todo $c \geq 1+2+1 \geq 4$ satisfaz a condição. Para n infinito?

n	*$(n+1)^2$*	*$4n^2$*
0	*1*	*0*
1	*4*	*4*
2	*9*	*16*
3	*16*	*36*
4	*25*	*64*

COMPLEXIDADE

4.2 Notação W (*big* W ou "W" *grande*)

$$g(n) = \Omega(f(n))$$

- Lê-se g(n) é de ordem no mínimo f(n).

- Ω define um limite inferior para a função, por um fator constante.

***f (n) é* um limite assintótico inferior para g(n)**

Se $g(n) \in \Omega(f(n))$ então $g(n) = \Omega(f(n))$

Formalmente:

$$\mathbf{g(n) = \Omega(f(n)), \exists\ \exists\ c > 0 \ \ e \ \ n_0 \ \ | \ \ 0 \leq cf(n) \leq g(n), \forall\ \forall\ n \geq n_0}$$

A notação Ω é usada para expressar um limite inferior intrínseco ao problema. O **limite inferior** para qualquer algoritmo de **ordenação** é Ω(*n)* (vetor já ordenado).

4.3 Notação q (*big* q ou "q" *grande*)

$$g(n) = \Theta(f(n))$$

- Lê-se g(n) é da mesma ordem de f(n).
- Θ limita a função por fatores constantes.

f (n) é um limite assintótico *superior e inferior* para g(n)

ou

f(n) é um limite assintótico firme para g(n).

Θ diz que para $n \geq n_o$, o valor de *g(n)* está sempre entre $c_1 f(n)$ e $c_2 f(n)$ inclusive.

Formalmente:

$g(n) = \Theta(f(n)), \exists\ \exists\ c_1>0\ ,\ c_2>0$ e n_o | $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\ ,\ \forall\ \forall\ n \geq n_o$

Exemplo: Seja: $1/2n^2 - 3n = \Theta(n^2)$

Prova : Para provar esta afirmação encontre constantes $c_1>0$, $c_2>0$, $n>0$, tal que $c_1 n^2 \leq 1/2n^2-3n \leq c_2 n^2$ para todo $n \geq n_0$

Ao dividir a expressão acima por n^2 tem-se: $c_1 \leq ½ -3/n \leq c_2$

A inequação mais a direita será sempre válida para qualquer valor de $n \geq 1$ ao escolher $c_2 \geq 1/2$.

A inequação mais a esquerda será sempre válida para qualquer valor de $n \geq 7$ ao escolher $c_1 \geq 1/14$.

Assim: ao escolher $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, verifica-se que $1/2n^2-3n = \Theta(n^2)$.

COMPLEXIDADE

5. Análise assintótica de funções:

Pode-se dizer que:

- O(f(n)) depende do algoritmo (limite superior)
- **W**(f(n)) depende do problema (limite inferior)
- (f(n)) depende de ambos ("limite ótimo")

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. 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, sejam F e G dois algoritmos aplicados à mesma classe de problemas. O algoritmo F leva três vezes o tempo de G para ser executado, ou seja, 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.

6. Classes de comportamento assintótico

f (n) = O(1) (complexidade constante)

O uso do algoritmo independe do tamanho de n. Neste caso as instruções do algoritmo são executadas um número fixo de vezes.

f(n)= O(n) (complexidade de linear)

Em geral um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho o tempo de execução dobra.

f(n) = O(log n) (complexidade logarítmica)

Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Por exemplo, quando n é um milhão, log2 n ≈ 20.

f(n)= O(nlogn)

Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois ajuntando as soluções. Por exemplo, quando n é um milhão e a base do logaritmo é 2, nlog2 n é cerca de 20 milhões.

f(n)= O(n^2) (complexidade quadrática)

Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro. Por exemplo, quando n é mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos.

f(n)= O(2^n) (complexidade exponencial)

Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa força bruta para resolvê-los. Por exemplo, quando n é vinte, o tempo de execução é cerca de um milhão. Problemas que somente podem ser resolvidos através de algoritmos exponenciais são ditos *intratáveis*.

7. Operações críticas para ordenação e busca

Para ordenação e busca as operações primitivas relevantes (operações críticas) são :

- comparar chave
- trocar elementos

Incrementar um índice, por exemplo, não é considerado uma operação crítica.

5.1 Ordenação Bolha

É o método de ordenação mais básico que se conhece: é o primeiro a ser utilizado por estudantes de computação. Ele funciona da seguinte maneira: compara-se X_i a X_{i+1} e, se X_i for maior (ou menor, dependendo da ordem), é feita a troca de posição. A classificação é encerrada quando não há mais trocas.

INTERAÇÃO	
0	25 57 48 37 12 92 86 33
1	25 48 37 12 57 86 33 92
2	25 37 12 48 57 33 86 92
3	25 12 37 48 33 57 86 92
4	12 25 37 33 48 57 86 92
5	12 25 33 37 48 57 86 92
6	12 25 33 37 48 57 86 92
7	12 25 33 37 48 57 86 92

A cada alteração um elemento é posicionado corretamente, logo:

Arquivo com n elementos → interações (n-1)

A cada interação i, todos os elementos acima ou igual posição a n-i já estão classificados. Existe a possibilidade de todo o arquivo estar classificado antes das (n-1) interações máximas. Existe uma melhoria: marcar a última posição com troca em que houve troca e não fazer comparações a partir daí.

Algoritmo:

```
Para cada { interação e troca > 0} faça
    Para j de 0 até trocou faça
        Se X[j] > X [j+1]
            trocou ¬ j;
            troca (X[j], X[j+1]);
        Fim se;
    Fim para;
Fim para;
```

Eficiência:

01. Tempo de programação: simples
02. Espaço para armazenamento: apenas uma variável auxiliar para troca
03. Tempo de execução: f(comparações e trocas)

MELHOR CASO:

- Conjunto ordenado:

n comparações e 0 troca → O(n)

PIOR CASO:

- Conjunto desordenado:

$$(n-1) + (n-2) + (n-3) + \ldots + (n-k)$$

$$(2kn-k^2-k)/2 \Rightarrow k \to n-1 \rightarrow k \text{ é } O(n)$$

$$\text{logo: } O(n^2)$$