01.1 - Introdução à Complexidade de Algoritmos

BCC - Estruturas de Dados

Prof. Dr. Paulo César Rodacki Gomes
paulo.gomes@ifc.edu.br

Blumenau, 2022

Roteiro

1. Algoritmos
2. Ordens Assintóticas de Complexidade
3. Recursividade e Análise de Algoritmos Recursivos
4. Considerações Finais

1: Algoritmos

- Algoritmo: qualquer método especial para resolver um certo tipo de problema

- Algoritmo: seqüência de passos computacionais não ambíguos que transforma a entrada em saída

Escolha de um algoritmo

- simplicidade: mais fácil de implementar, menor probabilidade de erros
- clareza: clareza no código fonte. Algorimtos mais simples tender a levar a códigos mais legíveis e mais fáceis de manter
- eficiência: tempo que um algoritmo leva para resolver uma instância do problema

Análise de Algoritmos

- estimar a quantidade de recursos requeridos pelo algoritmo, em especial o **tempo** necessário para solucionar um problema.
- Em geral, ao se analizar vários algoritmos que solucionam um determinado problema, o mais eficiente é identificado.
- Além disso, esta análise pode indicar mais de um candidato e apontar vários algoritmos inferiores que devem ser descartados.

Análise de Algoritmos

Modelo computacional RAM (Cormen et al., 2002):

- operações simples (+, -, etc) consomem uma unidade de tempo para serem executadas
- laços e chamadas a sub-rotinas não são considerados operações simples
- cada operação de acesso a memória consome uma unidade de tempo
- operações são executadas seqüencialmente, sem processos concorrentes

Análise de Algoritmos

Abordagem por *benchmarking*

- os algoritmos precisam ser implementados a priori
- certos algoritmos são intratáveis computacionalmente, seu tempo para produzir uma resposta é impraticável
- é melhor fazer análise teórica da complexidade de um algoritmo
- instâncias diferentes podem produzir tempos diferentes
- diferentes tamanhos de problemas, idem.

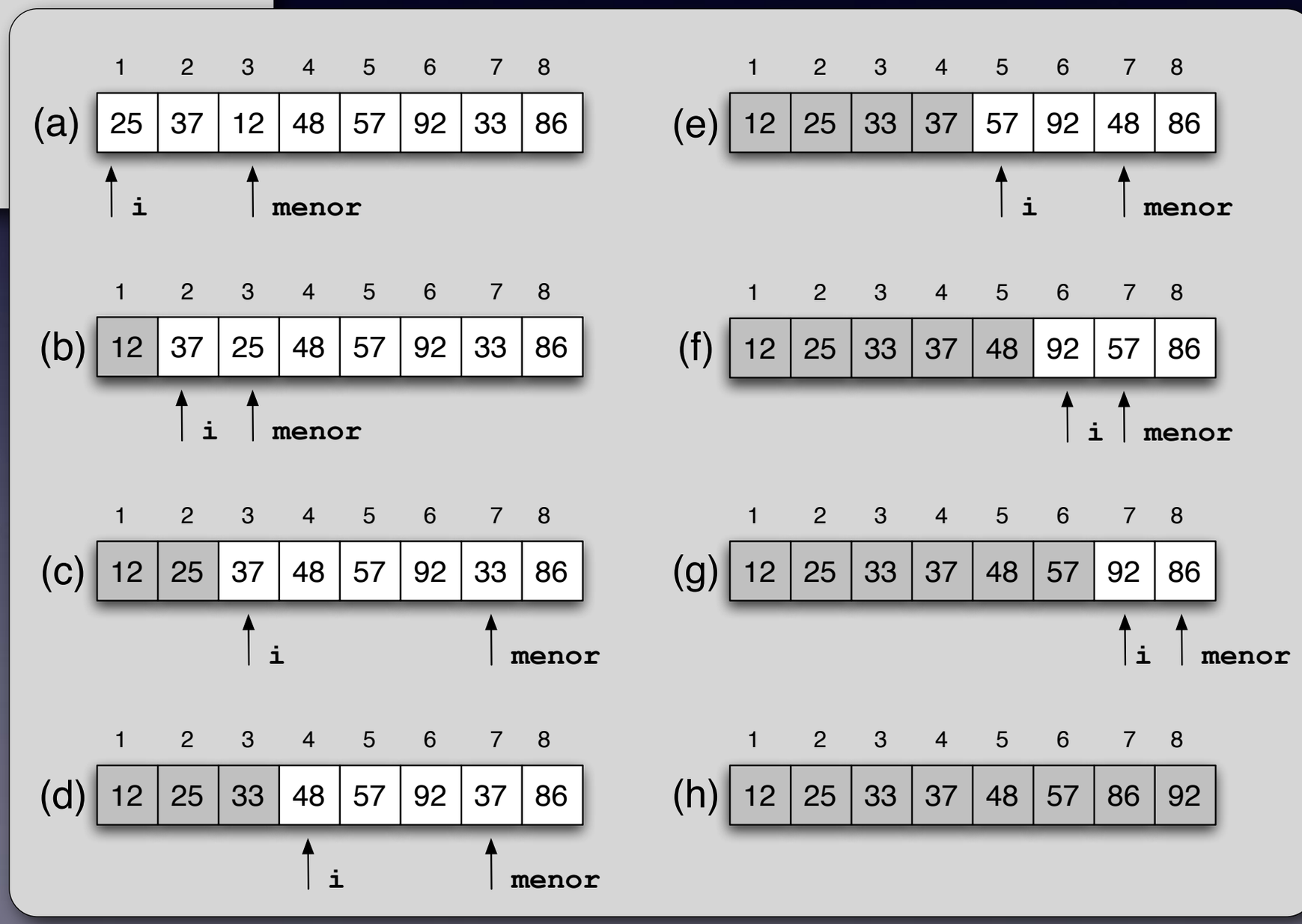
Tamanho de entrada

- Depende do problema:
    - para ordenação o tamanho pode ser a quantidade de valores a ordenar
    - para sistemas de n equações e n incógnitas, o tamanho pode ser n
    - grafos: vertices + arestas

- **O tempo de execução** será uma função do tamanho de entrada: $T(n) = c.f(n)$

```
1 Algoritmo: SelectionSort(A)
2 para i ← 1 até n − 1 faça
3 |   menor ← i;
4 |   para j ← i + 1 até n faça
5 |   |   se A[j] < A[menor] então
6 |   |   |   menor ← j;
7 |   temp ← A[menor];
8 |   A[menor] ← A[i];
9 |   A[i] ← temp;
```

Exemplo



Cálculo do tempo de execução

- verificar quantas vezes a k-ésima linha, de custo c_k, é executada
- t_i: quantidade de vezes que o laço da linha 4 é executado

```
1 Algoritmo: SelectionSort(A)
2 para i ← 1 até n − 1 faça
3 |   menor ← i;
4 |   para j ← i + 1 até n faça
5 |   |   se A[j] < A[menor] então
6 |   |   |   menor ← j;
7 |   temp ← A[menor];
8 |   A[menor] ← A[i];
9 |   A[i] ← temp;
```

$$T(n) = an^2 + bn + c$$

Tabela 1.1: Custo do SelectionSort

linha	custo	n.º de execuções
2	c_2	n
3	c_3	n-1
4	c_4	$\sum_{i=1}^{n-1} t_i$
5	c_5	$\sum_{i=1}^{n-1}(t_i - 1)$
6	c_6	$\sum_{i=1}^{n-1}(t_i - 1)$
7	c_7	n-1
8	c_8	n-1
9	c_9	n-1

Ordem de Crescimento

- Tempo de execução: $T(n) = an^2 + bn + c$
- As constantes a, b e c dependem dos custos c_i
- Nós ignoramos os custos c_i...
- O que nos interessa é a taxa de crescimento ou ordem de crescimento dos tempos de execução de algoritmos
- Quando observamos tamanhos de entrada suficientemente grandes, somente a taxa de crescimento é relevante → eficiência assintótica

Taxa de Crescimento de Funções

n	$\xi = \log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 anos
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 s	$10^{15} anos$
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 dias	
10^2	0.007 μs	0.10 μs	0.644 μs	10 μs	8×10^{13} anos	
10^3	0.010 μs	1 μs	9.966 μs	1 ms		
10^4	0.013 μs	10 μs	130 μs	100 ms		
10^5	0.017 μs	0.1 ms	1.67 ms	10 s		
10^6	0.020 μs	1 ms	19.93 ms	16.7 min		
10^7	0.023 μs	0.01 s	0.23 s	1.16 dias		
10^8	0.027 μs	0.10 s	2.66 s	115.7 dias		
10^9	0.030 μs	1 s	29.90 s	31.7 anos		

2: Ordens assintóticas de complexidade

Análise assintótica de algoritmos:

- avaliação da tendência de crescimento do tempo de execução de algoritmos à medida que o tamanho de entrada do problema tende a um limite
- **complexidade do algoritmo**: quantidade de "trabalho" requerido pelo algoritmo
- ordens assintóticas de complexidade seguem notações assintóticas definidas como funções no domínio dos números naturais

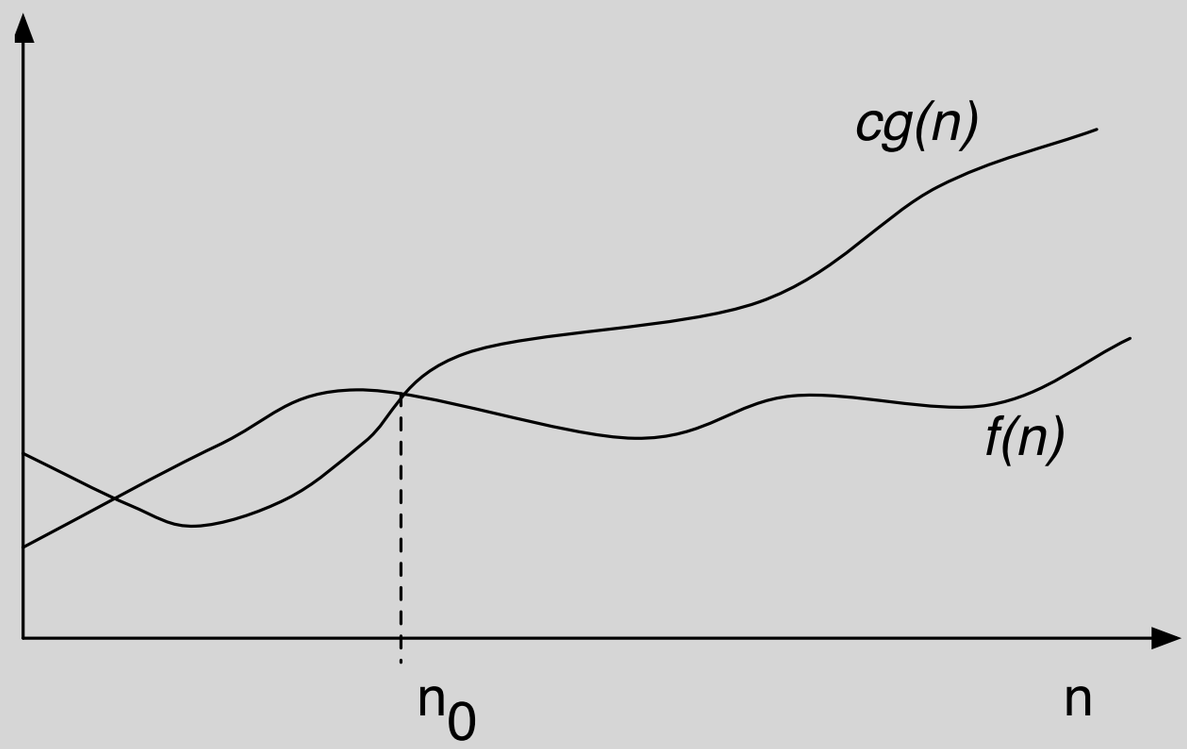
Notações assintóticas

- ordens assintóticas de complexidade seguem notações assintóticas definidas como funções no domínio dos números naturais
- as notações são usadas para descrever o tempo de execução assintótico de um algoritmo
- notações básicas:
  - O, limite superior
  - Θ, limite restrito
  - Ω, limite inferior

Notação *O*

Para uma dada função $g(n)$ (com n $\in \mathbb{N}$), denotamos por $O(g(n))$ o *conjunto de funções*

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todos } n \geq n_0\}.$$



$$f(n) = O(g(n))$$

Notação *O*

Princípios gerais:

- Fatores constantes não importam:

> Para qualquer constante $d > 0$ e
>
> qualquer função $f(n)$, $f(n)$ é $O(df(n))$

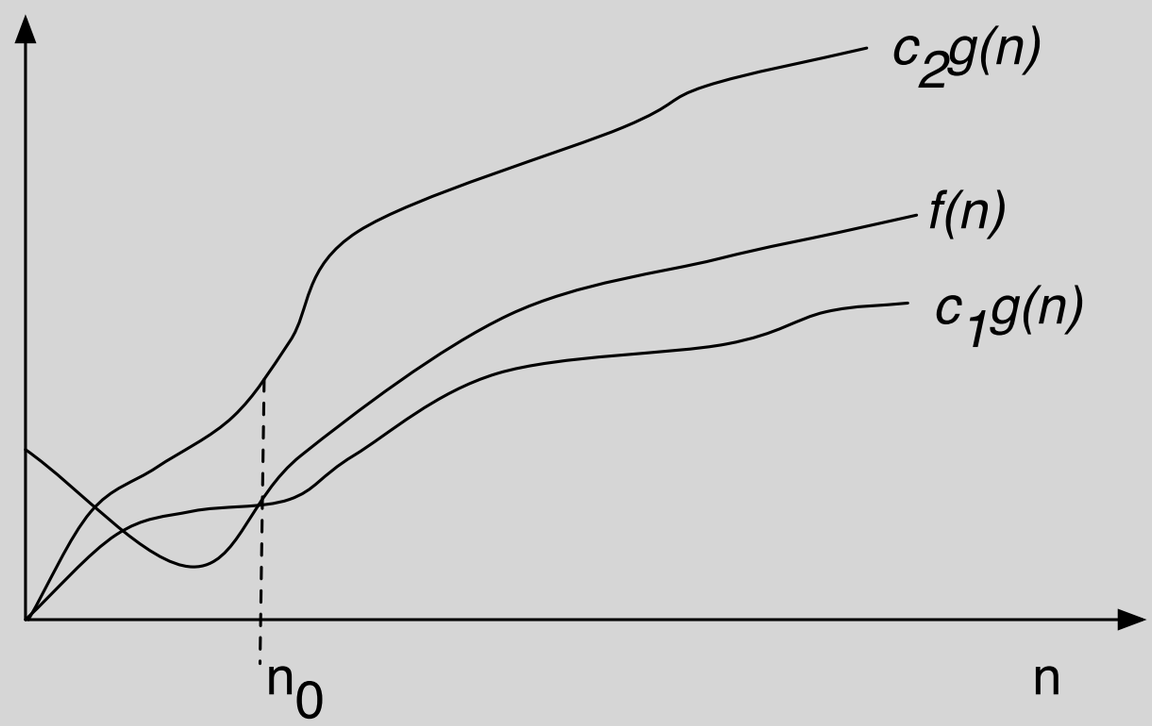
- Termos de menor ordem podem ser desprezados

> Para $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$,
> se $a_k > 0$, então $f(n) = O(n^k)$

Notação Θ

Para uma dada função $g(n)$ (com n $\in \mathbb{N}$), denotamos por $\Theta(g(n))$ o *conjunto de funções*

$$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1,\ c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todos } n \geq n_0\}.$$

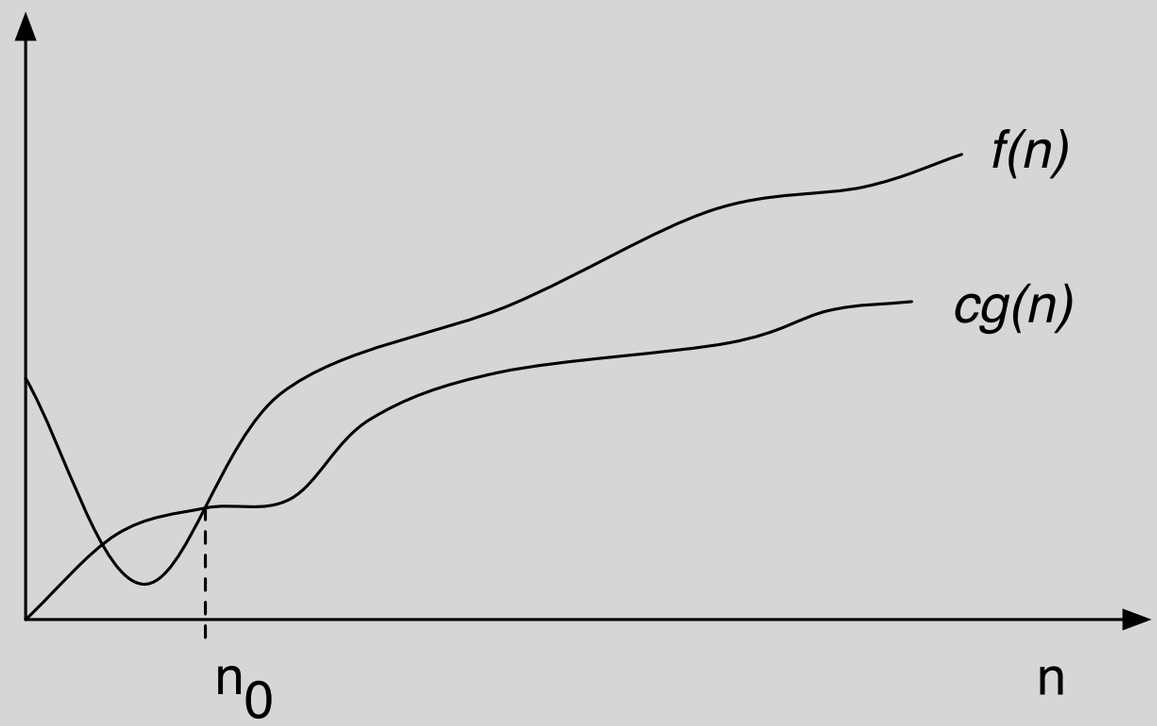


$$f(n) = \Theta(g(n))$$

Notação Ω

Para uma dada função $g(n)$ (com n $\in \mathbb{N}$), denotamos por $\Omega(g(n))$ o *conjunto de funções*

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n) \text{ para todos } n \geq n_0\}.$$



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

Exemplo: BubbleSort

25	**37**	12	48	57	92	33	86	25 x 37
25	**37**	**12**	48	57	92	33	86	37 x 12 troca
25	12	**37**	**48**	57	92	33	86	37 x 48
25	12	37	**48**	**57**	92	33	86	48 x 57
25	12	37	48	**57**	**92**	33	86	57 x 92
25	12	37	48	57	**92**	**33**	86	92 x 33 troca
25	12	37	48	57	33	**92**	**86**	92 x 86 troca
25	12	37	48	57	33	86	**92**	final da 1ª passada

25	**12**	37	48	57	33	86	**92**	25 x 12 troca
12	**25**	**37**	48	57	33	86	**92**	57 x 37
12	25	**37**	**48**	57	33	86	**92**	37 x 48
12	25	37	**48**	**57**	33	86	**92**	48 x 57
12	25	37	48	**57**	**33**	86	**92**	57 x 33 troca
12	25	37	48	33	**57**	**86**	**92**	57 x 86 troca
12	25	37	48	33	57	**86**	**92**	final da 2ª passada

12	**25**	37	48	33	57	**86**	**92**	12 x 25
12	**25**	**37**	48	33	57	**86**	**92**	25 x 37
12	25	**37**	**48**	33	57	**86**	**92**	37 x 48
12	25	37	**48**	**33**	57	**86**	**92**	48 x 33 troca
12	25	37	33	**48**	**57**	**86**	**92**	48 x 57 troca
12	25	37	33	48	**57**	**86**	**92**	final da 3ª passada

12	**25**	37	33	48	**57**	**86**	**92**	12 x 25
12	**25**	**37**	33	48	**57**	**86**	**92**	25 x 37
12	25	**37**	**33**	48	**57**	**86**	**92**	37 x 33 troca
12	25	33	**37**	**48**	**57**	**86**	**92**	37 x 48 troca
12	25	33	37	**48**	**57**	**86**	**92**	final da 4ª passada

Cálculo do tempo de execução

```
1  Algoritmo: BubbleSort(A)
2  int i, j;
3  para i ← n até 1 faça
4      boolean troca ← falso;
5      para j ← 1 até (i-1) faça
6          se A[j] > A[j + 1] então
7              int temp ← A[j];
8              A[j] ← A[j + 1];
9              A[j + 1] ← temp;
10             troca ← verdadeiro;
11     se troca ≠ verdadeiro então retorna
```

Tabela 2.1: Custo do BubbleSort

linha	custo	n.º de execuções
2	c_2	1
3	c_3	n+1
4	c_4	n
5	c_5	$\sum_{i=1}^{n} t_i$
6, 7, 8, 9, 10	$c_6, c_7, c_8, c_9, c_{10}$	$\sum_{i=1}^{n}(t_i - 1)$
11	c_{11}	n

$$T(n) = c_2 + c_3(n+1) + c_4 n + c_5 \sum_{i=1}^{n} t_i + (c_6 + c_7 + c_8 + c_9 + c_{10}) \sum_{i=i}^{n} (t_i - 1) + c_{11} n.$$

$$T(n) = c_2 + c_3 + c_4 + c_5 n + c_6(n-1) + c_{11} \therefore$$

$$T(n) = (c_5 + c_6)n + (c_2 + c_3 + c_4 + -c_6 + c_{11}).$$

$$T(n) = an + b$$

O tempo de execução do *BubbleSort* é linear no melhor caso e quadrático no pior, portanto podemos dizer que é $\Omega(n)$ e $O(n^2)$.

Comparações assintóticas

Muitas propriedades relacionais que se aplicariam a números, tambem se aplicam a notações assintóticas

- transitividade
- reflexividade
- simetria

Transitividade

"se $A \leq B$ e $B \leq C$, então $A \leq C$".

- se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$;
- se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$;
- se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$, então $f(n) = \Omega(h(n))$.

Reflexividade

- $f(n) = \Theta(f(n))$;
- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$.

Simetria

- Simetria: $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$;
- Simetria transposta: $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$.

3: Recursividade

Definições recursivas:

A. Uma ou mais regras-base nas quais objetos simples ou elementares são definidos, e

B. Uma ou mais regras indutivas, pelas quais objetos maiores são definidos em termos de objetos menores pertencentes à coleção de objetos definidos.

Exemplo: função fatorial

$$n! = \begin{cases} 1, & \text{se } n \in \{0, 1\} \ (\textit{regra-base}); \\ n(n-1)!, & \text{se } n > 1 \ (\textit{regra indutiva}). \end{cases}$$

Algoritmos recursivos

Geralmente utilizam uma estratégia de divisão e conquista para resolver o problema

A. Divisão: dividir o problema em sub-problemas menores
B. Conquista: feita pela resolução de sub-problemas elementares
C. Combinação: sub-problemas resolvidos são combinados para constituirem a solução de um sub-problema maior

$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq c; \\ aT(n/b) + D(n) + C(n), & \text{caso contrário.} \end{cases}$$

Análise de complexidade de Algoritmos Recursivos

A. Método da árvore de recorrência: consiste em expandir a árvore de recorrência e expressar a soma de seus termos em função de n e de certas condições iniciais

B. Método de resolução por substituição: inicialmente estima-se uma solução hipotética para o problema e utiliza-se indução matemática para determinação das constantes envolvidas e provar corretude da solução

C. Teorema geral: método geral para resolução de equações de recorrência na forma $T(n) = aT(n/b) + f(n)$, onde $a \geq 1$, $b > 1$ e $f(n)$ é assintoticamente positiva

Método da Árvore de Recorrência

Exemplo: MergeSort

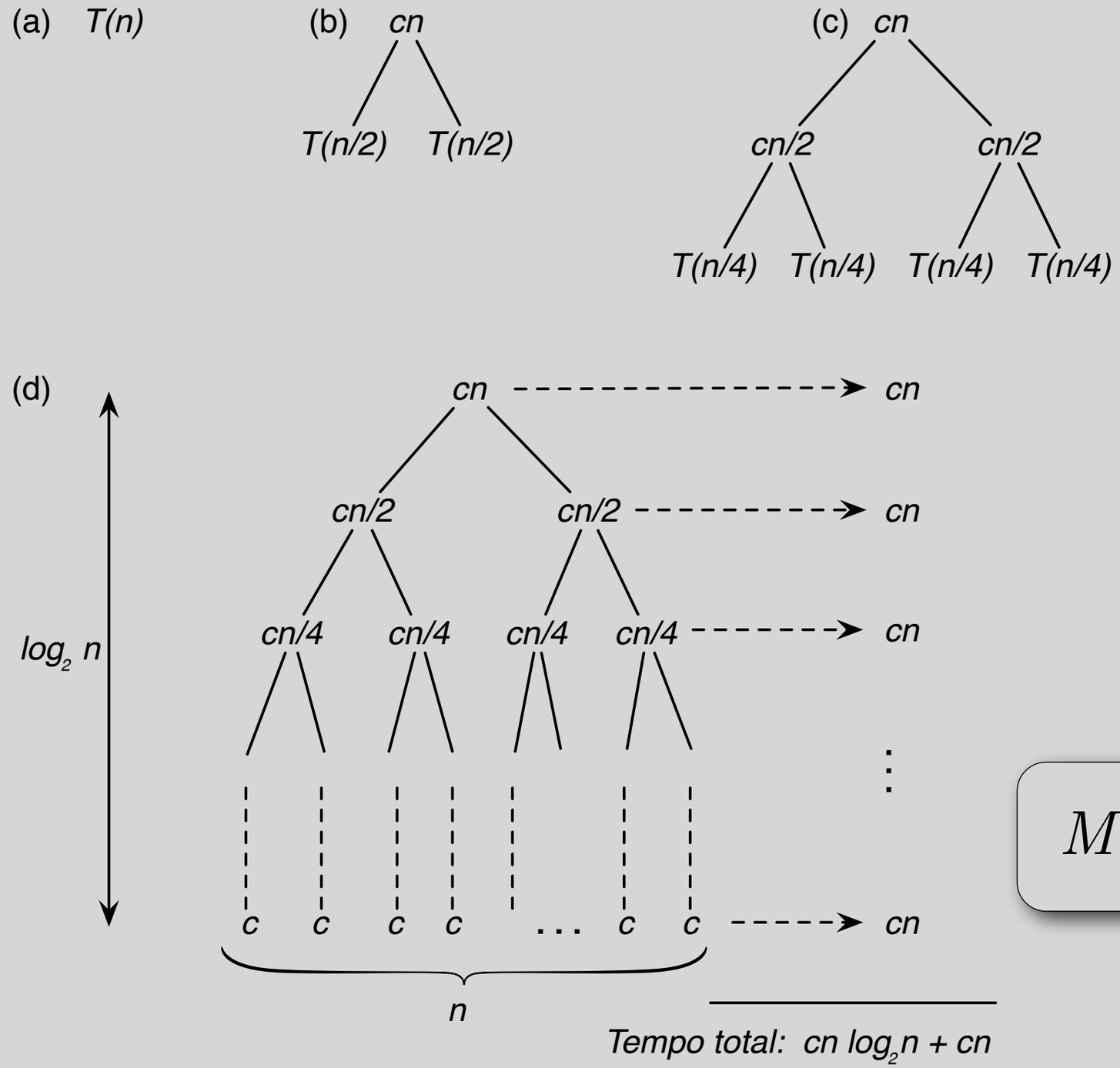
$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq c; \\ aT(n/b) + D(n) + C(n), & \text{caso contrário.} \end{cases}$$

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases}$$

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases}$$

$$\rightarrow$$

$$T(n) = \begin{cases} c, & \text{se } n = 1; \\ 2T(n/2) + cn, & \text{se } n > 1. \end{cases}$$



$MergeSort$ é $\Theta(n \log n)$

Método da Árvore de Recorrência

Estimativa inicial: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ é $T(n) = O(n^2)$.

Prova: queremos provar que $T(n) \leq dn^2$ para alguma constante $d > 0$.

$$T(n) \leq 3T(\lfloor n/4 \rfloor) + cn^2 \therefore T(n) \leq 3d\lfloor n/4 \rfloor^2 + cn^2$$

$$T(n) \leq 3d(n/4)^2 + cn^2 \therefore T(n) = \frac{3}{16}dn^2 + cn^2 \therefore T(n) \leq dn^2.$$

Onde $T(n) \leq dn^2$ é satisfeito com $d \geq (16/13)c$.

Teorema geral

Teorema Geral
Sejam as constante a e b onde $a \geq 1$ e $b > 1$. Seja $f(n)$ uma função e seja $T(n)$ definida no domínio dos inteiros não negativos pela equação de recorrência $T(n) = aT(n/b) + f(n)$, onde n/b pode ser interpretado tanto por $\lfloor n/b \rfloor$ quanto por $\lceil n/b \rceil$. Então $T(n)$ pode ser limitada assintoticamente da seguinte forma:

1. se $f(n) = O(n^{\log_b a - \varepsilon})$ para uma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;
2. se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
3. se $f(n) = \Omega(n^{\log_b a + \varepsilon})$, para uma constante $\varepsilon > 0$ e se $af(n/b) \leq cf(n)$ para uma constante $c < 1$ e para todos n suficientemente grandes, então $T(n) = \Theta(f(n))$.

Considerações finais

- Introdução à Análise de Algoritmos como parte dos requisitos para avaliação da eficiência das estruturas de dados a serem implementadas na disciplina
- Conteúdo dividido em 3 partes:
  1) Algoritmos,
  2) Ordens Assintóticas de Complexidade, e
  3) Recusividade
- Devido à extensão do tema (Complexidade de Algoritmos), apenas conceitos iniciais foram apresentados.

Obrigado!

Prof. Paulo C. Rodacki Gomes