GA-026: Algoritmos I
Introdução

Prof. Luiz Gadelha

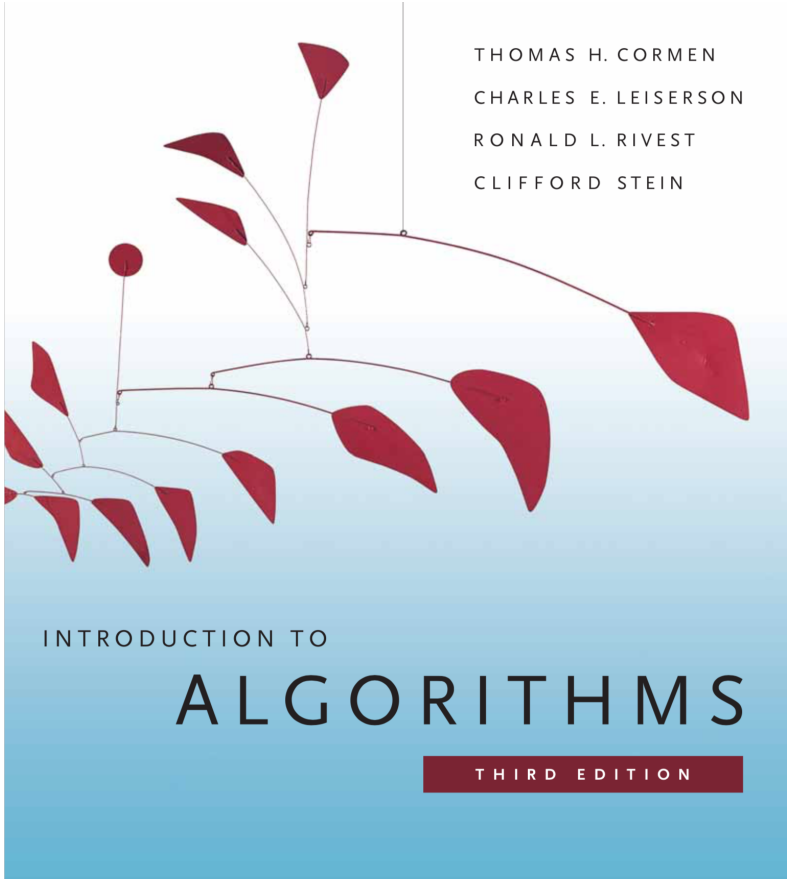
Programa de Pós-Graduação em Modelagem Computacional, P4/2019
Laboratório Nacional de Computação Científica

24 de setembro de 2019

Visão Geral do Curso

- Objetivo geral do curso:
  - Análise de complexidade de algoritmos.
  - Análise de correção de algoritmos.
  - Técnicas para projeto de algoritmos.
  - Algoritmos para problemas clássicos:
    - Ordenação.
    - Busca.
    - Operações com Matrizes.
    - Grafos.
  - Projeto e análise de algoritmos multiprocessados.
- Informações sobre o curso:
  - Atendimento às quartas de 15-16:30 na sala 2A-07.
  - Página do curso:
    `https://www.lncc.br/~lgadelha/GA026.html`

Livro-texto da Parte I do Curso



- Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009). Introduction to Algorithms. MIT Press.

Avaliação do Curso

- Agenda:
  - Exercícios (30%).
  - Trabalho (35%).
  - Prova (35%).
- Trabalho:
  - Sobre algum algoritmo a combinar com o professor (p.ex. da parte "Selected Topics" do livro-texto).
  - Apresentação de 20min descrevendo o algoritmo, sua complexidade e correção.
  - Implementação e demonstração.
  - Relatório de até 10 páginas.

Agenda do Curso

- Agenda:

24/09 Introdução a Algoritmos, Ordenação por inserção.
26/09 Mergesort, Crescimento de funções, notação assintótica.
01/10 Dividir para conquistar: problema do sub-vetor máximo, algoritmo de Strassen.
03/10 Relações de recorrência, método de substituição.
08/10 Relações de recorrência, árvores de recorrência, método mestre.
10/10 Ordenação: introdução, Heapsort.
15/10 Ordenação: Quicksort, análise de complexidade com árvores de decisão.
17/10 Busca: sequencial, árvores binárias de busca, tabelas de hash.
22/10 Busca: tabelas de hash, árvores balanceadas.
24/10 Algoritmos multiprocessados: introdução, multiplicação de matrizes.
29/10 Algoritmos multiprocessados: mergesort.
31/10 Técnicas para projeto de algoritmos: programação dinâmica.

Agenda do Curso

- Agenda:

05/11 Técnicas para projeto de algoritmos: programação dinâmica.
07/11 Técnicas para projeto de algoritmos: programação dinâmica.
12/11 Técnicas para projeto de algoritmos: algoritmos gulosos.
14/11 Algoritmos para grafos: representação, buscas em largura e profundidade.
19/11 Algoritmos para grafos: ordenação topológica.
21/11 Algoritmos para grafos: árvores de cobertura mínimas.
26/11 Algoritmos para grafos: caminhos mínimos.
28/11 Algoritmos para grafos: caminhos mínimos.
03/12 Algoritmos para álgebra linear: solução de sistemas equações lineares.
05/12 Algoritmos para álgebra linear: inversão de matrizes.
10/12 Apresentações de trabalhos.
12/12 Apresentações de trabalhos.
17/12 Prova.

Algoritmos

- Um **algoritmo** é um procedimento computacional bem definido que toma um ou mais valores como **entrada** e produz um ou mais valores como **saída**.
- Pode ser visto também como uma ferramenta para resolver um **problema computacional** bem especificado.
- O problema define a relação entre a entrada e a saída.
- O algoritmo descreve um procedimento computacional para obter a relação entre a entrada e a saída.

Algoritmos

- O **problema da ordenação** (ordenar não-decrescentemente uma sequência de números):
  - **Entrada:** Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Saída:** Uma permutação da sequência de entrada $\langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- P.ex. Dada a sequência de entrada $\langle 34, 76, 45, 98, 12, 64, 47 \rangle$, um algoritmo de ordenação deve gerar como saída a sequência $\langle 12, 34, 45, 47, 64, 76, 98 \rangle$.

Algoritmos

- Um algoritmo é considerado **correto** quando, para cada instância de entrada, ele para produzindo a saída correta (i.e. a saída definida pelo problema computacional).
- Neste caso, dizemos que o algoritmo **soluciona** o problema computacional.
- Algoritmos incorretos podem não parar para determinadas instâncias de entrada ou produzir saídas incorretas.
- Algoritmos incorretos podem ser úteis também, quando uma aproximação para a saída esperada é suficiente.
- Pode ser que existam problemas computacionais para os quais não existam algoritmos que os solucionem (problemas indecidíveis).

Algoritmos

- O *Problema da Correspondência de Post* é um exemplo de problema indecidível.
- Dados um conjunto de dominós da forma:

$$\frac{a_1 a_2 \ldots a_n}{b_1 b_2 \ldots b_m}$$

  onde $a_1 a_2 \ldots a_n$ e $b_1 b_2 \ldots b_m$ são palavras de um alfabeto.
- O problema consiste de determinar se é possível colocar os dominós em uma sequência tal que se tenha a mesma palavra na parte de cima e na parte de baixo dos dominós.

Aplicações de Algoritmos

- Na bioinformática as sequências biológicas obtidas com sequenciadores precisam:
  - ter seus fragmentos montados para obtenção de sequências completas;
    - montagem de grafos de *de Bruijn* e identificação de caminhos *Eulerianos* nos mesmos;
  - ser comparadas com sequências existentes em bancos de dados de sequências (p.ex. GenBank) para identificação;
    - algoritmos para casamento aproximado de caracteres (p.ex. Smith-Waterman);

Aplicações de Algoritmos

- A Internet apresenta diversos problemas que requerem algoritmos:
  - Roteamento de pacotes TCP/IP.
    - algoritmos para grafos;
  - Busca de páginas na web.
    - busca por palavras/termos em textos na web;
    - contagem de referências a uma página (PageRank);
  - Comércio eletrônico seguro.
    - algoritmo RSA para criptografia de chaves públicas (teoria dos números);

Ordenação por Inserção

- O algoritmo de **ordenação por inserção** soluciona o problema de ordenação.
- O **problema da ordenação** (ordenar não-decrescentemente uma sequência de números):
  - **Entrada:** Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Saída:** Uma permutação da sequência de entrada $\langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Chamaremos os elementos da sequência de **chaves**.

Ordenação por Inserção

- O algoritmo de **ordenação por inserção** (ou *InsertionSort*) soluciona o problema de ordenação.
- O **problema da ordenação** (ordenar não-decrescentemente uma sequência de números):
  - **Entrada:** Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Saída:** Uma permutação da sequência de entrada $\langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Chamaremos os elementos da sequência de **chaves**.

Ordenação por Inserção

- Idéia básica:
  - Uma parte da sequência já está ordenada (que representaremos em azul).
  - Uma chave (que representaremos em vermelho) adjacente à essa parte é inserida na mesma.
- Exemplo de ordenação por inserção:

$$\langle 12, 4, 75, 34, 28, 9 \rangle$$

$$\langle 4, 12, 75, 34, 28, 9 \rangle$$

$$\langle 4, 12, 75, 34, 28, 9 \rangle$$

$$\langle 4, 12, 34, 75, 28, 9 \rangle$$

$$\langle 4, 12, 28, 34, 75, 9 \rangle$$

$$\langle 4, 9, 12, 28, 34, 75 \rangle$$

Ordenação por Inserção

```
1:  procedure InsertionSort(A[1, ..., n])
2:      for j ← 2 to j ≤ A.lenght do
3:          key ← A[j]
4:          i ← j − 1
5:          while i > 0 and A[i] > key do
6:              A[i + 1] ← A[i]
7:              i ← i − 1
8:          end while
9:          A[i + 1] ← key
10:     end for
11: end procedure
```

Ordenação por Inserção: Correção

- Uma maneira para demonstrar que um *invariante de laço* (IdL), uma propriedade se mantém válido ao longo das iterações.
- No caso da ordenação um IdL é dado por:
  - a subsequência $A[1, \ldots, j-1]$ está ordenada e é constituída por uma permutação dos elementos que originalmente estavam nessas posições.
- Podemos usar indução matemática para demonstrar o IdL.

Ordenação por Inserção: Correção

- para $j = 2$, a subsequência $A[1, \ldots, 1]$ é unitária, é naturalmente ordenada;
- como nenhum elemento foi deslocado ainda, é constituída por uma permutação do elemento que originalmente estava nessa posição.
- Logo a IdL é válida antes da primeira iteração do algoritmo.

Ordenação por Inserção: Correção

- para uma iteração onde $j = k$ $(2 < k \leq n)$
  - o algoritmo desloca para a direita os elementos $A[k-1], A[k-2], \ldots$ até encontrar um que seja menor que $A[k]$, onde ele é inserido;
  - assim, o algoritmo mantém $A[1, \ldots, k]$ ordenada e como uma permutação desses elementos.
- Assim a IdL se mantém válida após uma iteração do algoritmo.

Ordenação por Inserção: Correção

- O laço **for** termina pois a cada iteração j é incrementado, o que garante que a condição de parada será satifeita ($j > A$.lenght).
- Nesse momento, o valor de j será $n + 1$.
- Pelo IdL, sabemos que $A[1, \ldots, n]$ está ordenada e é uma permutação da sequência original.
- Logo, o algoritmo é correto!

Ordenação por Inserção: Análise de Complexidade

```
1:  procedure InsertionSort(A[1, ..., n])                    ▷ Custo, Vezes
2:      for j ← 2 to j ≤ A.lenght do                          ▷ c1, n
3:          key ← A[j]                                        ▷ c2, n − 1
4:          i ← j − 1                                         ▷ c3, n − 1
5:          while i > 0 and A[i] > key do                     ▷ c4, Σ_{j=2}^{n} t_j
6:              A[i + 1] ← A[i]                               ▷ c5, Σ_{j=2}^{n} (t_j − 1)
7:              i ← i − 1                                     ▷ c6, Σ_{j=2}^{n} (t_j − 1)
8:          end while
9:          A[i + 1] ← key                                    ▷ c7, n − 1
10:     end for
11: end procedure
```

Ordenação por Inserção: Análise de Complexidade

Custo total:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j +$$

$$c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1) =$$

$$c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) + (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7$$

Ordenação por Inserção: Análise de Complexidade

Melhor caso (lista já ordenada), $t_j = 1$:

$$T(n) = c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) + (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7$$

$$= c_4(n-1) + (c_5 + c_6)0 + (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7$$

$$= (c_4 + c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7 - c_4$$

$$= an - b$$

Logo, $T(n)$ é uma função linear de n.

Ordenação por Inserção: Análise de Complexidade

Pior caso (lista reversamente ordenada), $t_j = j$:

$$T(n) = c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) + (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7$$

$$= c_4 \left(\frac{n(n+1)}{2} - 1\right) + (c_5 + c_6)\frac{n(n-1)}{2} + (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7$$

$$= \frac{c_4 + c_5 + c_6}{2} n^2 + \left(\frac{c_4 - c_5 - c_6}{2} + c_1 + c_2 + c_3 + c_7\right)n - \frac{c_4}{2} - c_2 - c_3 - c_7$$

$$= an^2 + bn + c$$

Logo, $T(n)$ é uma função quadrática de n.

Análise de Complexidade

- No caso da ordenação por inserção, analisamos o melhor e pior caso para o tempo de execução.
- Seria possível analisar o caso médio também (exercício).
- É comum analisar apenas pior caso:
  - é um limite superior para o tempo de execução, assegura que não levaria mais tempo que esse limite;
  - o pior caso pode ocorrer com frequência (busca de uma informação ausente em um banco de dados);
  - o caso médio frequentemente tem a mesma ordem de grandeza (p.ex. pior e melhor caso quadráticos).
- A análise em caso médio é mais difícil pois às vezes não é claro o que seria o caso médio mas esta pode ser apoiada por **análise probabilística**.

Análise de Complexidade: Ordem de Crescimento

- No caso da ordenação por inserção, o tempo de execução no pior caso é da forma $an^2 + bn + c$.
- Para *n grande*, $bn + c$ são menos relevantes para determinar o comportamento de crescimento de $T(n)$.
- O coeficiente do termo que domina o crescimento da função, an^2 também pode ser ignorado.
- Assim, podemos dizer que a ordenação por inserção tem tempo de execução $\Theta(n^2)$ no pior caso.

Obrigado!

E-mail: lgadelha@lncc.br