Projeto e Análise de Algoritmos
Análise de Complexidade

Prof. Luiz Chaimowicz

AGENDA – Modulo 1

Data	Assunto	Capítulos
05/03	Algoritmos / Invariantes / Intro Análise de Complexidade	1,2
07/03	Não Haverá Aula	
12/03	Intro Análise de Complexidade / Notação Assintótica	3
14/03	Recursão / Eq. de Recorrência	4
19/03	Recursão / Eq. de Recorrência	4
21/03	Análise Probabilística	5
26/03	Algoritmos Randomizados (Intro) / Análise Amortizada	17
28/03	**Prova 1 e Entrega da Lista 1**	

O que é um Algoritmo?

Algorithm

An *algorithm is any well-defined computational procedure that takes* some value, or set of values, as *input and produces some value, or set of values, as output. An algorithm is thus a* ***sequence of computational steps that transform the*** **input into the output**

[Cormen, Chapter 1]

Origem da Palavra

Abu Ja'Far Mohammed Ibn Musa **al-Khowarizmi** (780–850), astrônomo e matemático árabe. Era membro da "Casa da Sabedoria", uma academia de cientistas em Bagdá. O nome al-Khowarizmi significa da cidade de Khowarizmi, que agora é chamada Khiva e é parte do Uzbequistão. al-Khowarizmi escreveu livros de matemática, astronomia e geografia. A álgebra foi introduzida na Europa ocidental através de seus trabalhos. A palavra álgebra vem do árabe al-jabr, parte do título de seu livro Kitab al-jabr w'al muquabala. Esse livro foi traduzido para o latim e foi usado extensivamente. Seu livro sobre o uso dos numerais hindu descreve procedimentos para operações aritméticas usando esses numerais. Autores europeus usaram uma adaptação latina de seu nome, até finalmente chegar na palavra **algoritmo** para descrever a área da aritmética com numerais hindu.

Slide: Prof. Antônio Alfredo Loureiro

Algoritmos

- Presentes em todas as áreas da computação na resolução dos mais diversos tipos de problemas

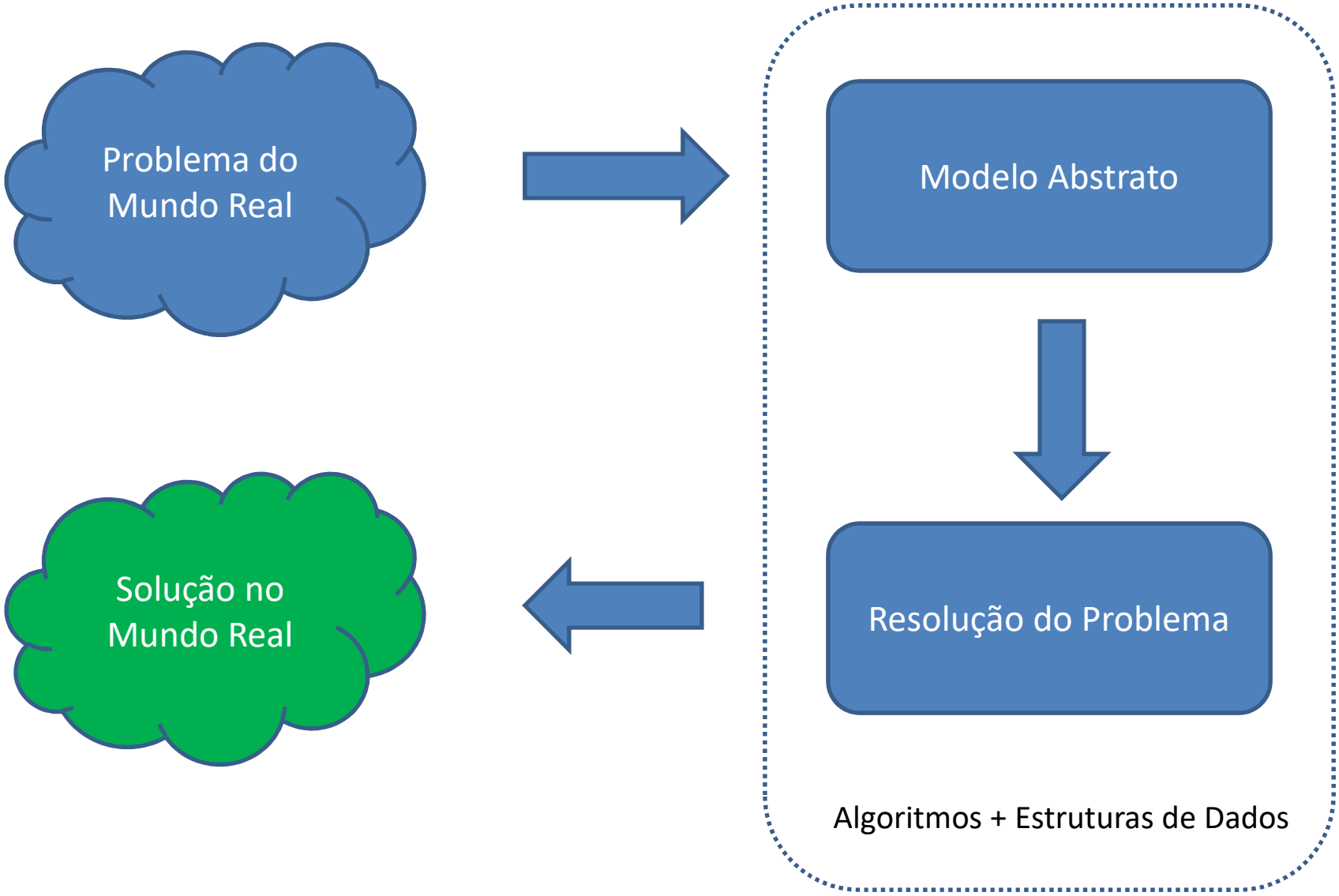
- Permitem que problemas do mundo real possam ser trabalhados de forma estruturada e consequentemente possam ser resolvidos por um computador

Estrutura de Dados

- Uma estrutura de dados é uma forma de se armazenar e organizar os dados de um determinado problema de forma a facilitar o acesso e modificações por um algoritmo

- Diferentes estruturas de dados se aplicam a diferentes problemas e algoritmos

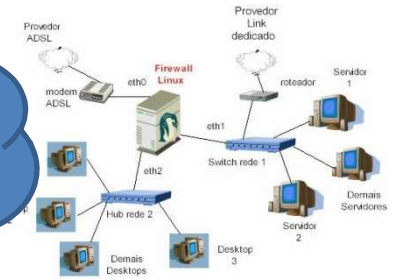
Abstração


Problema do Mundo Real
Modelo Abstrato
Resolução do Problema
Solução no Mundo Real
Algoritmos + Estruturas de Dados

Abstraction

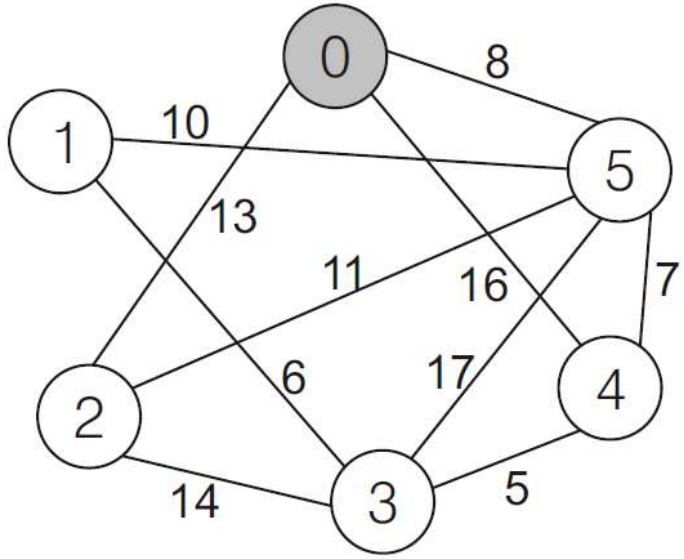
Diferentes problemas podem usar a mesma abstração

Roteamento de Mensagens



Caminho de carro entre 2 cidades

Navegação de um robô no ICEx



Grafo

+

Algoritmo de Djkstra

Problema: Ordenação

Dado um conjunto de números inteiros
ordená-los em ordem crescente

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Existem **vários algoritmos** para resolver
esse problema

Escolha de Algoritmos

- Para escolher um algoritmo para um determinado problema, devemos considerar diversas características.

- Mas duas das principais são:
  - O algoritmo **funciona?**
  - O algoritmo é **eficiente?**

Exemplo: *Insertion Sort*

- Um dos algoritmos mais simples para se resolver o problema da ordenação

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Exemplo: *Insertion Sort*

Em ritmo de dança romena...

AlgoRythmics: http://www.youtube.com/watch?v=ROalU379l3U

O *Insertion Sort* Funciona?

- **Loop Invariants**
  - Ajudam a entender / provar se um algoritmo é correto.
  - De certa forma, similar a uma prova por indução

Loop Invariants

Define-se uma propriedade de interesse do algoritmo e verifica-se se ela é satisfeita

Inicialização: a propriedade é verdadeira antes do início da primeira iteração do loop

Manutenção: se a propriedade é satisfeita antes da iteração, ela permanece verdadeira após a iteração

Terminação: quando o loop termina, o invariante permite verificar se o algoritmo é correto

Invariante para o *Insertion Sort*

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Ao início de cada iteração do **for** (linhas 1..8) o subvetor *A[1..j-1]* consiste dos elementos originais de *A[1..j-1]* mas de forma ordenada

Invariante para o *Insertion Sort*

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Inicialização: ao início do vetor, quando j=2, o subvetor A[1..j-1] contém apenas um elemento que está ordenado

Invariante para o *Insertion Sort*

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Manutenção: o loop interno compara o elemento a[j] com os seu antecessores e os move até encontrar a posição de inserção do elemento. Com isso, o novo subvetor [1..j] fica ordenado.

Invariante para o *Insertion Sort*

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Terminação: o loop termina quando j = n+1.

Logo pelo invariante, o subvetor A[1..j-1], estará ordenado, ou seja:

A[1..n] estará ordenado

O *Insertion Sort* é Eficiente?

- Análise de algoritmos
  - Recursos necessários para a execução do algoritmo: **tempo**, memória, etc...

- Necessário um **modelo computacional**
  - Modelo abstrato do funcionamento do computador na resolução de algoritmos
  - Foca nas operações relevantes para a análise

Modelo Computacional

- ***RAM – Random Access Machine***
  - Um processador que executa uma ação por vez
  - Memória que armazena os dados
  - Operações básicas de custo constante
    - Acesso a memória
    - Testes condicionais
    - Operações aritméticas
    - Etc...

Análise de algoritmos

- Para analisar um algoritmo, vamos estudar os recursos necessários para a sua execução no modelo computacional escolhido
- Cada operação executada pelo processador, incluindo cálculos aritméticos lógicos e acesso a memória, implica num **custo de tempo:**
  - Função de complexidade de tempo.
- Cada operação e dado armazenado na memória, implica num **custo de espaço:**
  - Função de complexidade de espaço.

Análise de Algoritmos

- Função de Complexidade de Tempo: número de vezes que uma operação de interesse (ou todas as operações) é executada em função **do tamanho de entrada n**

- Requer o uso de matemática discreta: somatórios, combinações recorrências, etc...

Análise do *Insertion Sort*

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Operação sendo analisada
Comparação de elementos

Melhor Caso: *f(n) = n-1* → $\Theta(n)$

Pior Caso: *f(n) = n(n-1)/2*

Caso Médio: *análise de probabilidades*

$\Theta(n^2)$

“Para Casa”

- Estudar Capítulos 1 e 2 do Cormen
- Resolver o seguinte exercício:

  Implemente o **algoritmo da seleção** no qual um vetor é ordenado selecionado-se o menor elemento e trocando-o em A[1], o segundo menor e trocando-o com A[2], etc...

  1 - O seu algoritmo funciona (use invariantes)?

  2 - Qual a sua função de complexidade para o número de comparações e de trocas?

Solução

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

- **Invariante:** o vetor *A[1..j-1]* contém os *j-1* menores elementos do vetor e esses se encontram ordenados

Solução

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

- **Inicialização:** *j=1*, logo o vetor tem 0 elementos o que satisfaz o invariante

Solução

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

- **Manutenção:** cada iteração do loop pesquisa o j-ésimo menor elemento e o coloca na posição *j*. Logo, ao final da iteração o sub-vetor a *A[1..j]* é acrescido de um elemento em sua ordem correta

Solução

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

- **Terminação:** ao final do loop, *j = n* e pelo invariante, todos os elementos *A[1..n-1]* estão ordenados. O elemento *A[n]* restante é o maior de todos e se encontra na sua posição.
- **Logo, o algoritmo funciona!**

Solução

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Comparação de elementos

Troca de elementos

- Função de Complexidade - número de comparações: ***f(n) = n(n-1)/2***
- Função de Complexidade - número de trocas: ***f(n) = n-1***