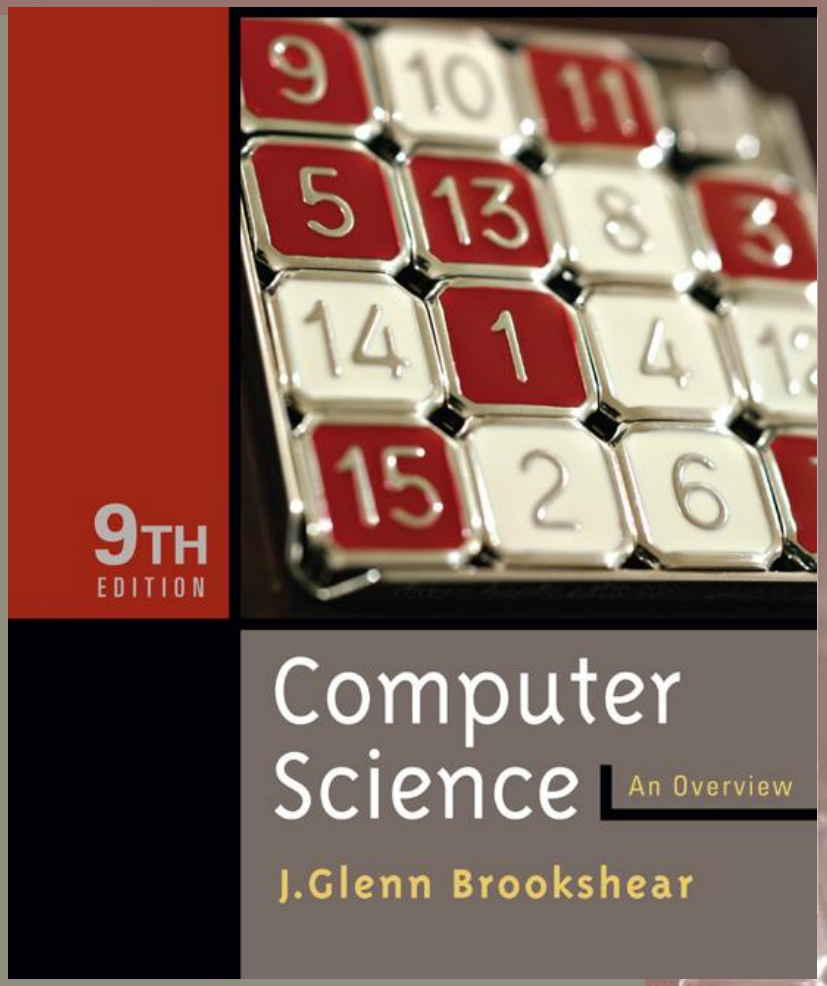
Capítulo 5

Algoritmos



Modificados para MIM - Introdução à Informática (2425) por Pedro Ribeiro, Miguel Areias e Nuno Guimarães

Capítulo 5: Algoritmos

- 5.1 O conceito de algoritmo
- 5.2 Representação algorítmica
- 5.3 Desenho de algoritmos
- 5.4 Estruturas iterativas
- 5.5 Estruturas recursivas
- 5.6 Eficiência e correção

Algoritmo: definição

Um algoritmo é um conjunto **ordenado** de instruções **executáveis** e **não ambíguas** que definem como um processo **termina**.

- Lista de todos os inteiros positivos: não termina
- Organize os documentos importantes na pasta: ambíguo
  - o que são documentos importantes?
  - são os documentos que estão dentro ou fora da pasta?

Algoritmo: níveis de abstração

- **Problema:** motivação para o algoritmo
- **Algoritmo:** procedimento para resolver o problema
  - Geralmente uma de várias possibilidades
- **Representação:** descrição suficiente do algoritmo para que este possa ser executado
  - Geralmente uma de várias possibilidades

Algoritmo: Pseudocódigo

- Sistema para representação informal de código
  - Usado no desenvolvimento de algoritmos
- Sintaxe e semântica semelhantes a linguagem de programação mas menos formal que a mesma
  - Conhecimento prévio da linguagem onde o algoritmo vai ser implementado

Primitivas - pseudocódigo

- Atribuição *nome* ← *expressão*
- Seleção condicional **if** *condição* **then** *acção*
- Execução repetida **while** *condição* **do** *actividade*
- Procedimento **procedure** *nome* (nome genérico)

“Hello” em pseudocódigo

```
procedure Greetings
Count ← 3;
while (Count > 0) do
    (print the message “Hello” and
        Count ← Count –1)
```

Passos para resolver problemas (George Pólya, matemático húngaro)

- 1. Perceber o problema.
- 2. Pensar num plano (algoritmo) para o resolver.
- 3. Executar o plano (formular o algoritmo e implementá-lo como um programa).
- 4. Avaliar a solução (programa) na sua correção e potencial como ferramenta para resolver outros problemas.

Algumas abordagens usuais

- **Objetivo -> get your foot on the door**
  - Resolver o problema "ao contrário" (de trás para a frente)

    Começar pelo output
  - Resolver um problema relacionado mais simples

    Relaxando algumas das restrições

    Resolvendo partes do problema primeiro (metodologia *bottom up)*
  - Refinamento passo a passo: *top-down*

    Dividir o problema em vários sub-problemas

    Técnica popular que produz programas modulares

Resolvendo um problema

- Vamos tentar resolver um problema.

- Dada uma lista com nomes, **procurar** se um dado nome existe na lista:
  - Entrada (***input***): lista de nomes e nome a procurar
  - Ex: [Pedro, Carlos, Ana, João, Sofia]
  - Saída (***output***): sim ou não

Criando o pseudo-código

- **procedure** procura (lista, elemento):
  - (corpo do procedimento)
  - (no final "**return** true" ou "**return** false")

- Primitiva básica de manipulação de listas:
  - **lista[i]** permite aceder ao *i*-ésimo elemento
  - Ex: lista = [Pedro, Carlos, Ana, João, Sofia]
  - lista[1] é Pedro; lista[2] é Carlos

Pesquisa linear

- Pesquisar os elementos um a um e verificar se o elemento está em alguma posição

- [**Pedro**, Carlos, Ana, João, Sofia] Está na 1ª?
- [Pedro, **Carlos**, Ana, João, Sofia] Está na 2ª?
- [Pedro, Carlos, **Ana**, João, Sofia] Está na 3ª?
- [Pedro, Carlos, Ana, **João**, Sofia] Está na 4ª?
- [Pedro, Carlos, Ana, João, **Sofia**] Está na 5ª?

Pesquisa linear (estrutura iterativa)

- **procedure** procura(lista, elemento):
  - posicao ← 1
  - tamanho ← tamanho(lista)
  - **while** (posicao <= tamanho) **do**:
    - **if** (lista[posicao] = elemento) **then**:
      - **return** true
    - posicao ← posicao + 1
  - **return** false

Componentes da estrutura iterativa (com "ciclo")

- **Inicializar:** estabelecer uma condição inicial que vai ser modificada à medida que vamos evoluindo para a condição de terminação
- **Testar:** comparar o "estado" corrente com a condição de terminação e terminar se for esse o caso
- **Modificar:** mudar o estado de forma a que caminhemos para a condição de terminação

Estrutura recursiva

- Como resolver este problema sem ciclos?
- Imaginemos que pode chamar **recursivamente** o mesmo procedimento...
  -

Pesquisa linear (estrutura recursiva)

- **procedure** procura(lista, elemento):
  - **if** (lista[1] = elemento) **then**:
    - **return** true
  - novaLista ← lista retirando o 1º elemento
  - **return** procura(novaLista, elemento)

- Falta algo? Quando é que o procedimento termina ?

Pesquisa linear (estrutura recursiva)

- **procedure** procura(lista, elemento):
  - **if** (lista = []) **then**:
    - **return** false
  - **if** (lista[1] = elemento) **then**:
    - **return** true
  - novaLista ← lista retirando o 1º elemento
  - **return** procura(novaLista, elemento)

('[]' significa lista vazia)

Condição de Terminação

Eficiência de um programa

- Como avaliar a eficiência de um programa?
  - número de instruções/operações executadas
- Notação Θ para classes
  - Ex: pesquisa linear é Θ(n)
- Avaliar o melhor caso, o pior caso e caso médio
  - Ex: o pior caso na pesquisa linear é percorrer a lista toda e não encontrar o elemento da procura
    - Se a lista tem n elementos então o pior caso é percorrer os n elementos da lista

Revisitar a pesquisa de um elemento

- Será possível fazer melhor?
- Imaginemos que a lista está ordenada
- Ex: [Ana, Bruno, Carla, Duarte, Hugo, Paula, Pedro, Rui, Rute, Vasco]

Pesquisa linear (estrutura iterativa) [assumindo que a lista está ordenada]

- **procedure** procura(lista, elemento):
  - posicao ← 1
  - tamanho ← tamanho(lista)
  - **while** (posicao <= tamanho) **do**:
    if (lista[posicao] = elemento) **then**:
    return true
    if (lista[posicao] > elemento) **then**:
    return false
    posicao ← posicao + 1
  - **return** false

Melhora caso médio? E pior?

Pesquisa binária

- Como pesquisam um nome numa lista telefónica?
- Ideia:
  - Ver o nome no meio da lista
  - O nome do meio é igual, encontramos.
  - Caso seja menor (alfabeticamente) procurar **recursivamente** na primeira metade da lista
  - Caso seja maior, procurar **recursivamente** na segunda metade da lista

Pesquisa binária (continuação)

- 1: Ana
- 2: Bruno
- 3: Carla
- 4: Duarte
- 5: Hugo
- 6: Paulo
- 7: Pedro
- 8: Rui
- 9: Rute
- 10: Vasco

Procurar **Rute**

Pesquisa binária (continuação)

- 1: Ana
- 2: Bruno
- 3: Carla
- 4: Duarte
- **5: Hugo ← elemento do meio**
- 6: Paulo
- 7: Pedro
- 8: Rui
- 9: Rute
- 10: Vasco

Procurar **Rute**

Pesquisa binária (continuação)

Procurar
Rute

- 6: Paulo
- 7: Pedro
- **8: Rui ← elemento do meio**
- 9: Rute
- 10: Vasco

Pesquisa binária (continuação)

Procurar **Rute**

- **9: Rute ← elemento do meio**
- 10: Vasco

Encontrado!

Pesquisa binária (implementação recursiva)

- **procedure** procura(lista, elemento):
  - pesquisaBinaria(lista, elemento, 1, tamanho(lista))

- **procedure** pesquisaBinaria(lista, elemento, inicio, fim):
  - **if** (inicio > fim) then: return false
  - meio ← (inicio + fim) / 2
  - **if** (lista[meio] = elemento) **then**: return true
  - **if** (lista[meio] > elemento) **then**:
    return pesquisaBinaria(lista, elemento, inicio, meio-1)
  - **else**:
    return pesquisaBinaria(lista, elemento, meio+1, fim)

Pesquisa binária (continuação)

- 1: Ana
- 2: Bruno
- 3: Carla
- 4: Duarte
- 5: Hugo
- 6: Paulo
- 7: Pedro
- 8: Rui
- 9: **Rute**
- 10: Vasco

- procura(L, Rute)
- pesquisaBinaria(L, Rute, 1, 10)
  - meio = 5
  - L[meio] = Hugo < Rute
- pesquisaBinaria(L, Rute, 6, 10)
  - meio = 8
  - L[meio] = Rui < Rute
- pesquisaBinaria(L, Rute, 9, 10)
  - meio = 9
  - L[meio] = **Rute** = Rute

Pesquisa binária: eficiência

- Quantas "operações" tem a pesquisa binária no pior caso?
- $10 \rightarrow 5 \rightarrow 2 \rightarrow 1$
- $100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$
- $1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 62 \rightarrow 31 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1$

- Pesquisa binária é $\Theta(\log_2 n)$ [ou simplesmente $\Theta(\log n)$]
  - Ex: para uma lista com $2^{20} = 1\,048\,576$ (mais de 1 milhão) nomes, apenas 20 comparações bastavam para verificar um elemento

Revisitar a pesquisa de um elemento

- Será possível fazer ainda melhor? Imaginem que a cada palavra associamos um código numérico.
  - Ex: cada letra é um número: a=0, b=1, c=2, …, tal que, codigo(nome) = (somatório de todas as letras de nome)
    ex: codigo(joana) = (9+14+0+13+0) = 36
- Podemos guardar os nomes numa posição específica e quando procuramos vamos directamente a essa posição

Tabela de *Hash*

#	Conteúdo
0	
1	mario
2	
3	
4	
5	
6	joana
7	
8	nuria
9	

- Função de hash:

  codigo(nome) = soma das letras mod 10

- Exemplo:

  codigo(joana) = (9+14+0+13+0) mod 10 = 36 mod 10 = 6
  codigo(nuria) = (13+20+17+8+0) mod 10 = 8
  codigo(mario) = (12+0+17+8+14) mod 10 = 1

- Procurar(rita):

  codigo(rita) = (17+8+19+0) mod 10 = 4

Tabela de *Hash: eficiência e problemas*

- Se a função de hash for boa, então temos Θ(1) (tempo constante)

- Claro que na realidade podem acontecer problemas:
  - Quando dois nomes têm o mesmo código **(colisão)**. O que fazer?
    - Ex: criar uma lista de listas (uma lista para cada posição da tabela de *hash*)

Tabela de *Hash: Eficiência e problemas*

#	Conteúdo
0	
1	mario
2	
3	
4	
5	
6	joana
7	
8	nuria
9