Análise de Algoritmos: funcionamento correto *(correctness)*

J. Pascoal Faria, R. Rossetti, L. Ferreira
CAL, MIEIC, FEUP

Uma história …

- Funcionamento incorreto do novo sistema de colocação de professores em Portugal em 2004 causou o caos e o atraso de mais de uma semana no início das aulas do Ensino Básico e Secundário
- A empresa inicialmente contratada e que desenvolveu o sistema durante vários meses, não conseguir resolver o problema
- Uma outra empresa, descobriu um algoritmo para efetuar a colocação de acordo com as novas regras, **provou que estava correto**, e resolveu a situação numa semana
- Uma investigadora da UP mostrou depois que se tratava de uma instância do problema dos casamentos estáveis, resolúvel com algoritmos conhecidos (https://www.dcc.fc.up.pt/Pubs/TR05/dcc-2005-02.pdf)

Análise de algoritmos

	Análise estática (teórica)	**Análise dinâmica (experimental)**
Eficiência temporal e espacial	complexidade assintótica	testes de desempenho; *profiling*
Funcionamento correto	**prova ou argumentação sobre correção**	testes pontuais ou aleatórios (*)

(*) "*Testing shows the presence, not the absence of bugs.*" [Dijkstra]

Especificações

- Para provar que um algoritmo resolve corretamente um problema, precisamos de:
  - Especificação rigorosa do problema
  - Descrição rigorosa do algoritmo

- Muitos problemas podem ser especificados por um par:
  - Entradas: Dados de entrada e restrições associadas (**pré-condições**)
  - Saídas: Dados de saída e restrições associadas (**pós-condições**) *

    * Objetivos de maximização/minimização são redutíveis a restrições.

Correção parcial e total

- Correção parcial: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então, se terminar, produz saídas corretas, i.e., que obedecem às pós-condições.

- Correção total: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então termina produzindo saídas que obedecem às pós-condições.

Pré-condições e pós-condições

- double squareRoot(double x)

- Pré-condições?
  - x >= 0 (senão, implementação lança exceção)

- Pós-condições?
  - RESULT * RESULT == x … a menos de um certo erro!
  - RESULT >= 0

- Algoritmo?
  - Método babilónico (derivado de Netwon-Rapshon), …

Pré-condições e pós-condições

- template <typename T>
  int binarySearch(T a[ ], unsigned n, T x)

- Pré-condições?
  - Array a ordenado
  - a != NULL
  - Operadores de comparação definidos para o tipo T

- Pós-condições?
  - (0 <= RESULT < n && a[RESULT] == x) ||
  - (RESULT == -1 && *x não existe em a*)

Pré-condições e pós-condições

- template <typename T>
  void sort(T a[ ], unsigned n)

- Pré-condições?
  - a != NULL
  - Operadores de comparação definidos para o tipo T

- Pós-condições?
  - Array "a" está ordenado, isto é a[0] <= a[1] <= … <= a[n-1]
  - Array final tem os mesmos elementos que o array inicial

Pré-condições e pós-condições

- template <typename T>
  T max(T a[ ], unsigned n)

- Pré-condições?
  - a != NULL
  - n > 0
  - Operadores de comparação definidos para o tipo T

- Pós-condições?
  - RESULT pertence a *a*
  - Nenhum elemento de *a* é maior que RESULT

Invariantes e variantes de ciclos

- A maioria dos algoritmos são iterativos, com um ciclo principal.
- Para provar que um ciclo está correto, temos de encontrar um invariante do ciclo - uma expressão booleana (nas variáveis do ciclo) ‘sempre verdadeira’ ao longo do ciclo, e mostrar que
  - é verdadeira inicialmente, i.e., é implicada pela pré-condição;
  - é mantida em cada iteração, i.e., é verdadeira no fim de cada iteração, assumindo que é verdadeira no início da iteração;
  - quando o ciclo termina, garante (implica) a pós-condição.
- Para provar que um ciclo termina, temos de encontrar um variante do ciclo – uma função (nas variáveis do ciclo)
  - inteira; positiva; estritamente decrescente. (porquê?)

11

Exemplo 1: Insertion Sort

Ideia principal …

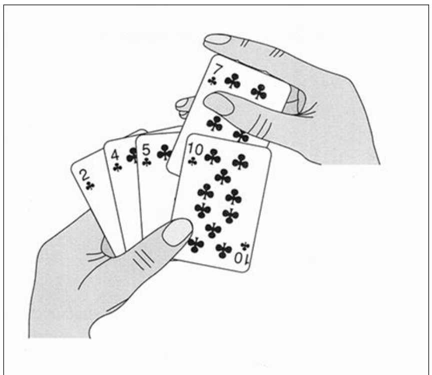
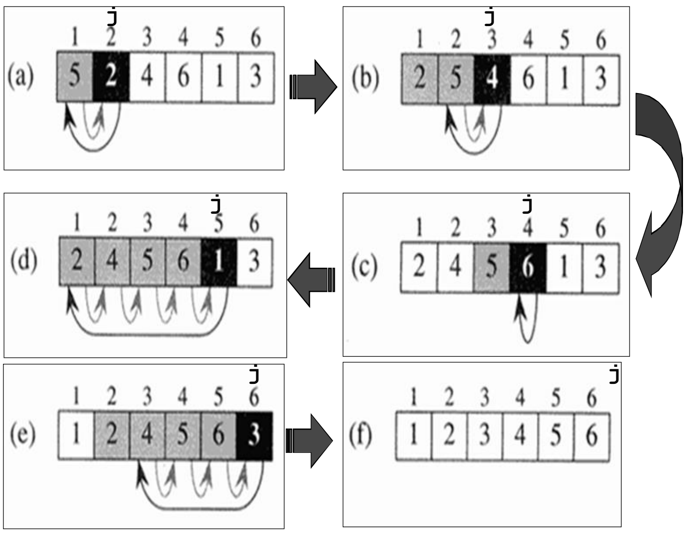


Figure 2.1 Sorting a hand of cards using insertion sort.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

12

Funcionamento



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

13

Algoritmo em pseudo-código

INSERTION-SORT(A, n)
for $j = 2$ **to** n
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

14

Análise (1/2)

- Invariante do ciclo principal [I(j)] ?

A[1, …, j–1] contém os elementos originais, mas ordenados (j = 2, …, n+1)

- ✓ É válido inicialmente (j=2)
  - É óbvio que A[1,…,1] contém os elementos originais, mas ordenados
- ✓ É mantido em cada iteração
  - Assume-se que o invariante se verifica no início da iteração
  - O alg. insere A[j] na posição certa em A[1, …, j] e incrementa j
  - Logo, no fim da iteração (com novo j), verifica-se o invariante
- ✓ No fim do ciclo (j = n+1), garante a pós-condição
  - Invariante refere-se a A[1,…, n-1] ou seja todo o array
  - Logo, implica trivialmente a pós-condição, pois é coincidente

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

Análise (2/2)

- Variante do ciclo principal [v(j)] ?

`n + 1 – j    (j=2, …, n+1)`

  - ✓ Inteiro, pois `n` e `j` são inteiros
  - ✓ Não negativo, pois o valor máximo de `j` é `n`+1
  - ✓ Estritamente decrescente, pois `j` é sempre incrementado

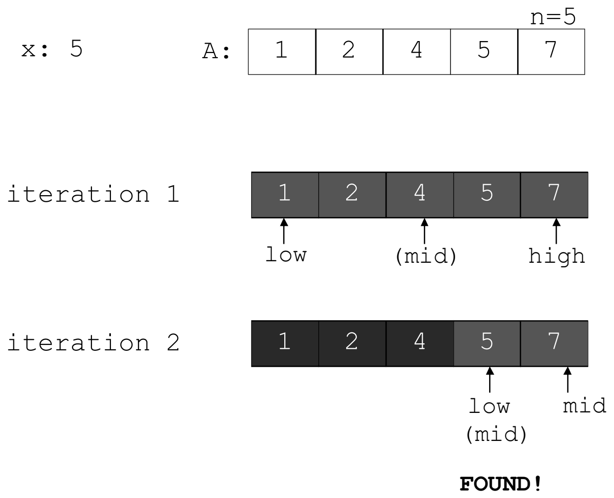
- Logo, o algoritmo está correto e termina (correção total)

Exemplo 2: Binary Search

```
BINARY-SEARCH(A, n, x)
   low ← 1
   high ← n
   while low ≤ high
      mid ← ⌊ (low + high) / 2⌋
      if x = A[mid] then
           return mid
      else if  x < A[mid] then
           high ← mid – 1
      else
            low ←  mid + 1
  return -1
```

17

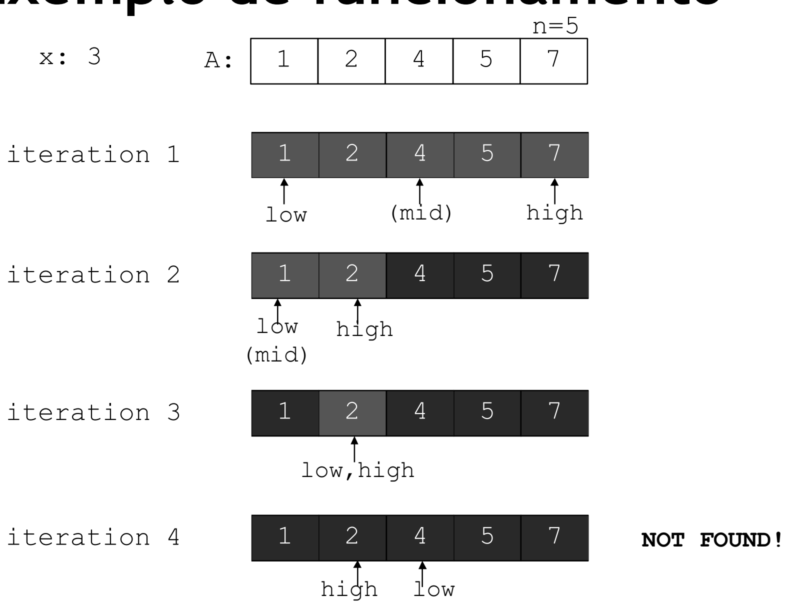
Exemplo de funcionamento



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

18

Exemplo de funcionamento



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

Análise (1/2)

- Invariante do ciclo [`I(low, high)`]?

  `x` só pode existir na área de pesquisa, entre `low` e `high`

  - ✓ É válido inicialmente (`low=1, high = n`), pois a área de pesquisa é todo o array
  - ✓ É mantido em cada iteração
    - ✓ Uma vez que se assume que o array está ordenado …
    - ✓ quando se recua `high`, excluem-se apenas elementos < `x`
    - ✓ quando se avança `low`, excluem-se apenas elementos > `x`
  - ✓ No fim do ciclo, garante a pós-condição
    - ➢ Se o ciclo é interrompido (`A[mid] = x`), garante-se a cláusula em que se encontra `x`
    - ➢ Se o ciclo for até ao fim, a área de pesquisa fica vazia, o que, pelo invariante, implica que `x` não existe em `A`

Análise (2/2)

- Variante do ciclo [`v(low, high)`] ?

  - ➢ Largura da área de pesquisa: **`high – low + 1`**

  - ✓ Inteiro, pois `low` e `high` são inteiros
  - ✓ Não negativo, pois no pior caso é `low = high + 1`
  - ✓ Estritamente decrescente, pois em cada iteração ou aumenta `low` ou diminui `high`

- ❑ Logo, o algoritmo está correto e termina (correção total)

Revisão

- Quais são as 3 qualidades principais a analisar num algoritmo?
- Que técnicas de análise estática e dinâmica existem?
- O que são pré-condições e pós-condições? Para que servem?
- O que é correção parcial e total?
- O que são invariantes e variantes de ciclos? Para que servem?
- Quais são as 3 propriedades que temos de verificar num invariante de ciclo?
- Quais são as 3 propriedades que temos de verificar num variante de ciclo?

Referências

- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
  - Capítulo 2 (Getting Started)