



# Introdução

Algoritmos e Estruturas de Dados  
Inverno 2008

Cátia Vaz

1



## Algoritmos

- Um **algoritmo** é um procedimento que termina sempre.
- Um **procedimento** é um método bem definido e detalhado passo-a-passo para resolver um dado problema.
  - **Exemplo de um problema:** ordenar uma sequência de números por ordem crescente.
    - Input:  $a_1, a_2, \dots, a_n$
    - Output Uma permutação  $(a_1', a_2', \dots, a_n')$  da sequência de input tal que  $a_1' \leq a_2' \leq \dots \leq a_n'$
- um algoritmo diz-se **correcto** se, quando termina, retorna a resposta correcta para cada instância do problema.

Cátia Vaz

2




# Estruturas de Dados

- Uma **estrutura de dados** consiste num modo de armazenar os dados de forma a poderem ser utilizados eficientemente:
  - uma estrutura de dados bem escolhida para um determinado problema vai permitir a utilização de um algoritmo mais eficiente;
  - a escolha da estrutura de dados normalmente começa a partir da escolha da estrutura de dados abstracta;
  - uma estrutura de dados bem desenhada permite a realização de uma variedade de operações críticas, utilizando o menos possível de recursos, tempo de execução e espaço em memória.

Cátia Vaz

3



# Algoritmos e Estruturas de Dados

- **Especificar**
  - Algoritmo e estrutura de dados
- **Análise dos algoritmos**
  - Tempo de execução e memória necessária
- **Implementar**
  - Numa linguagem de programação
- **Testar**
  - Verificar a observância das propriedades para dados de entradas

Cátia Vaz

4

# Algoritmos de Ordenação

- **Objective:** estudar métodos de ordenação de qualquer tipo de dados.
- Para a ordenação de um determinado tipo de dados, é associado a cada elemento uma **chave** que o caracteriza.
- **exemplo de chave:** a ordem natural do tipo de dados
- algoritmos que resolvem o problema da ordenação: **insertion sort**, **selection sort**, **bubble sort** ...

Cátia Vaz

5

## Insertion Sort

- **Ideia do Algoritmo:** considerar os elementos um a um e inseri-los no seu lugar entre os elementos já tratados (mantendo essa ordenação)

```
INSERTION-SORT(A)
1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2    do  $\text{key} \leftarrow A[j]$ 
3       $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > \text{key}$ 
6        do  $A[i+1] \leftarrow A[i]$ 
7           $i \leftarrow i - 1$ 
8       $A[i+1] \leftarrow \text{key}$ 
```

- **Invariante do ciclo:** No início de cada iteração do ciclo externo o sub-array  $A[1:j-1]$  consiste nos elementos originalmente em  $A[1:j-1]$ , mas ordenados.

Cátia Vaz

6

## *Invariante de ciclo e correcção do Insertion Sort*

- Para verificar que um algoritmo é **correcto** é necessário (mas não é suficiente) que o invariante de ciclo:
  - **Inicialização**: é verdadeiro antes da primeira iteração do ciclo.
  - **Continuação**: se é verdadeiro antes de uma iteração de ciclo, permanece verdadeiro antes da iteração seguinte.
  - **Terminação**: Quando o ciclo termina, o invariante dá-nos uma propriedade útil que ajuda a mostrar que o **algoritmo é correcto**.

Cátia Vaz

7

## *Insertion Sort: Implementação em Java*

```
static void insertionSort(int[] a, int left, int right){  
    int v;  
    for(int i = left+1; i <=right; i++) {  
        v = a[i];  
        j = i;  
        while (j > left && less(v, a[j-1]))  
            a[j] = a[j-1]; j--;  
        a[j] = v;  
    }  
}
```

- As operações abstractas nos dados a utilizar são: a comparação (less); a troca (exch).

```
static void exch(int[] a, int i, int j) {  
    int t = a[i]; a[i] = a[j]; a[j] = t;  
}  
static boolean less(int v, int w) { return v < w;}
```

Cátia Vaz

8



# Ordenação dos dados

- Para a ordenação de um determinado tipo de dados, é associado a cada elemento uma **chave** que o caracteriza. Podemos ordenar objectos:
  - consoante a sua ordem natural;
  - consoante uma outra ordem.
- **Interface Comparable**
  - A classe que descreve o tipo de dados tem que implementar a interface Comparable para especificar uma ordem natural:
    - o método compareTo especifica uma **ordem total**;
    - exemplo de tipos comparáveis: String, Integer e Double
    - para que os objectos de uma classe definida pelo utilizador sejam comparáveis, é necessário que a classe implemente a interface Comparable.

Cátia Vaz

9



# Ordenação dos dados

- **Interface Comparator.**
  - A classe que **implementar esta interface** tem que **implementar o método compare** tal que compare(v, w) especifique uma ordem total.
  - Vantagem: separa a definição do tipo de dados da forma de comparação de dois objectos daquele tipo.
    - adiciona uma nova ordem a um tipo de dados.
    - adiciona uma ordem a um tipo de dados de uma biblioteca que não tenha ordem natural.

Cátia Vaz

10

# Exemplo de alterações para utilizar um *Comparator*

```
static boolean less(Comparator c, Item v, Item w){
    return c.compare(v, w) < 0;}

static void exch(Item[] a, int i, int j) {
    Item t = a[i]; a[i] = a[j]; a[j] = t;}

static void insertionSort(Item[] a,int left,int right,Comparator c){
    Item v;
    for(int i = left+1; i <=right; i++) {
        v = a[i];
        j = i;
        while (j > left && less(c, v, a[j-1])) {
            a[j] = a[j-1]; j--;
        }
        a[j] = v;
    }
}
```

Cátia Vaz

11

## *Selection Sort*

- Ideia do Algoritmo:
  - procurar o menor elemento e trocar com o elemento na primeira posição
  - procurar o segundo menor elemento e trocar com o elemento na segunda posição
  - proceder assim até a ordenação estar completa

```
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]
```

Cátia Vaz

12



# *Selection Sort :*

## *Implementação em Java*

```
static void selectionSort(int[] a, int left, int right){  
    int min;  
    for(int i= left; i<right; i++){  
        min=i;  
        for(int j=i+1; j<=right; j++)  
            if( less(a[j], a[min]) ) min=j;  
        exch(a,i,min);  
    }  
}
```

**Cátia Vaz**

13



# *Bubble Sort*

Ideia do Algoritmo :

- em cada iteração "i" começa-se sempre com o elemento mais à direita e troca-se com o elemento à sua esquerda sempre que esse for maior, caso contrário continua-se com o que estava direita. Continua-se o processo até chegar à posição "i".

```
BUBBLESORT(A)  
1  for i ← 1 to length[A]  
2      do for j ← length[A] downto i + 1  
3          do if A[j] < A[j - 1]  
4              then exchange A[j] ↔ A[j - 1]
```

**Cátia Vaz**

14



## Exercício

- Desenvolver uma implementação em Java do algoritmo *Bubble Sort*.



Cátia Vaz

15



## Merge Sort

- Ideia do Algoritmo :

- ordenar as duas metades dos dados independentemente (de forma recursiva)
- Combinar as duas partes depois de ordenadas, de forma a que os dados fiquem totalmente ordenados.
  - criar dois *arrays* com cada uma das partes ordenadas
  - Juntar os dois arrays de forma a que o total fique ordenado
- Algoritmo do tipo ***dividir para conquistar***

```
MERGE-SORT(A, p, r)  
  if p < r  
    then q ← ⌊(p + r)/2⌋  
          MERGE-SORT(A, p, q)  
          MERGE-SORT(A, q + 1, r)  
          MERGE(A, p, q, r)  
          ▷ Check for base case  
          ▷ Divide  
          ▷ Conquer  
          ▷ Conquer  
          ▷ Combine
```

*Initial call:* MERGE-SORT(*A*, 1, *n*)

Cátia Vaz

16



# Merge Sort:

## Implementação em Java

```
static void mergesort(int [] a, int left, int right){
    if (left < right) {
        int mid = (left+right)/2;
        mergesort(a, left, mid);
        mergesort(a, mid+1, right);
        merge(a, left, mid, right);
    }
}

static void mergesort(int [] a){
    mergesort(a, 0, a.length-1);
}
```

Cátia Vaz

17

# Merge:

## Implementação em Java

```
MERGE(A, p, q, r)
    n1 ← q - p + 1
    n2 ← r - q
    create arrays L[1..n1 + 1] and R[1..n2 + 1]
    for i ← 1 to n1
        do L[i] ← A[p + i - 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                i ← i + 1
            else A[k] ← R[j]
                j ← j + 1
```

p=left

q=right

Cátia Vaz

18

# Binary Search

## Algoritmo:

- Se o domínio for vazio não existe elemento
  - Se o elemento for igual ao do meio foi encontrado
  - Se o elemento for menor que o do meio procurar desde o início até ao meio menos 1
  - Se o elemento for maior que o do meio procurar desde o meio mais 1 até ao fim
- **Versão Iterativa:**

```
/*Procura Binária Iterativa*/
public static int pBIterativa(int num,int[] array,int first,int last){
    while (last >= first){
        int medio = (last+first)/2 ;
        if (num == a[medio]) return medio;
        if (num <a[medio]) last = medio-1;
        else first = medio + 1;
    }
    return -1 ;
}
```

**Cátia Vaz**

19

# Binary Search

## Versão Recursiva:

```
/*Procura binária recursiva*/
public static int pBRecursiva(int num,int[] array,int first,int last){
    int mid;
    if(first>last) return -1;
    else{
        mid=(first+last)/2;
        if(num==array[mid]) return mid;
        else{
            if(num<array[mid])
                return pBRecursiva(num,array,first,mid-1);
            else
                return pBRecursiva(num,array,mid+1,last);
        }
    }
}
```

**Cátia Vaz**

20



# Definições

- Tipos de algoritmos de ordenação:
  - **não adaptativos** - sequência de operações independente da ordenação original dos dados;
  - **adaptativos** - sequência de operações dependente do resultado de comparações.
- Um algoritmo de ordenação é dito **estável** se preserva a ordem relativa dos elementos com chaves repetidas
- Um algoritmo de ordenação é dito **interno**, se o conjunto de todos os dados a ordenar couber na memória; caso contrário é dito **externo**.

Cátia Vaz

21



# Definições

- Um algoritmo de ordenação é dito **directo** se os dados são acedidos directamente nas operações de comparação e troca; caso contrário é dito **indirecto**.

Cátia Vaz

22