# Aula 09

## Ordenação e Complexidade Algorítmica

*Programação II, 2016-2017*

*v1.5, 18-04-2017*

DETI, Universidade de Aveiro

09.1

## Conteúdo



09.2

# 1 Complexidade Algorítmica: Introdução

### Ordenação

- O acto de se colocar os elementos de uma sequência de informações (dados) numa ordem predefinida:

| **3** | **9** | **6** | **7** | **5** | **1** | **7** | **2** | **8** | **3** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

| **1** | **2** | **3** | **3** | **5** | **6** | **7** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Para que uma sequência de dados possa ser ordenada, os seus elementos têm de estabelecer uma relação de ordem entre si;
- Essa relação de ordem pode ser:
  - numérica, se forem números;
  - lexicográfica, se foram palavras;
  - cronológica, se forem datas;
  - . . . .
- Independentemente do tipo de elementos, a ordenação pode ser crescente ou decrescente.

09.3

## 1.1 Ordenação

### Algoritmos de Ordenação

- Sequencial;
- Tipo "bolha" (*BubbleSort*);
- Inserção (*InsertionSort*);
- Fusão (*MergeSort*);
- Rápida *QuickSort*;
- ...

Se é um facto que qualquer algoritmo de ordenação correctamente implementado tem exactamente o mesmo resultado: *um vector (array) ordenado*; porquê então tantos algoritmos de ordenação?

A resposta a esta questão prende-se com a eficiência na utilização de dois aspectos essenciais na execução de programas: *Tempo de execução* e *Espaço de memória utilizado*.

É precisamente para abordar estes problemas que se estuda a chamada *Complexidade Algorítmica*
09.4 dos programas[1].

### Complexidade Algorítmica

- Abordagem para medir o desempenho de diferentes algoritmos/estruturas de dados em dois aspectos essenciais:
  1. Tempo de execução
  2. Espaço de memória utilizado
- Tentativas para medir com exactidão estas duas facetas estão votadas ao fracasso (para além de casos muito particulares)
- Assim, faz-se uma aproximação ao problema identificando os parâmetros mais determinantes
  - No caso da ordenação de um vector (array), será a apenas a dimensão do vector
- Temos assim uma aproximação à ordem de magnitude dos recursos consumidos em função da dimensão do problema

09.5

## 1.2 Notação *Big-O*

**Notação *Big-O***: Diz-se que uma função $f(n)$ (representando a métrica em análise) tem uma complexidade $O(g(n))$ se, para valores de $n$ tão grandes quanto necessário, se verifica a equação: $f(n) < K \cdot g(n)$, para uma certa constante $K$.

- Temos assim que:
  - Factores multiplicativos constantes não são relevantes.
    - Exemplos: $O(100000 \cdot n) \approx O(n)$ ; $O(100000) \approx O(1)$
  - Parcelas constantes também não contam.
    - Exemplo: $O(100000 + n^2) \approx O(n^2)$
  - Uma função de complexidade $g(n)$ também é de $h(n)$ se $h(n)$ for majorante de $g(n)$.
    - Exemplos: $O(n^2 + n^3) \approx O(n^3)$; $O(n)$ é também $O(n^3)$
- Estamos, é claro, interessados em descobrir a menor função majorante possível!

09.6

[1]Como se verá, não é a mesma coisa do que a complexidade do código fonte, pelo que estes dois aspectos não devem ser confundidos.

- Classes mais comuns (ordem crescente de complexidade):
  - Constante: $O(1)$
  - Logarítmica: $O(log(n))$
  - Linear: $O(n)$
  - Pseudo-linear: $O(n \cdot log(n))$
  - Quadrática: $O(n^2)$
  - Cúbica: $O(n^3)$
  - ( Polinomial: $O(n^p)$ )
  - Exponencial: $O(p^n)$
  - Factorial: $O(n!)$
- Faz sentido fazer esta análise tendo em consideração a complexidade *média* ou a complexidade *máxima* (a complexidade mínima não é, em geral, tão útil)

09.7

# 2 Ordenação

## 2.1 Sequencial

### Ordenação por Selecção

- A ordenação por selecção consiste em percorrer (por ordem) todos os índices do vector, procurando e colocando o valor mínimo encontrado nessa posição.

```
void selectionSort(int[] a, int start, int end)
{
  assert validSubarray(a, start, end);

  for(int i = start; i < end-1; i++)
  {
    int indexMin = searchMinimum(a, i+1, end); // minimum in [i+1;end[
    if (a[i] > a[indexMin])
      swap(a, i, indexMin); // swaps values a[i] and a[indexMin]
  }

  assert isSorted(a, start, end);
}
```

09.8

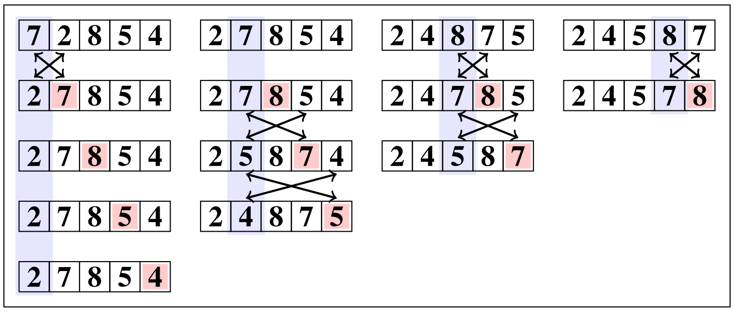
### Ordenação Sequencial

- O ordenação sequencial é um caso particular da ordenação por selecção, mas em que se junta a procura do mínimo e a respectiva troca (tornando o algoritmo um pouco mais simples à custa de mais trocas).

```
void sequentialSort(int[] a, int start, int end)
{
  assert validSubarray(a, start, end);

  for(int i = start; i < end-1; i++)
    for(int j = i+1; j < end; j++)
      if (a[i] > a[j])
        swap(a, i, j); // swaps values a[i] and a[j]

  assert isSorted(a, start, end);
}
```

09.9

### Ordenação Sequencial: Complexidade



- Para um vector de dimensão $n$ é necessário fazer $(n-1)+(n-2)+\cdots 1$ comparações, ou seja complexidade $O(n^2)$;
- O número de trocas (no pior caso) terá também a mesma complexidade.

09.10

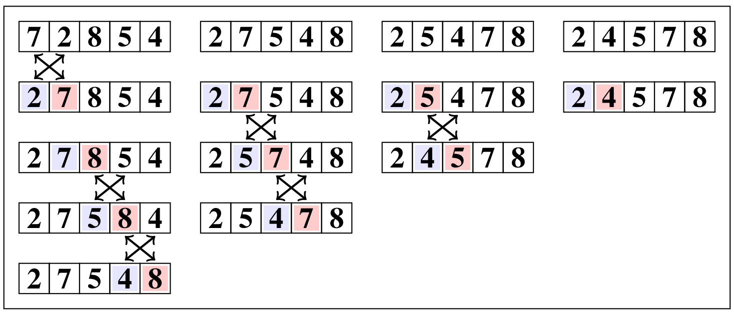
## 2.2 Bolha

- A ordenação tipo "bolha" consiste em percorrer (por ordem) todos os índices do vector, comparando e trocando os pares de valores consecutivos sempre que não estiverem na ordem certa.
- Sempre que tiver havido pelo menos uma troca o procedimento é repetido (quando não houver lugar a trocas então , por definição, o vector está ordenado).
- O algoritmo designa-se por "bolha" porque têm a propriedade de em cada iteração os maiores valores (ordem crescente) irem sendo"empurrados" para o fim do vector.

```
void bubbleSort(int[] a, int start, int end) {
  assert validSubarray(a, start, end);

  boolean swapExists;
  int f = end-1;
  do {
    swapExists = false;
    for(int i = start;i < f;i++)
      if (a[i] > a[i+1]) {
        swap(a, i, i+1);
        swapExists = true;
      }
    f--;
  }
  while(swapExists);

  assert isSorted(a, start, end);
}
```

09.11

### Ordenação "Bolha": Complexidade



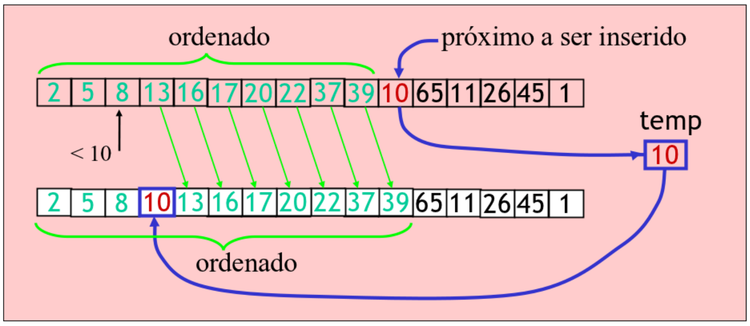
- Para um vector de dimensão $n$ é necessário fazer $(n-1)+(n-2)+\cdots 1$ comparações, ou seja complexidade $O(n^2)$;

- O número de trocas (no pior caso) terá também a mesma complexidade.
- O pior caso ocorre quando o vector está ordenado pela ordem inversa.
- O melhor caso ocorre quando o vector já está ordenado ($O(n)$ comparações). 09.12

## 2.3 Inserção

É um método simples de inserção assente na partição do vector em duas partes: uma ordenada e outra por ordenar.

- Existem duas partes no vector:
  - ordenada (vai aumentar)
  - não-ordenada (vai diminuir)
- Ordena através da inserção no segmento ordenado (na sua posição correcta) de um elemento retirado da parte não ordenada;
- Inicialmente, o segmento ordenado contém apenas o primeiro elemento do vector. 09.13




1. Pega no próximo elemento (não ordenado) a ser inserido;
2. Vai comparar este elemento com cada um dos elementos da parte já ordenada até encontrar um elemento que seja maior (menor -> pesq. fim) ;
3. Desloca para a direita os restantes elementos do vector ordenado (i.e. todos os elementos maiores que o elemento a inserir);
4. Insere o elemento pretendido. 09.14

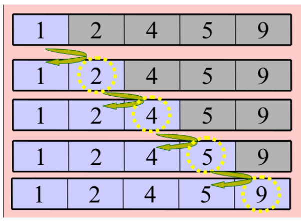
### Ordenação por Inserção: Implementação

```
void insertionSort(int[] a, int start, int end)
{
  assert validSubarray(a, start, end);

  for(int i = start+1;i < end;i++)
  {
    int j;
    int v = a[i];
    for(j = i-1;j >= start && a[j] > v;j--)
      a[j+1] = a[j];
    a[j+1] = v;
  }

  assert isSorted(a, start, end);
}
```

- Uma vantagem deste algoritmo reside no facto de a procura ser sempre feita num subvector ordenado;
- Podemos reduzir ainda mais a complexidade aplicando o método da procura binária (TPC). 09.15

### InsertionSort - Complexidade

- *Pior caso*: vector ordenado ao contrário
  - N.º de Comparações: $1+2+\cdots+(n-2)+(n-1) => O(n^2)$
- *Melhor caso*: vector já ordenado
  - N.º de Comparações: $(n-1) => O(n)$




09.16

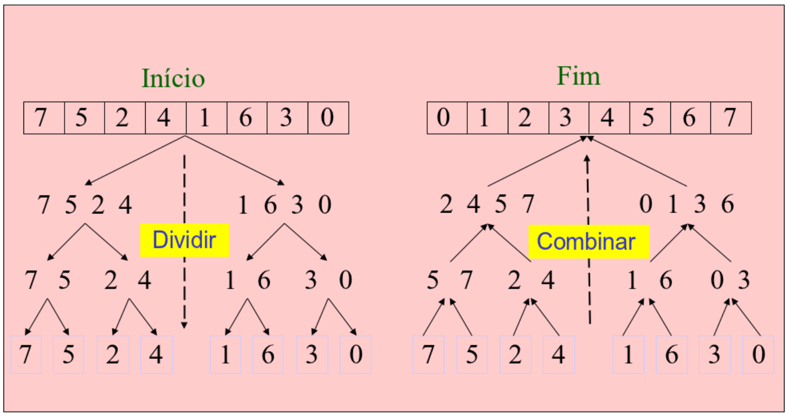
## 2.4 Fusão

### Fusão - Merge

- *MergeSort*
  - Um algoritmo eficiente.
- Características:
  - Recursivo;
  - "Dividir para Conquistar";
  - Divide um vector de $n$ elementos em duas partes de tamanho $n/2$;
  - Ordenar cada vector chamando o *Merge Sort* recursivamente;
  - No final: combinar as sub-vectores ordenados formando uma única lista ordenada;
  - Caso limite: vector com um elemento.

09.17

### Fusão: Merge Sort




09.18

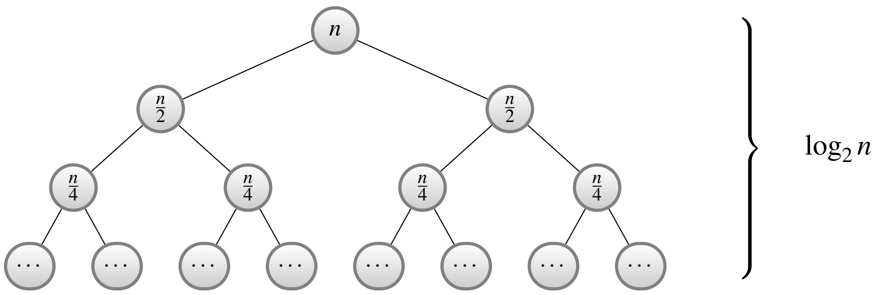
## Fusão: Implementação

```
static void mergeSort(int[] a, int start, int end) {
  assert validSubarray(a, start, end);
  if (end - start > 1) {
    int middle = (end + start) / 2;
    mergeSort(a, start, middle);
    mergeSort(a, middle, end);
    mergeSubarrays(a, start, middle, end);
  }
  assert isSorted(a, start, end);
}

static void mergeSubarrays(int[] a, int start, int middle, int end) {
  int[] b = new int[end-start];
  int i1 = start;
  int i2 = middle;
  int j = 0;
  while(i1 < middle && i2 < end) {
    if (a[i1] < a[i2])
      b[j++] = a[i1++];
    else
      b[j++] = a[i2++];
  }
  while(i1 < middle)
    b[j++] = a[i1++];
  while(i2 < end)
    b[j++] = a[i2++];
  arraycopy(b, 0, a, start, end-start);
}
```

09.19

## Merge - Complexidade

- *Melhor Caso*, *Caso Médio* e *Pior Caso*: $O(n \cdot log(n))$




09.20

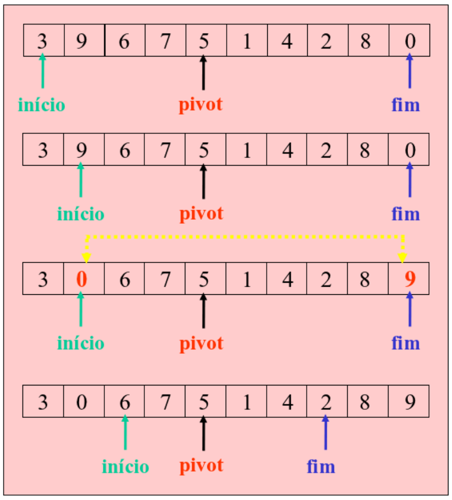
# 2.5 *Quick Sort*

## QuickSort

- Algoritmo de Ordenação Rápida;
- Características:
  - Recursivo;
  - "Dividir para Conquistar";
  - Tal como o *Merge Sort*, divide o vector em duas partes e "ataca" cada um dos sub-vectores de forma recursiva;
  - Mas neste caso:
    - Define um elemento de referência no vector (*pivot*);
    - Posiciona à esquerda do *pivot* os elementos inferiores;
    - Posiciona à direita do *pivot* os elementos superiores.

09.21

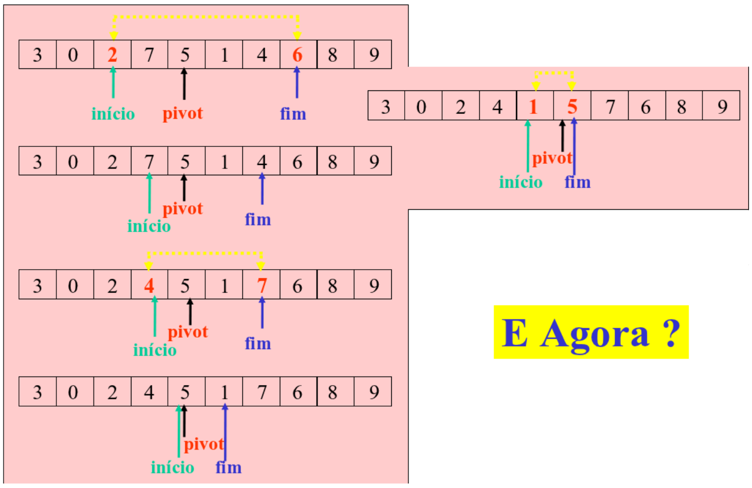
## QuickSort




1. Escolher o pivot;
2. Movimentar o “inicio” até encontrar um elemento maior que o pivot;
3. Movimentar o “fim” até encontrar um elemento menor que o pivot;
4. Trocar o elemento encontrado no ponto 2 com o elemento encontrado no ponto 3;
5. Recomeçar o processo (i.e. voltar ao ponto 2) até que: “inicio” > “fim”

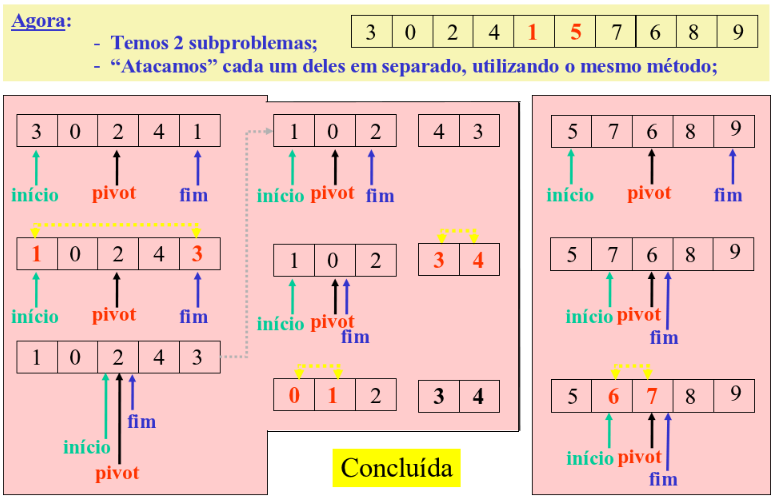
09.22

## QuickSort




09.23

## QuickSort




09.24

### QuickSort: Implementação

```
static void quickSort(int[] a, int start, int end) {
  assert validSubarray(a, start, end);
  int n = end-start;
  if (n < 2) // should be higher (10)!
    sequentialSort(a, start, end);
  else {
    int posPivot = partition(a, start, end);
    quickSort(a, start, posPivot);
    if (posPivot+1 < end)
      quickSort(a, posPivot+1, end);
  }
  assert isSorted(a, start, end);
}

static int partition(int[] a, int start, int end) {
  int pivot = a[end-1];
  int i1 = start-1;
  int i2 = end-1;
  while(i1 < i2) {
    do
      i1++;
    while(a[i1] < pivot);
    do
      i2--;
    while(i2 > start && a[i2] > pivot);
    if (i1 < i2)
      swap(a, i1, i2);
  }
  swap(a, i1, end-1);
  return i1;
}
```

09.25

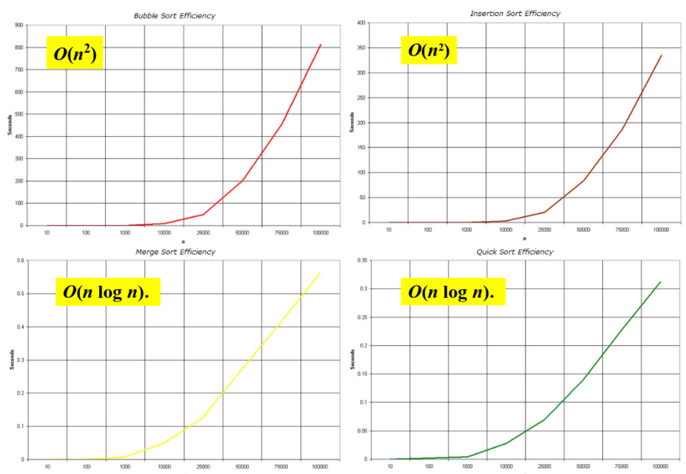
### QuickSort: Complexidade

- Algoritmo muito eficiente;
- *Caso Médio*: $O(n \cdot log(n))$
- *Melhor Caso*: o pivot escolhido representar um valor mediado do conjunto de elementos;
- *Pior Caso*: o pivot escolhido, por exemplo, representar o valor máximo do conjunto de elementos: $O(n^2)$

09.26

## 2.6 Complexidade: comparação

### Complexidade: Gráficos Comparativos




09.27

## Complexidade: Conclusões

- Com um número relativamente baixo de elementos, o desempenho dos diferentes algoritmos não se distingue muito bem;
- Quando o número de elementos é pequeno ($n < 50$) convém escolher o *Bubble* ou o *Insertion* que são muito rápidos devido à sua simplicidade;
- Quando o número de elementos aumenta, o *QuickSort* é aquele que apresenta melhor desempenho (médio) logo seguido do *MergeSort*[2].

09.28

[2] Dos algoritmos de ordenação apresentados!