Aula 09

Ordenação e Complexidade Algorítmica

Programação II, 2018-2019

v1.6, 2018-04-14

DETI, Universidade de Aveiro 09.1

Conteúdo

 09.2

1 Complexidade Algorítmica: Introdução

1.1 Motivação

Motivação

- *Ordenação* é o acto de colocar os elementos de uma sequência de dados numa ordem predefinida:

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

$\Downarrow$

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, é preciso haver uma relação de ordem definida entre os seus elementos.
- Essa relação de ordem pode ser:
  - numérica, se forem números;
  - lexicográfica, se forem palavras;
  - cronológica, se forem datas.

  - ....
- 09.3 Independentemente do tipo de elementos, a ordenação pode ser crescente ou decrescente.

Algoritmos de Ordenação

- Ordenação por Seleção (*SelectionSort*);
- Ordenação por flutuação ou tipo "bolha" (*BubbleSort*);
- Ordenação por Inserção (*InsertionSort*);
- Ordenação por Fusão (*MergeSort*);
- Ordenação Rápida *QuickSort*;
- ...

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

A resposta a esta questão prende-se com a eficiência na utilização de dois recursos essenciais na execução de programas: *tempo de execução* e *espaço de memória utilizado*.

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

1.2 Complexidade Algorítmica: definição

Complexidade Algorítmica: definição

Complexidade (computacional) de um algoritmo: É uma medida da *quantidade de recursos* computacionais necessários para executar esse algoritmo.

- Os recursos mais importantes a considerar são:
  1. Tempo de execução.
  2. Espaço de memória utilizado.
- Normalmente, a quantidade de recursos necessários para um algoritmo resolver um certo problema depende da *dimensão* do problema.
- Por isso, a complexidade de um algoritmo é uma *função da dimensão* do problema.
  - Por exemplo, o tempo para ordenar um vector depende da dimensão do vector.
- A complexidade também pode depender dos *dados* concretos do problema, mas é vulgar considerarmos apenas a complexidade *média* ou do *pior caso* para certa dimensão dos dados.
  - Por exemplo, alguns algoritmos de ordenação são mais rápidos se os dados já estiverem
09.5 ordenados.

Complexidade Algorítmica: dificuldades

Medir a complexidade apresenta alguns desafios.

- Computadores diferentes demoram tempos diferentes para executar as mesmas instruções e podem usar quantidades de memória diferentes para guardar os mesmos dados.
- Um mesmo programa, executado várias vezes no mesmo computador, pode demorar tempos diferentes, devido a fatores imprevisíveis como interrupções de hardware ou competição com outros processos no sistema.
- Assim, para medir a complexidade de um algoritmo sem depender de uma implementação concreta num certo sistema, é vulgar expressar os recursos necessários em unidades mais abstratas como o número de instruções executadas e o número de posições de memória ocupadas.
- Esses números, multiplicados por fatores adequados a um certo sistema, dão uma estimativa do
09.6 tempo (em segundos) e memória (em bytes) gastos nesse sistema concreto.

[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.

1.3 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 suficientemente grandes, se verifica a desigualdade: $f(n) < K \cdot g(n)$, para uma certa constante K.

- Temos assim que:
  - Factores multiplicativos constantes não são relevantes.
    * Exemplos: $O(100000 \cdot n) = O(n)$; $O(100000) = O(1)$
  - Só interessa a parcela que cresce "mais depressa".
    * Exemplos: $O(100000 + n^2) = O(n^2)$; $O(n^2 + n^3) = O(n^3)$
  - Uma função com complexidade $O(g(n))$ também tem complexidade $O(h(n))$ se $h(n)$ for majorante de $g(n)$.
    * Exemplo: $f \in O(n) \implies f \in O(n^3)$
- Estamos, é claro, interessados em descobrir a menor função majorante possível!

09.7

- 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).

09.8

2 Ordenação

Nesta secção vamos descrever vários algoritmos de ordenação e apresentar funções que os implementam em Java. Todas essas funções seguem o protótipo `someSort`(a, s, e), onde `someSort` é o nome do algoritmo e farão a ordenação *in-place* dos elementos do array a com índices $i \in [s, e[$. Naturalmente, exige-se que $0 \le s \le e \le$ `a.length`. Essa pré-condição é testada pela função `validSubarray(a, s, e)`.

2.1 Ordenação por Seleção

Ordenação por Seleção

A ordenação por seleção consiste em:

- Procurar o valor mínimo no vector e colocá-lo na primeira posição.
- Depois repetir o processo a partir de cada uma das posições seguintes, por ordem.

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

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

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

09.9

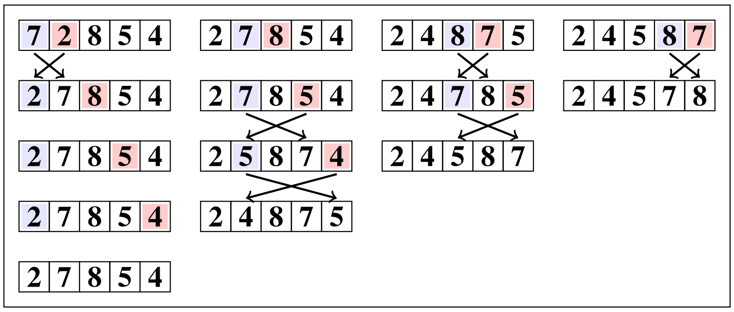
Ordenação Sequencial

- A ordenação sequencial é uma variante da ordenação por seleção, 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).

```java
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);
}
```

09.10

Ordenação Sequencial: Complexidade



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

09.11

2.2 Ordenação por Flutuação (Bolha)

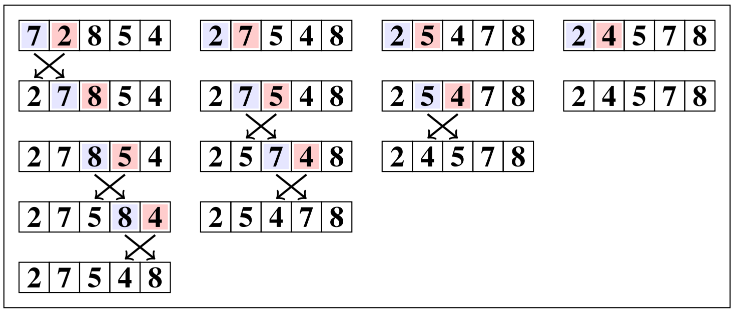
A ordenação tipo “bolha” consiste em:

- Comparar todos os pares de elementos consecutivos e trocá-los se não estiverem na ordem certa.
- No fim dessa passagem, se tiver havido pelo menos uma troca, repete-se o procedimento. Quando não houver trocas, o vector está ordenado e o algoritmo termina.
- O algoritmo designa-se por “bolha” porque o maior valor encontrado numa passagem vai “subindo” até ao topo do vector como uma bolha de ar numa coluna de água.

```java
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);
}
```

09.12

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. Nesse caso bastam $n-1$ comparações (complexidade $O(n)$).

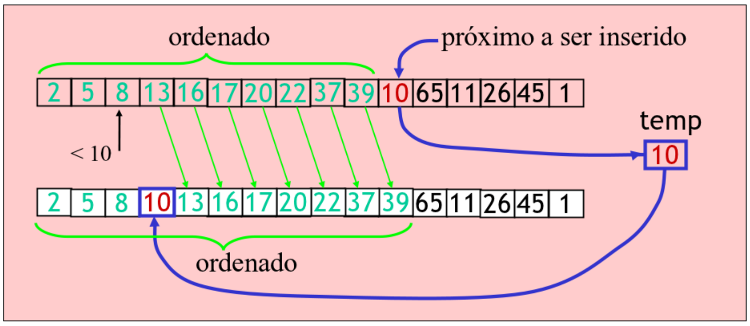
09.13

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 posição correcta) de um elemento retirado da parte não ordenada;
- Inicialmente, o segmento ordenado contém apenas o primeiro elemento.

09.14



1. “Retira” o primeiro elemento do segmento não ordenado.
2. Compara este elemento com os elementos da parte já ordenada até encontrar a posição que lhe cabe.
3. Desloca os elementos do vector ordenado para a direita dessa posição.
4. Insere o elemento na posição pretendida.

09.15

Ordenação por Inserção: Implementação

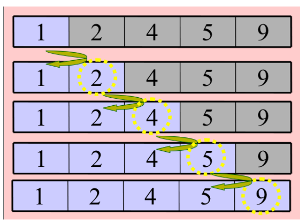
```
void insertionSort(int[] a, int start, int end) {
  assert validSubarray(a, start, end);

  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 resulta de a procura ser sempre feita num subvector ordenado;
- Podemos reduzir ainda mais a complexidade aplicando o método da procura binária (TPC).

09.16

InsertionSort - Complexidade

- *Pior caso*: quando o vector original está por ordem inversa.
  - N.º de Comparações: $1+2+\cdots+(n-2)+(n-1) \in O(n^2)$
- *Melhor caso*: quando o vector original já está na ordem certa.
  - N.º de Comparações: $(n-1) \in O(n)$



09.17

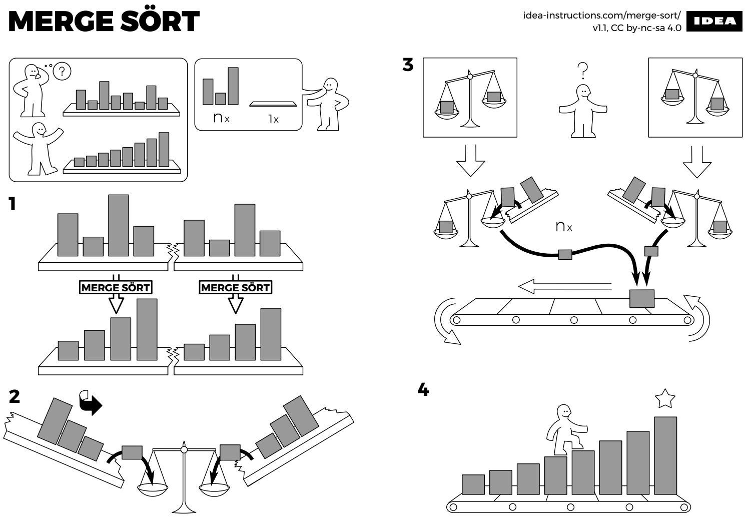
2.4 Fusão

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 ou menos.

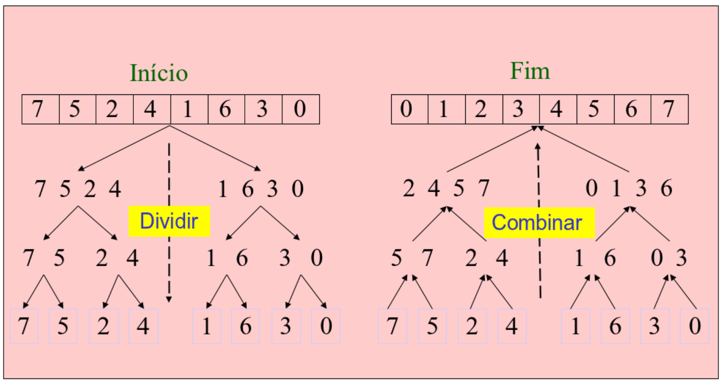
09.18

Fusão: Merge Sort



09.19

Fusão: Merge Sort



09.20

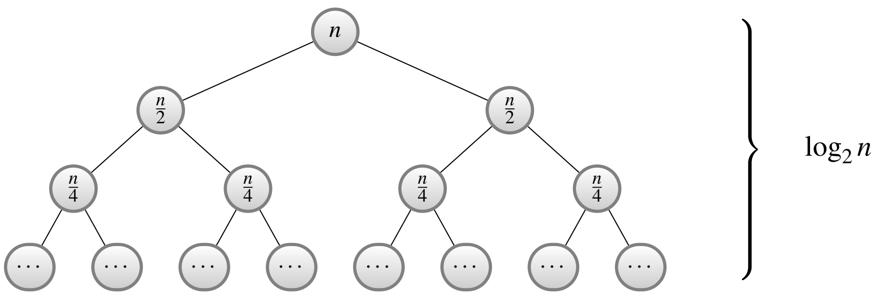
Fusão: Implementação

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

09.21

Merge - Complexidade

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



09.22

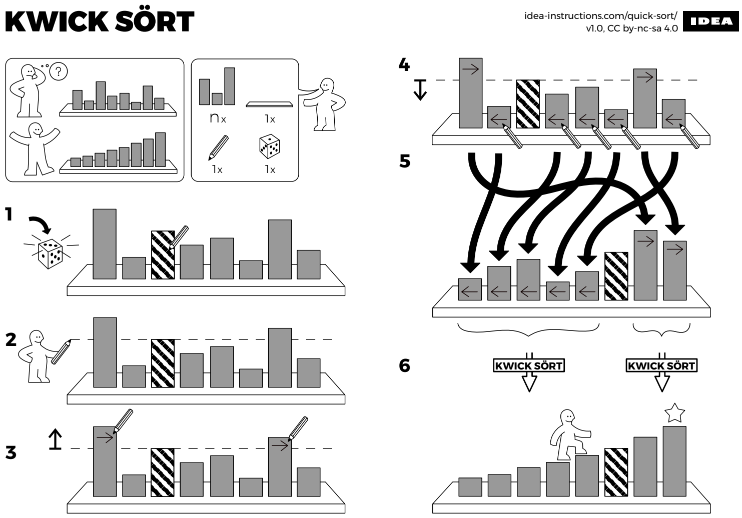
2.5 *Quick Sort*

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:
    * Seleciona um elemento de referência no vector (*pivot*);
    * Posiciona à esquerda do *pivot* os elementos inferiores;
    * Posiciona à direita do *pivot* os elementos superiores.

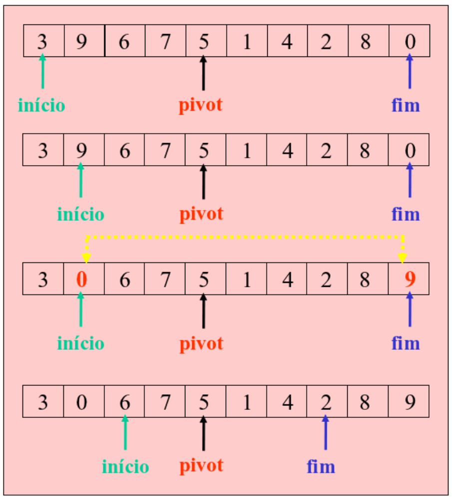
09.23

QuickSort



09.24

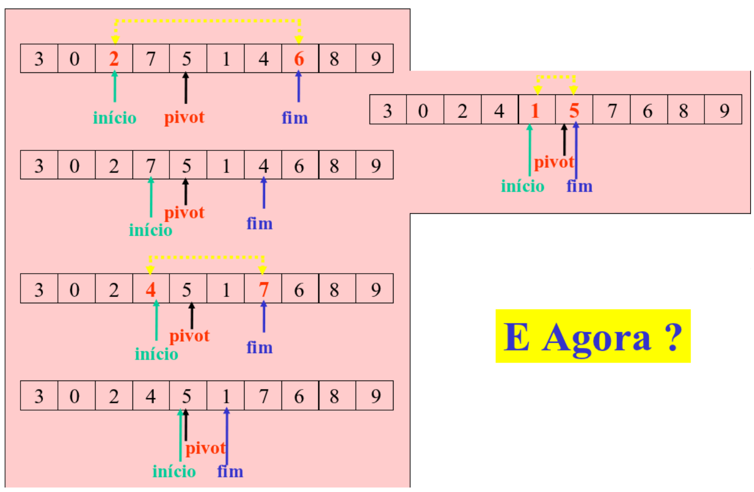
QuickSort



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”

09.25

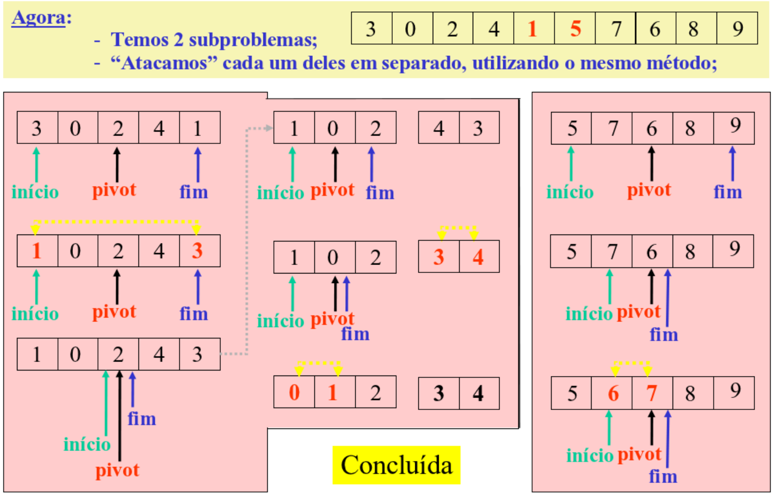
QuickSort



E Agora ?

09.26

QuickSort



09.27

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;
}
```

09.28

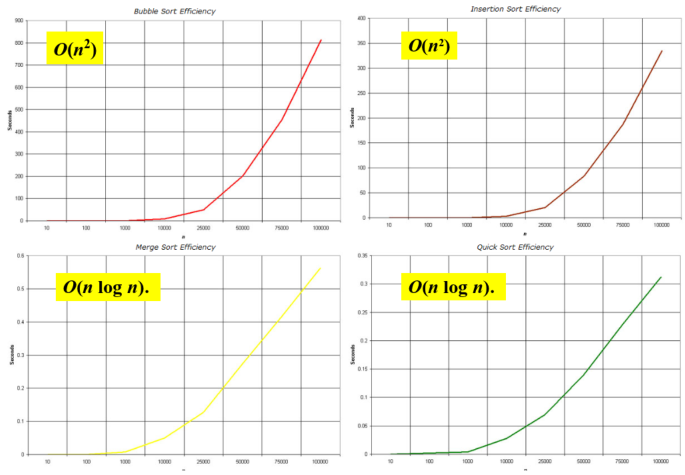
QuickSort: Complexidade

- Algoritmo muito eficiente;
- *Melhor Caso*: quando o pivot escolhido em cada invocação for um valor mediano do conjunto de elementos: $O(n \cdot log(n))$;
- *Pior Caso*: quando o pivot escolhido em cada invocação for um valor extremo do conjunto de elementos: $O(n^2)$
- *Caso Médio*: em casos normais os pivots ‘caiem’ entre a mediana e os extremos, mas mesmo assim o tempo é da ordem de $O(n \cdot log(n))$

09.29

2.6 Complexidade: comparação

Complexidade: Gráficos Comparativos



09.30

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 ($n < 50$), o *InsertionSort* é uma boa opção, porque é muito rápido e simples;
- Quando o número de elementos aumenta, o *QuickSort* é aquele que apresenta melhor desempenho (médio) logo seguido do *MergeSort*.[2]

09.31

[2] Dos algoritmos de ordenação apresentados!