

Aula 09

Ordenação e Complexidade Algorítmica

Programação II, 2020-2021

2021-05-10

Complexidade Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

1 Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica: definição

Notação *Big-O*

2 Ordenação

Ordenação por Seleção

Ordenação por Flutuação (Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade: comparação

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

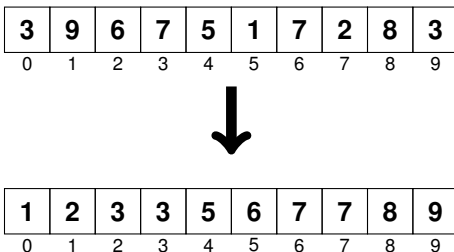
Ordenação por Fusão

Quick Sort

Complexidade:
comparação

Motivação

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



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

A ordenação pode ser crescente ou decrescente.

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

Porquê tantos algoritmos de ordenação?

Complexidade Algorítmica!

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 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 ordenados.

Complexidade Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

Complexidade Algorítmica: dificuldades

- 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.
- Estas diferenças devem-se exclusivamente a aspetos concretos dos sistemas de execução dos programas e não refletem qualquer variabilidade intrínseca aos algoritmos em si.
- Assim, para medir a complexidade de um algoritmo sem depender da sua 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 tempo (em segundos) e da memória (em bytes) necessários nesse

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!

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

- 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).

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.

```
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);  
}
```

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

- 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).

```
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);  
}
```

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

Ordenação Sequencial: Complexidade

Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

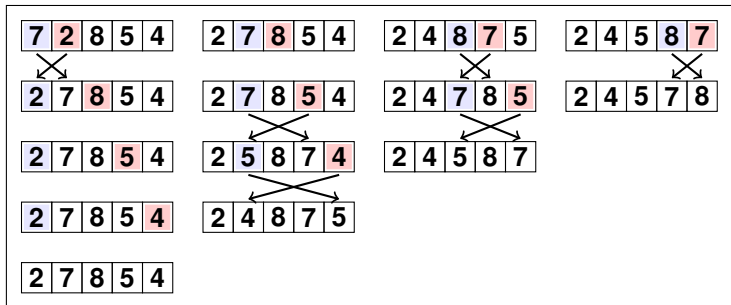
Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação



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

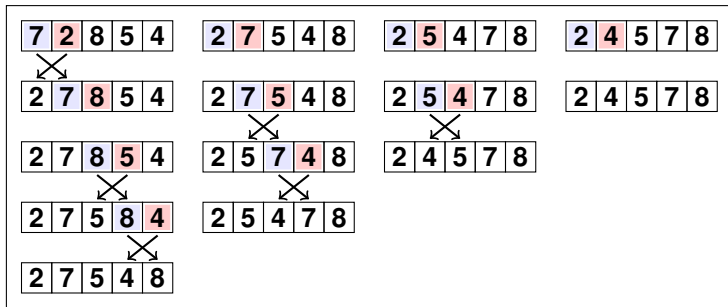
Ordenação por Flutuação (Bolha)

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.

```
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);  
}
```

Ordenação “Bolha”: Complexidade



- Para um vector de dimensão n é necessário fazer $(n - 1) + (n - 2) + \dots + 1$ 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 $O(n)$).

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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.

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

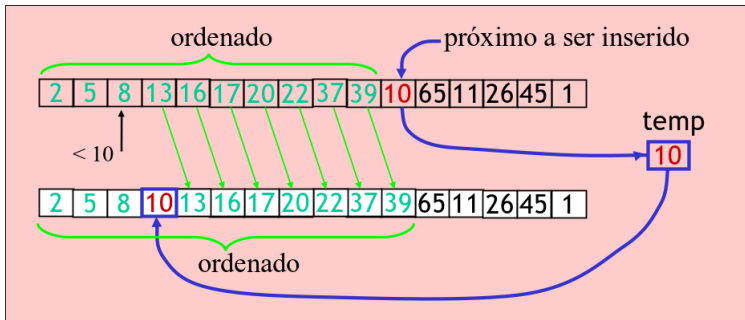
Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

Ordenação por Inserção



- 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.

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

```
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).

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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



- *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.

Fusão: Merge Sort

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

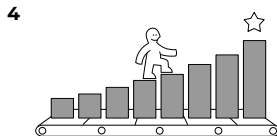
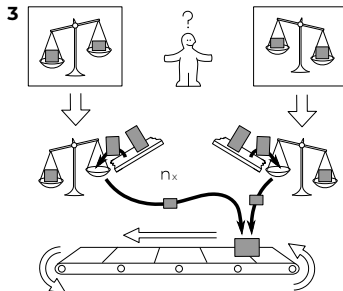
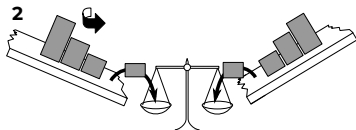
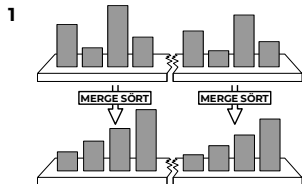
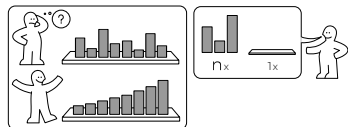
Quick Sort

Complexidade:
comparação

MERGE SÖRT

idea-instructions.com/merge-sort/
v1.1, CC by-nc-sa 4.0

IDEA



Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica: definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação (Bolha)

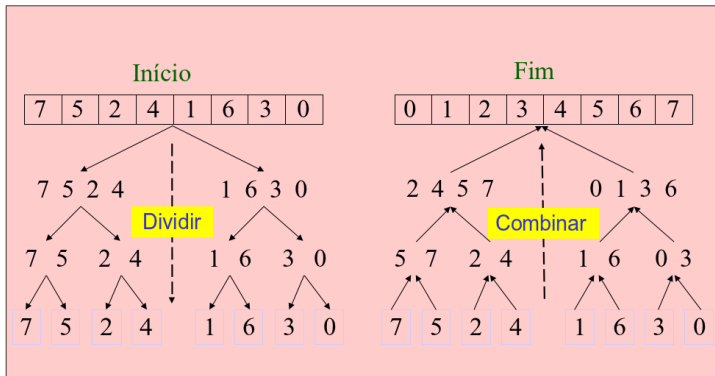
Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade: comparação

Fusão: Merge Sort




```
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);
}
```

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

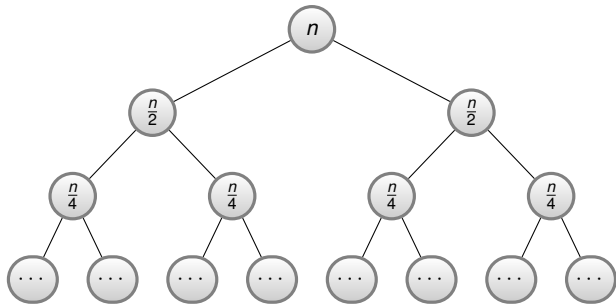
Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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



- 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.

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação *Big-O*

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

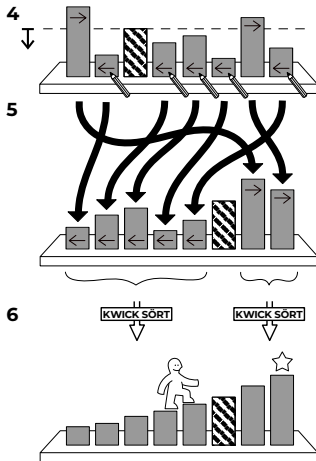
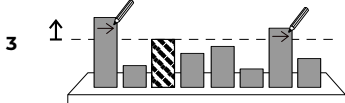
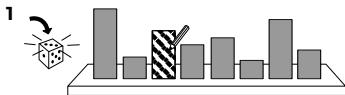
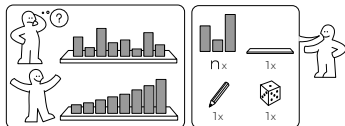
Quick Sort

Complexidade:
comparação

KWICK SÖRT

idea-instructions.com/quick-sort/
v1.0, CC by-nc-sa 4.0

IDEA



Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica: definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação (Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade: comparação

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

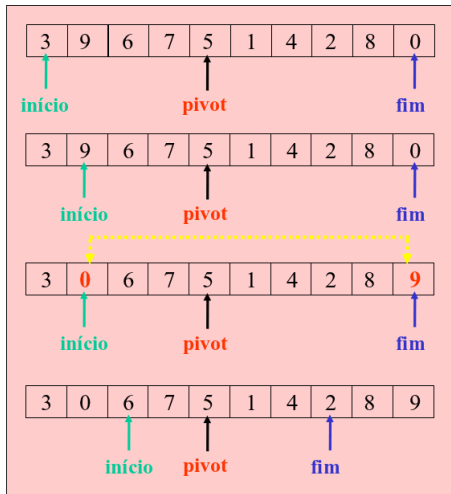
Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação



- 1 Escolher o pivot;
- 2 Movimentar o “início” 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: “início” > “fim”

QuickSort

Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

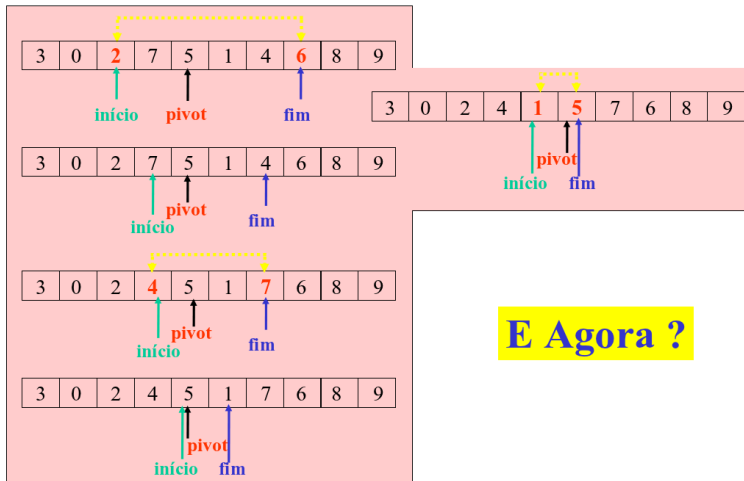
Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

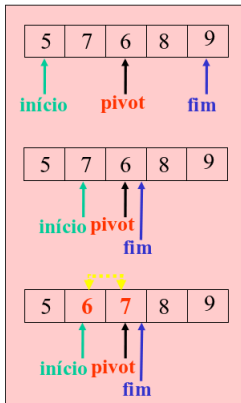
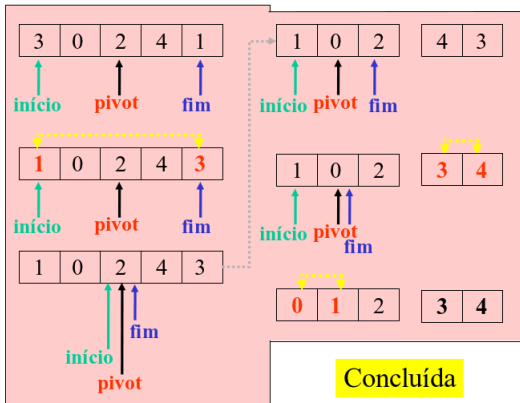


E Agora ?

Agora:

- Temos 2 subproblemas;
- “Atacamos” cada um deles em separado, utilizando o mesmo método;

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



Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

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;  
}
```

Complexidade
Algorítmica:
Introdução

Motivação

Complexidade Algorítmica:
definição

Notação Big-O

Ordenação

Ordenação por Seleção

Ordenação por Flutuação
(Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade:
comparação

- 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 $O(n \cdot \log(n))$

Complexidade: Gráficos Comparativos

Complexidade Algorítmica: Introdução

Motivação

Complexidade Algorítmica: definição

Notação Big-O

Ordenação

Ordenação por Seleção

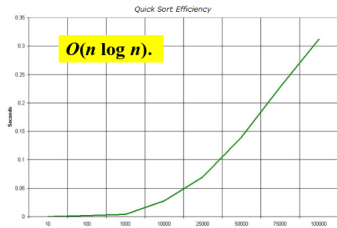
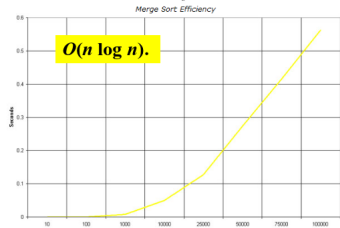
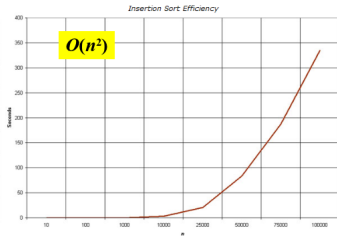
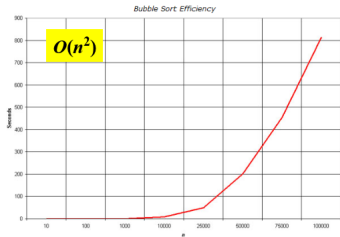
Ordenação por Flutuação (Bolha)

Ordenação por Inserção

Ordenação por Fusão

Quick Sort

Complexidade: comparação



- 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*.¹

¹ Dos algoritmos de ordenação apresentados!