ACH2002

Aula 16

Quicksort

(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

EACH USP

Aulas passadas

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort
  - HeapSort

EACH USP

Aula de hoje

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort
  - HeapSort
  - **QuickSort (ordenação rápida)**

QuickSort

- Algoritmo de ordenação interna mais rápido para várias situações.
- Autor: C. A. R. Hoare, em 1960
- Ideia básica:
  - dividir o problema de ordenar um conjunto com *n* itens em dois problemas menores;
  - problemas menores são ordenados independentemente;
  - resultados combinados para produzir solução do problema maior.

QuickSort

- Novamente temos um problema de dividir-e-conquistar:
- Ideia básica para um algoritmo de **ordenação genérico**:
  - Dividir: dividir o problema de ordenar um conjunto com *n* itens em dois problemas menores;
  - Conquistar: problemas menores são ordenados independentemente;
  - Combinar: resultados combinados para produzir solução do problema maior.

Como era o outro algoritmo de ordenação baseada nessa ideia?

Como era o outro algoritmo de ordenação baseada nessa ideia?

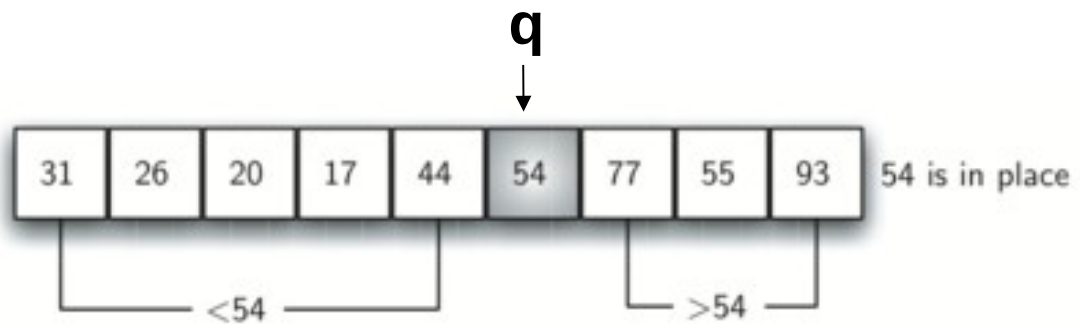
```
mergeSort (A, inicio, fim)
se inicio < fim
      m ← ⌊(inicio + fim) / 2⌋
      mergeSort(A, inicio, meio)
      mergeSort(A, meio+1, fim)
      merge(A, inicio, meio, fim)
```

QuickSort

- Quicksort fará diferente:
  - Não divide o vetor necessariamente no meio
  - Quem define o ponto de partição é a função de partição, que é também o coração da ordenação do quicksort (assim como a função merge é o coração do mergesort)

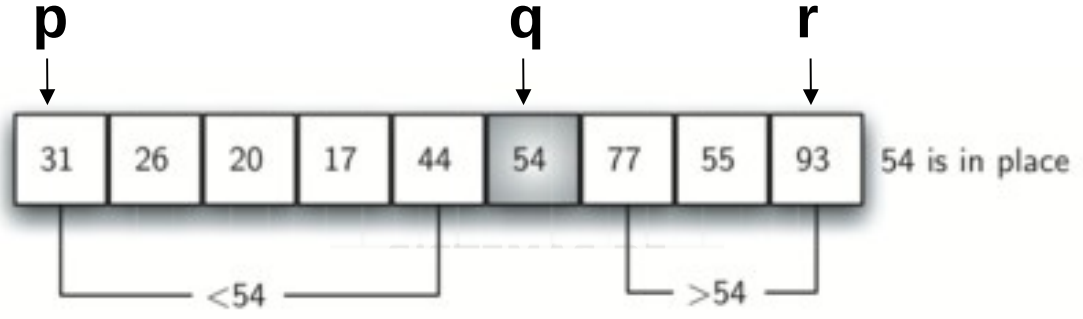
QuickSort

- Divisão e conquista do Quicksort:
  - Dividir: dividir o vetor em duas partes, separados pela posição **q** tal que todos os elementos à esquerda de **q** sejam menores ou iguais a **x** = A[**q**] (***pivô*** da partição) e todos os elementos à direita de **q** sejam maiores ou iguais a x = A[q];



QuickSort

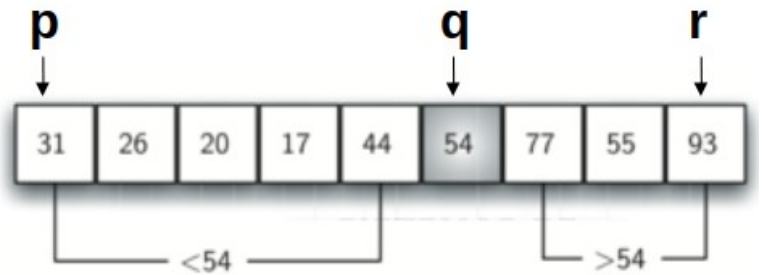
- Divisão e conquista do Quicksort:
  - Dividir: dividir o vetor em duas partes, separados pela posição **q** tal que todos os elementos à esquerda de **q** sejam menores ou iguais a **x** = A[**q**] (***pivô*** da partição) e todos os elementos à direita de **q** sejam maiores ou iguais a x = A[q] (na verdade, vai fazer com que isso aconteça!);
  - Conquistar: ordenar recursivamente os vetores A[p..q-1] e A[q+1..r];
  - Combinar: como os vetores foram ordenados na conquista e todos os elementos de A[p..q-1] <= A[q] <= A[q+1..r], nada a fazer neste passo.



QuickSort

- Divisão e conquista do Quicksort:
  - Dividir: dividir o vetor em duas partes, separados pela posição **q** tal que todos os elementos à esquerda de **q** sejam menores ou iguais a **x** = A[**q**] (***pivô*** da partição) e todos os elementos à direita de **q** sejam maiores ou iguais a x = A[q] (na verdade, vai fazer com que isso aconteça!);
  - Conquistar: ordenar recursivamente os vetores A[p..q-1] e A[q+1..r];
  - Combinar: como os vetores foram ordenados na conquista e todos os elementos de A[p..q-1] <= A[q] <= A[q+1..r], nada a fazer neste passo.

```
quickSort (A, p, r)
  se p < r
     q ← particao(A, p, r)
     quickSort(A, p, q-1)
     quickSort(A, q+1, r)
```



QuickSort

- Parte mais delicada do método - procedimento Partição:
- Há vários algoritmos para este procedimento;
- Alguns são melhores que outros;
- Veremos duas sugestões:
  - algoritmo sugerido por Ziviani;
  - algoritmo sugerido por Cormen *et al.*

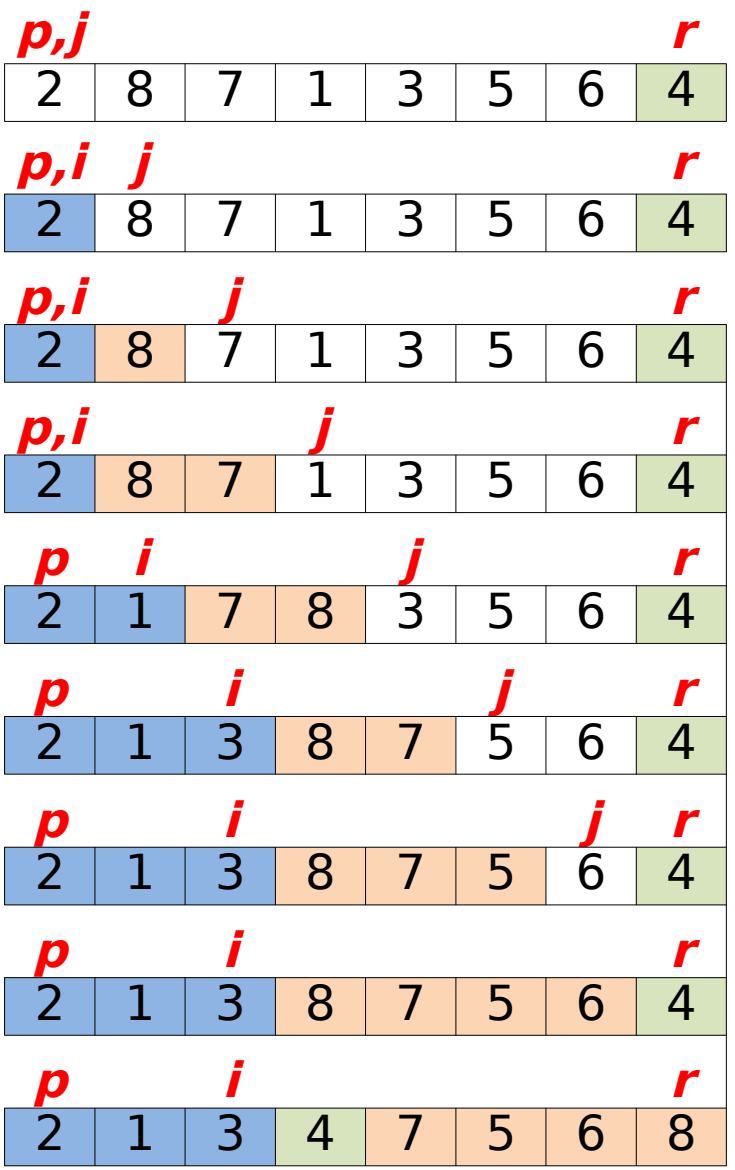
QuickSort

- Parte mais delicada do método - procedimento Partição:
- Há vários algoritmos para este procedimento;
- Alguns são melhores que outros;
- Veremos duas sugestões:
  - algoritmo sugerido por Ziviani;
  - algoritmo sugerido por Cormen *et al.*

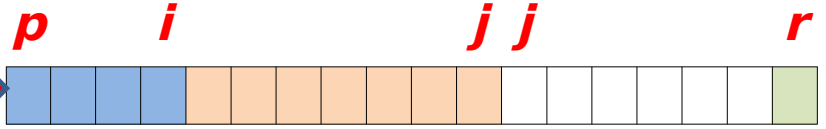
QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
 if A[j] ≤ x
       i ← i + 1
       trocar A[i] ↔ A[j]
fim para
troca   A[i+1] ↔ A[r]
retor     i + 1
```



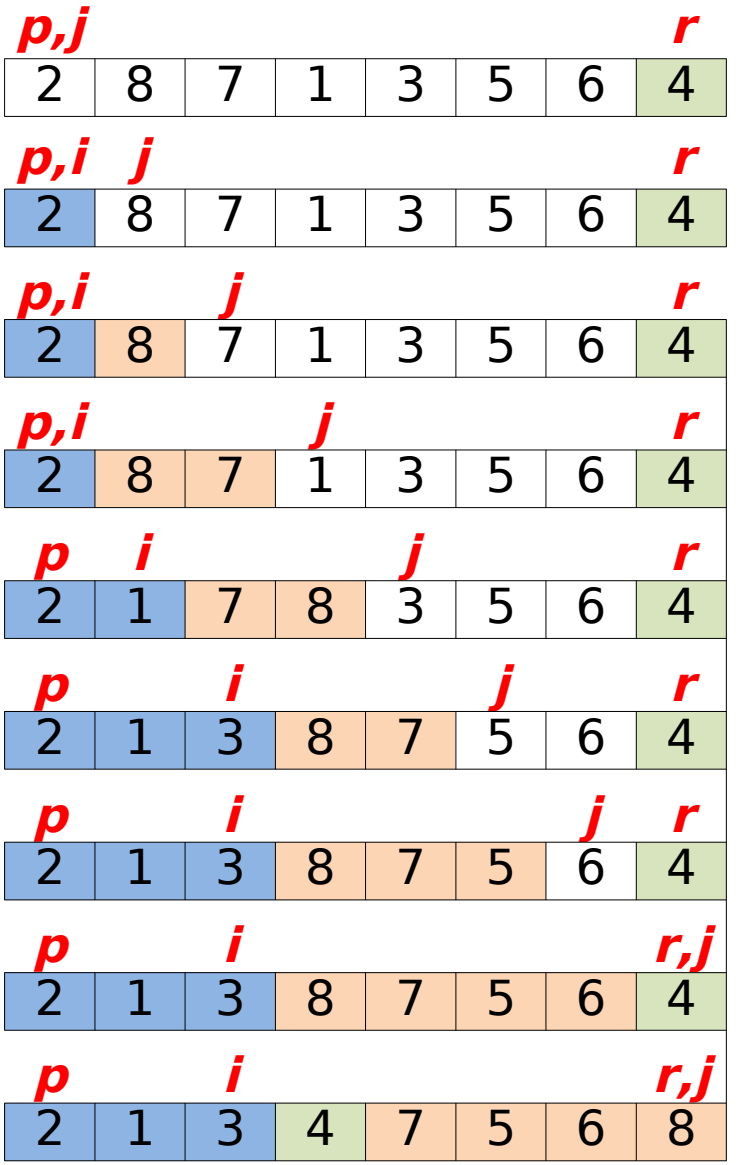
Define 4 áreas no arranjo!



QuickSort

- Algoritmo Cormen:

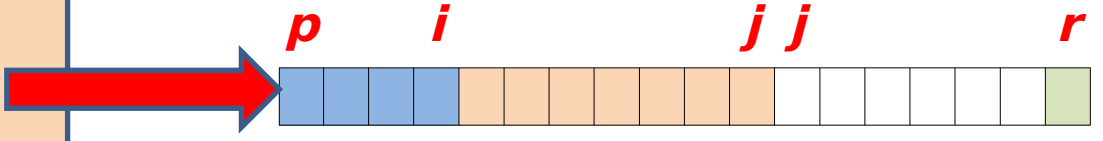
```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
 if A[j] ≤ x
        i ← i + 1
        trocar A[i] ↔ A[j]
fim para
troc   A[i+1] ↔ A[r]
```



Para qualquer índice de arranjo ***k:***

1. se $p \leq k \leq i$, então $A[k] \leq x$
2. se $i+1 \leq k \leq j-1$, então $A[k] > x$
3. se $k=r$, então $A[k]= x$

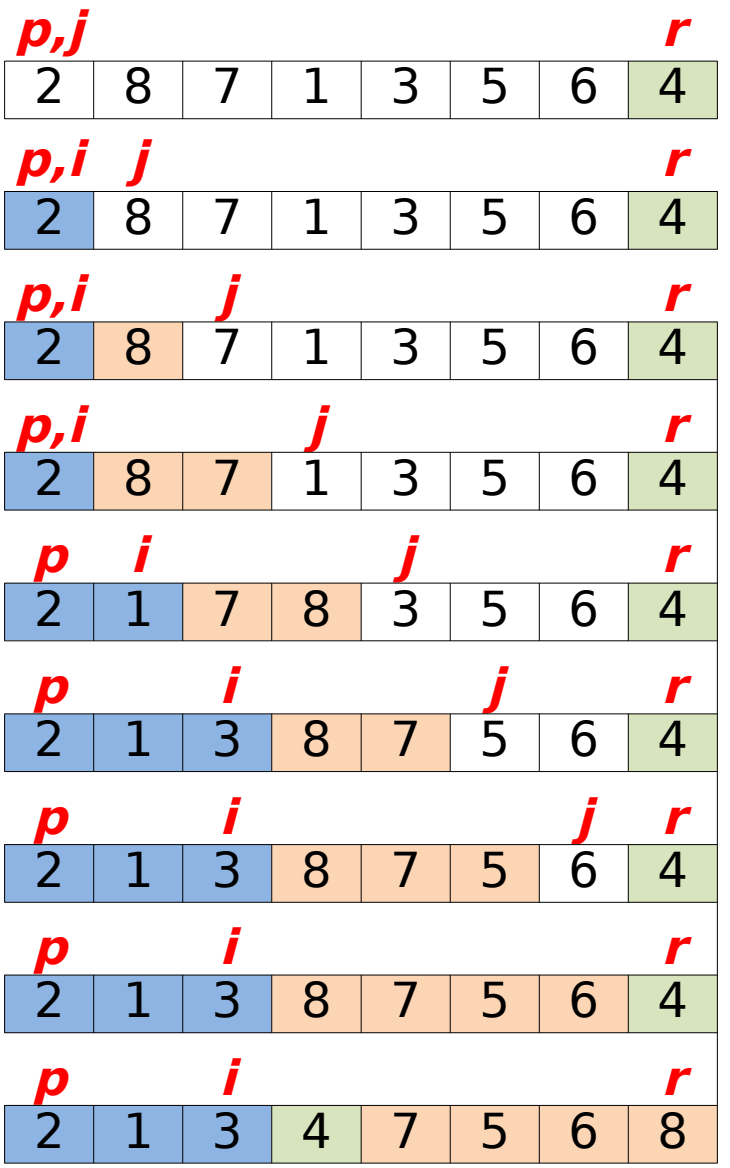
Quarta área: *elementos ainda não classificados.*



QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
 if A[j] ≤ x
        i ← i + 1
        trocar A[i] ↔ A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

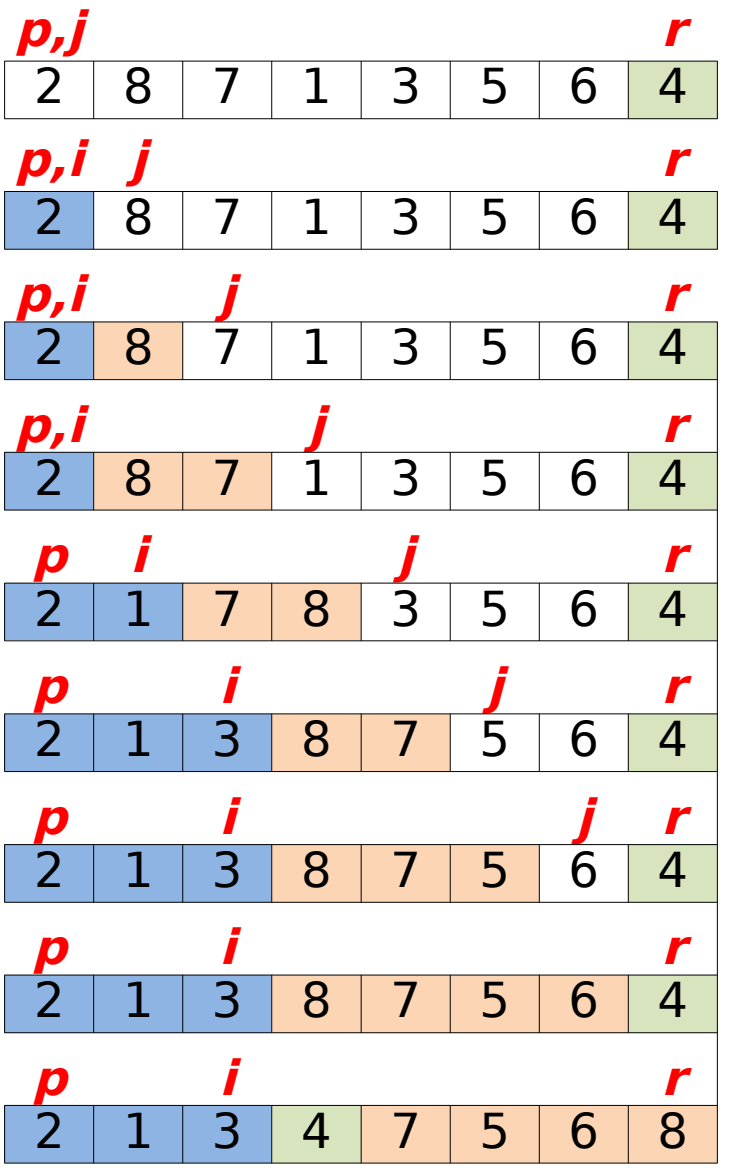


QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
  if A[j] ≤ x
          i ← i + 1
        trocar A[i] ↔ A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

```
quickSort (A,p,r)
se p < r
q ← particao(A,p,r)
quickSort(A, p, q-1)
quickSort(A, q+1, r)
```



QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
   if A[j] ≤ x
        i ← i + 1
        trocar A[i] ← A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

Quais são as operações que devemos considerar para analisar a complexidade deste algoritmo?

QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
    if A[j] ≤ x
        i ← i + 1
        trocar A[i] ← A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

Justamente o laço!

Qual é a complexidade deste trecho?

QuickSort

- Algoritmo Cormen:

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
   if A[j] ≤ x
        i ← i + 1
        trocar A[i] ← A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

Complexidade:

Θ(n), onde n = r - p + 1

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
		x			

QuickSort

- Algoritmo Ziviani:

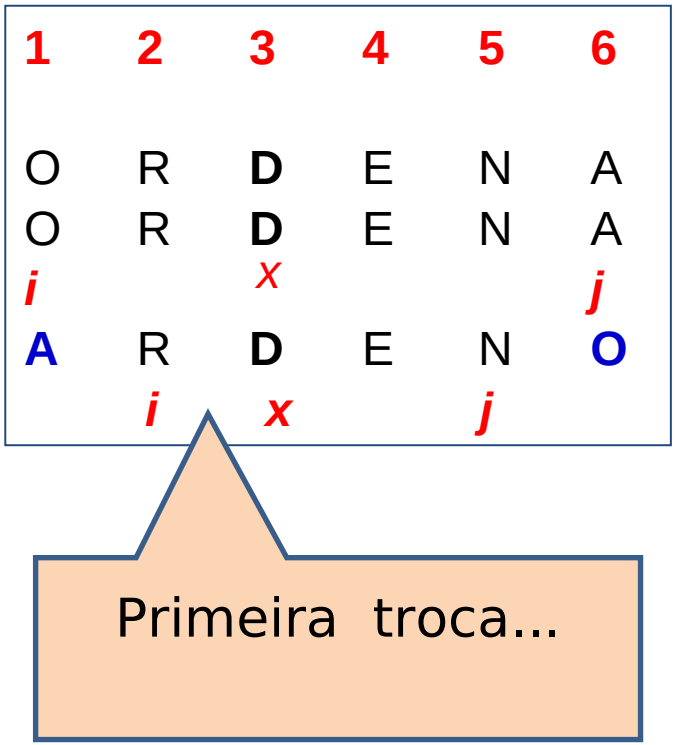
```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i		*x*			*j*

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```



QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x      i ← i + 1
   enquanto A[j] > x      j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	R	**D**	E	N	O
	i	*x*		*j*	

Volta ao *enquanto* mais externo

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x      i ← i + 1
   enquanto A[j] > x      j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	R	**D**	E	N	O
	i	*x*		*j*	

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	R	**D**	E	N	O
	i	*x*	*j*		

QuickSort

- Algoritmo Ziviani:

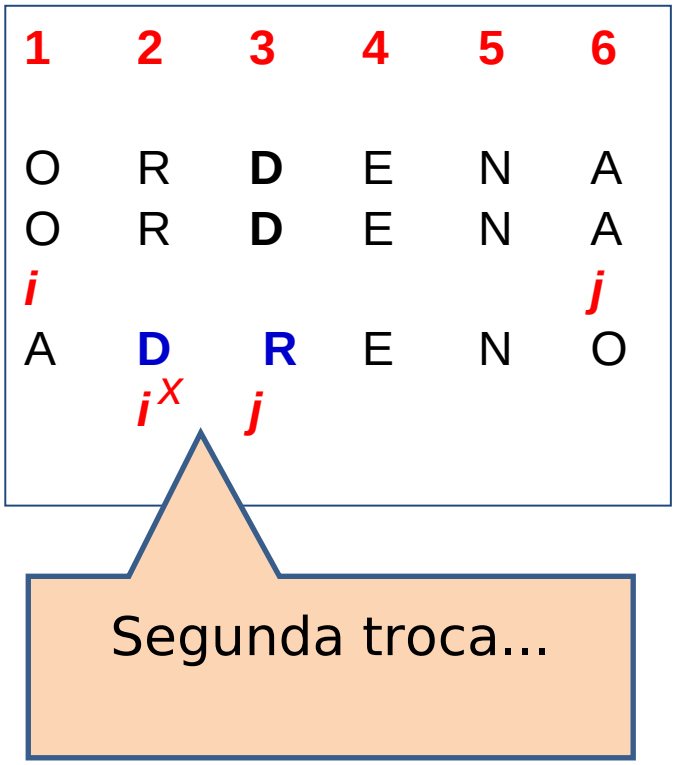
```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	R	**D**	E	N	O
	i	*j*^{X}			

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```



QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x      i ← i + 1
   enquanto A[j] > x      j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	**D**	**R**	E	N	O
	j *x*	*i*			

Volta ao *enquanto* mais externo ⇒
i e ***j*** se cruzam.
Sai do enquanto externo.
Finaliza algoritmo da Partição

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x      i ← i + 1
   enquanto A[j] > x      j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

1	2	3	4	5	6
O	R	**D**	E	N	A
O	R	**D**	E	N	A
i					*j*
A	**D**	**R**	E	N	O
	j *x*	*i*			

QuickSort

- Algoritmo Ziviani:

```
Particao(A[], p, r)
x ← A[(p+r) div 2]
i ← p
j ← r
enquanto i <= j
   enquanto A[i] < x     i ← i + 1
   enquanto A[j] > x     j ← j – 1
   se i <= j
      trocar A[i] ↔ A[j]
      i ← i + 1
      j ← j – 1
fim enquanto
retorna j
```

Complexidade:

Θ(n), onde n = r - p + 1

x

QuickSort

- Analisando a complexidade do QuickSort
  - É um algoritmo recursivo.
  - Portanto, devemos usar recorrência.
  - Temos que definir T(n):
    - problema: no caso do QuickSort, o tempo de execução depende do particionamento:
      - particionamento balanceado:
        - complexidade assintótica ≈ MergeSort (bem rápido!);
      - não balanceado:
        - complexidade assintótica ≈ Insertion Sort (bem lento!).

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?
    - quando o particionamento gera um subproblema com ***n-1*** elementos e outro com ***0*** elementos;
    - Por quê?

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?
    - quando o particionamento gera um subproblema com ***n-1*** elementos e outro com ***0*** elementos;
    - Por quê? Porque o elemento restante é justamente o pivô!

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?
    - quando o particionamento gera um subproblema com ***n-1*** elementos e outro com ***0*** elementos;
    - Por quê? Porque o elemento restante é justamente o pivô!
    - Se este particionamento não balanceado ocorrer em cada chamada recursiva, qual é a complexidade assintótica neste caso?

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

  - Pior caso?
    - Se este particionamento não balanceado ocorrer em cada chamada recursiva, qual é a complexidade assintótica neste caso?
      - procedimento de partição = **O(n)**
      - chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**
      - Para n > 0:

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n) = T(n-1) + n$$

QuickSort

- Expandindo a recorrência:
  - Pior caso?

$$T(n) = T(n-1) + n$$
$$T(n) = T(n-2) + n - 1 + n$$
$$T(n) = T(n-3) + n - 2 + n - 1 + n$$
$$\ldots$$
$$T(n) = T(n-k) + nk - \sum_{i=0}^{k-1} i$$
$$T(n) = T(n-k) + nk - k\left(\frac{k-1}{2}\right)$$
$$onde\, n - k = 0 \Rightarrow n = k$$

QuickSort

Expandindo a recorrência:

- Pior caso?

$$T(n) = T(n-1) + n$$
$$T(n) = T(n-2) + n - 1 + n$$
$$T(n) = T(n-3) + n - 2 + n - 1 + n$$
$$\cdots$$
$$T(n) = T(n-k) + nk - \sum_{i=0}^{k-1} i$$
$$T(n) = T(n-k) + nk - k\left(\frac{k-1}{2}\right)$$

$$onde\, n - k = 0 \Rightarrow n = k$$

$$T(n) = O(1) + n^2 - \frac{n^2 - n}{2}$$
$$T(n) = O(1) + \frac{2n^2 - n^2 + n}{2}$$
$$T(n) = O(1) + \frac{n^2}{2} + \frac{n}{2}$$
$$T(n) = O(n^2)$$

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?
    - procedimento de partição = **O(n)**
    - chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n) = T(n-1) + n$$

$$T(n) \in O(n^2)$$

Semelhante ao tempo da ordenação por inserção!
Quando o arranjo está ordenado, a ordenação por inserção é executada no tempo **O(n)** e aqui continua sendo executada em **O(n²).**

QuickSort

- Analisando o particionamento do QuickSort
  - Melhor caso?

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Melhor caso?
    - quando o particionamento gera dois subproblemas, cada um com tamanho não maior que ***n/2:*** um com tamanho $\lfloor n/2 \rfloor$ e outro com tamanho $\lceil n/2 \rceil - 1$
    - Qual é a recorrência neste caso?

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort
  - Melhor caso?
    - quando o particionamento gera dois subproblemas, cada um com tamanho não maior que ***n/2:*** um com tamanho $\lfloor n/2 \rfloor$ e outro com tamanho $\lceil n/2 \rceil - 1$
    - Qual é a recorrência neste caso?

$$T(n) \leq 2T(n/2) + O(n)$$

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

QuickSort

- Analisando o particionamento do QuickSort

```
quickSort (A,p,r)
se p < r
  q ← particao(A,p,r)
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
```

  - Melhor caso?
    - quando o particionamento gera dois subproblemas, cada um com tamanho não maior que ***n/2:*** um com tamanho $\lfloor n/2 \rfloor$ e outro com tamanho $\lceil n/2 \rceil - 1$
    - Qual é a recorrência neste caso?

$$T(n) \leq 2T(n/2) + O(n)$$

    - Aplicando caso 2 do Teorema Mestre:

$$T(n) = O(n \lg n)$$

$$f(n) = O\left(n^{\log_b a - \epsilon}\right), \textbf{algum} \epsilon > 0 \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$$

$$f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$$

$$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right), \textbf{algum} \epsilon > 0 \Rightarrow T(n) = \Theta\left(f(n)\right)$$

QuickSort

- Analisando o particionamento do QuickSort
  - Particionamento desbalanceado
    - caso médio do QuickSort é muito mais próximo do melhor caso do que do pior caso;
      Consideremos, por exemplo, que ***sempre*** a partição gere uma divisão proporcional em cada recursão. Exemplo: partição 9 para 1.
    - A equação de recorrência seria:

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

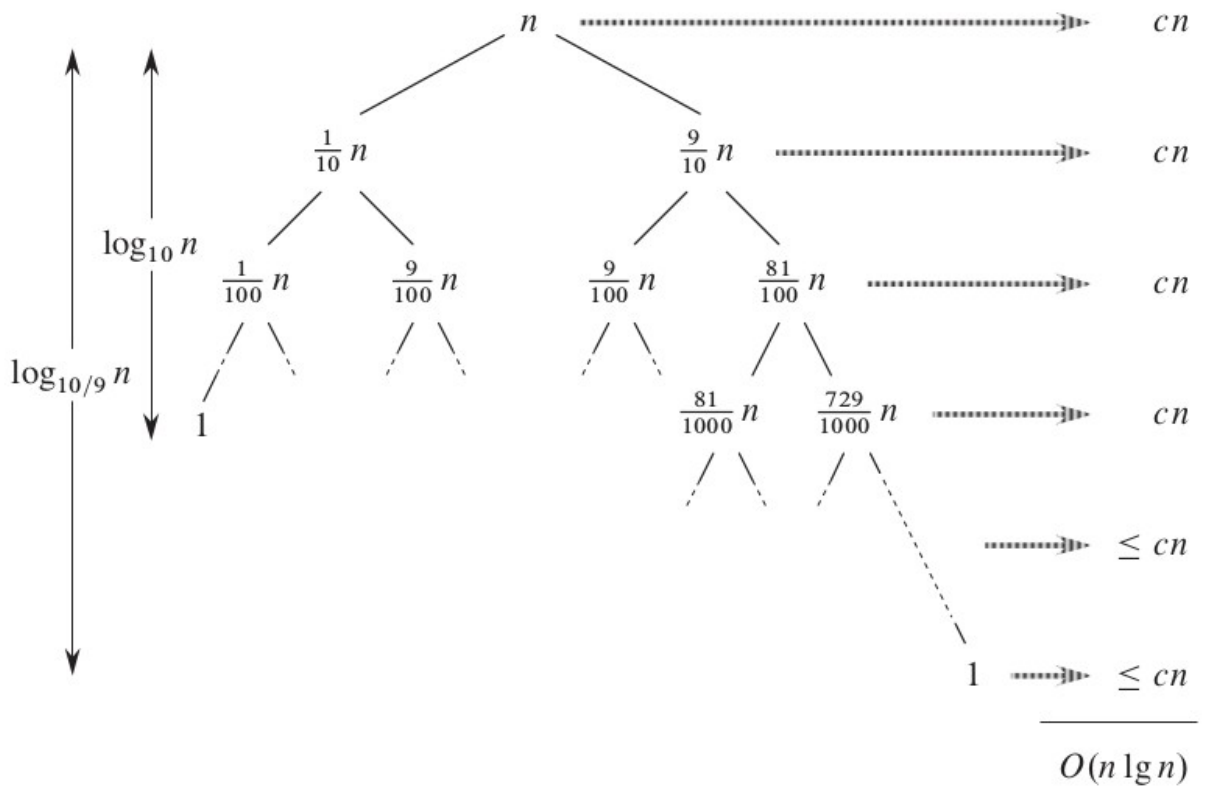
QuickSort

- Analisando o particionamento do QuickSort
  - Particionamento desbalanceado
    - A equação de recorrência seria:

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

Expandindo-se esta equação chegaríamos também a:

$$T(n) = O(n \lg n)$$



QuickSort

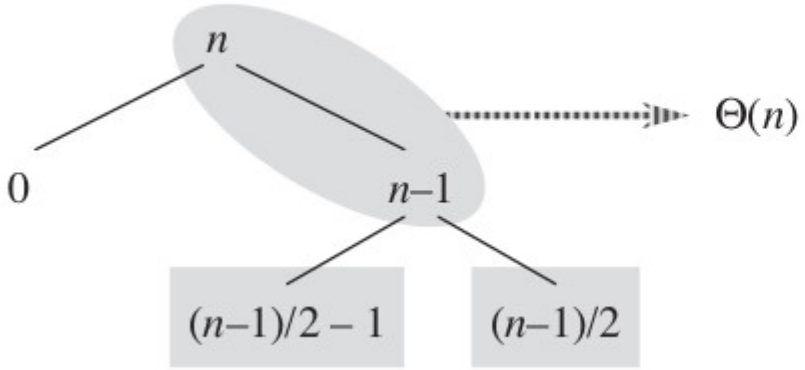
- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Não podemos garantir a condição anterior (divisão sempre igual), mas...

QuickSort

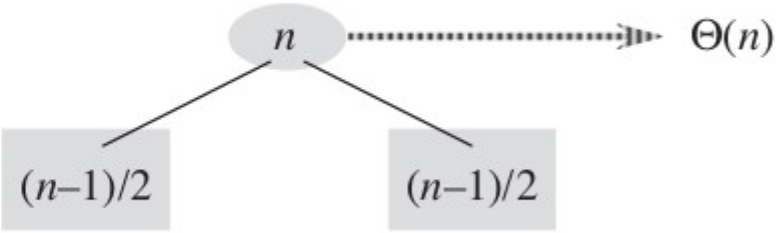
- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Não podemos garantir a condição anterior (divisão sempre igual), mas...
    - podemos considerar que algumas partições são ***boas*** e outras são ***ruins,*** distribuídas aleatoriamente na árvore de recursão;

QuickSort

- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Não podemos garantir a condição anterior (divisão sempre igual), mas...
    - podemos considerar que algumas partições são ***boas*** e outras são ***ruins,*** distribuídas aleatoriamente na árvore de recursão;
    - Supondo uma partição ***ruim*** seguida sempre de uma ***boa***:



Ruim / boa



Boa

QuickSort

- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Supondo uma partição ***ruim*** seguida sempre de uma ***boa***:
      1. Partição ruim (pior caso): $T(0)+T(n-1)$
      2. Partição boa (melhor caso): $T\left(\frac{n-1}{2}\right)+T\left(\frac{n-1}{2}\right)$

      ...

      n. Total: $T(0)+T\left(\frac{n-1}{2}\right)+T\left(\frac{n-1}{2}\right)$

Assim, o custo da partição ruim é absorvido pela partição boa!!!

QuickSort

- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Supondo uma partição ***ruim*** seguida sempre de uma ***boa***:
      1. Partição ruim (pior caso): $T(0)+T(n-1)$
      2. Partição boa (melhor caso): $T\left(\frac{n-1}{2}\right)+T\left(\frac{n-1}{2}\right)$

      ...

      n. Total: $T(0)+T\left(\frac{n-1}{2}\right)+T\left(\frac{n-1}{2}\right)$

Assim, o custo da partição ruim é absorvido pela partição boa!!!

Portanto, o tempo de execução do *QuickSort* quando os níveis se alternam entre partições boas e ruins é semelhante ao custo para partições boas sozinhas:

T(n) = O(n log n), mas com uma constante maior.

QuickSort

- Não podemos prever a ordem dos elementos do arranjo de entrada.
- E, portanto, não podemos garantir a aproximação da execução do algoritmo no caso médio considerando somente o arranjo de entrada.
- Então, como fazer para aproximar a execução do algoritmo ao caso médio? Onde podemos mexer?

QuickSort

- Podemos escolher o pivô aleatoriamente, em vez de fixar uma regra (no algoritmo do Cormen, escolhemos sempre o último elemento).

```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
    if A[j] ≤ x
          i ← i + 1
          trocar A[i] ← A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```

- Para isso, basta sortear uma posição entre p e q e trocar o elemento de lá com o que está na posição r

QuickSort

- Implementação com escolha aleatória do pivô:

```
#include <stdlib.h>
int particaoAleatoria(int[] A, int p, int r)
{
  int deslocamento, i, temp;
  time_t t;

   /* Inicializa gerador de número aleatorio */
   srand((unsigned) time(&t));

  // Escolhe um numero aleatorio entre p e r
  deslocamento = rand() % (r-p+1); //retorna um int entre 0 e (r-p)
  i = p + deslocamento;            // i eh tal que p <= i <= r

  // Troca de posicao A[i] e A[r]
  temp = A[r];
  A[r] = A[i];
  A[i] = temp;

  return particao(A, p, q);
}
```

QuickSort

- Implementação com escolha aleatória do pivô:

```
quickSortAleatorio(A, p, r)

  if (p < r)

    q = particaoAleatoria(vetor, p, r);
    quickSortAleatorio(vetor, p, q - 1);
    quickSortAleatorio(vetor, q + 1, r);
```

QuickSort

- Vimos que a complexidade assintótica do ***QuickSort*** no melhor caso é igual à complexidade do ***MergeSort.***
- Por que, então, o ***QuickSort*** é considerado melhor?

QuickSort

- Vimos que a complexidade assintótica do ***QuickSort*** no melhor caso é igual à complexidade do ***MergeSort.***
- Por que, então, o ***QuickSort*** é considerado melhor?
  - Porque executa a ordenação sem usar arranjo auxiliar (ordenação in loco!).

QuickSort

- Vimos que a complexidade assintótica do ***QuickSort*** no melhor caso é igual à complexidade do ***MergeSort.***
- Por que, então, o ***QuickSort*** é considerado melhor?
  - Porque executa a ordenação sem usar arranjo auxiliar (ordenação in loco!).
  - É estável?

QuickSort

- Vimos que a complexidade assintótica do ***QuickSort*** no melhor caso é igual à complexidade do ***MergeSort.***
- Por que, então, o ***QuickSort*** é considerado melhor?
  - Porque executa a ordenação sem usar arranjo auxiliar (ordenação in loco!).
  - Mas NÃO é estável...

Comparando...

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	O(n)	$O(n^2)$	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	O(1)	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)	O(n)	O(n)	sim	não
BubbleSort											
MergeSort	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	não	sim
HeapSort	O(n lgn)	O(n lgn)	O(n lgn)							sim	não
QuickSort	O(n lgn)	$O(n^2)$	O(n lgn)							sim	não

Comparação (Ziviani)

Tempo de execução:

- Oservação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória:

	5.00	5.000	10.000	30.000
Inserção	11,3	87	161	–
Seleção	16,2	124	228	–
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

Comparação (Ziviani)

- Registros na ordem ascendente:

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	–
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

Comparação (Ziviani)

Tempo de execução:

- Registros na ordem descendente:

	500	5.000	10.000	30.000
Inserção	40,3	305	575	–
Seleção	29,3	221	417	–
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

Comparação (Ziviani)

Observações sobre os métodos:

1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza.
2. O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação Heapsort/Quicksort é constante para todos os tamanhos.
4. A relação Shellsort/Quicksort aumenta se o número de elementos aumenta.
5. Para arquivos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort.
6. Se a entrada aumenta, o Heapsort é mais rápido que o Shellsort.
7. O Inserção é o mais rápido se os elementos estão ordenados.
8. O Inserção é o mais lento para qualquer tamanho se os elementos estão em ordem descendente.
9. Entre os algoritmos de custo $O(n^2)$, o Inserção é melhor para todos os tamanhos aleatórios experimentados.

Exercícios

- Qual a complexidade para o número de comparações (C(n)) e movimentações de registros (M(n)) para o Quicksort?
- Façam os exercícios dos capítulos dos livros de referências.

Referências (com exercícios!)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - 3a. ed. Edição Americana. Editora Campus, 2002. Cap 7
- Paulo Feofiloff. Algoritmos em C. Cap 11 https://www.ime.usp.br/~pf/algoritmos-livro/
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 3a. Edição, 2004. Cap 4.1.4 - http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes.php
- Apostila Prof. Marcio – cap 6.4