Algoritmos de Ordenação

QuickSort

Professora:

Fátima L. S. Nunes

SISTEMAS DE INFORMAÇÃO

EACH

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos:?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)
  - *MergeSort* (Ordenação por intercalação)

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos:?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)
  - *MergeSort* (Ordenação por intercalação)
  - Hoje: ***QuickSort*** (Ordenação rápida)

QuickSort

- Algoritmo de ordenação interna mais rápido para várias situações.
- Autor: C.A.R.Hoare, em 1960 (estudante visitante na Universidade de Moscou).
- 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:
  - 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.

QuickSort

- Parte mais delicada do método - procedimento Partição:
  - particionar o arranjo em dois subarranjos;
  - ordenar cada subarranjo escolhendo arbitrariamente um item ***x*** chamado ***pivô***;
  - Ao final, o vetor A deve estar particionado em duas partes:
    - ***Esquerda**:* itens menores ou iguais a ***x***
    - ***Direita:*** itens maiores ou iguais a ***x.***

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.*

SISTEMAS DE INFORMAÇÃO

EACH

QuickSort

- Algoritmo Ziviani:

```
Particao(A[],p,r)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
   enquanto A[i] < x i = i + 1;
   enquanto A[j] > x j = j - 1;
  trocar A[i] com A[j]
fim enquanto
```

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
   trocar A[i] com A[j]
fim enquanto
```

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

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
   trocar A[i] com A[j]
fim enquanto
```

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

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
   enquanto A[i] < x i = i + 1;
   enquanto A[j] > x j = j – 1;
  trocar A[i] com A[j]
fim enquanto
```

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

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
   trocar A[i] com A[j]
fim enquanto
```

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**

Primeira troca...
Volta ao *enquanto* mais externo

SISTEMAS DE INFORMAÇÃO

EACH

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
  trocar A[i] com A[j]
fim enquanto
```

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*

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j – 1;
  trocar A[i] com A[j]
fim enquanto
```

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*	

SISTEMAS DE INFORMAÇÃO

EACH

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j – 1;
  trocar A[i] com A[j]
fim enquanto
```

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*		

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
  trocar A[i] com A[j]
fim enquanto
```

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*			

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j – 1;
   trocar A[i] com A[j]
fim enquanto
```

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
	i	*j*			

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
    enquanto A[i] < x i = i + 1;
    enquanto A[j] > x j = j - 1;
   trocar A[i] com A[j]
fim enquanto
```

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
	i	*j*			

Segunda troca...
Volta ao *enquanto* mais externo

QuickSort

- Algoritmo:

```
Particao(A[],inicio,fim)
x ← item do vetor escolhido arbitrariamente
i ← 1
j ← tamanho do vetor
enquanto i < j
   enquanto A[i] < x i = i + 1;
   enquanto A[j] > x j = j – 1;
  trocar A[i] com A[j]
fim enquanto
```

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	*i*			

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

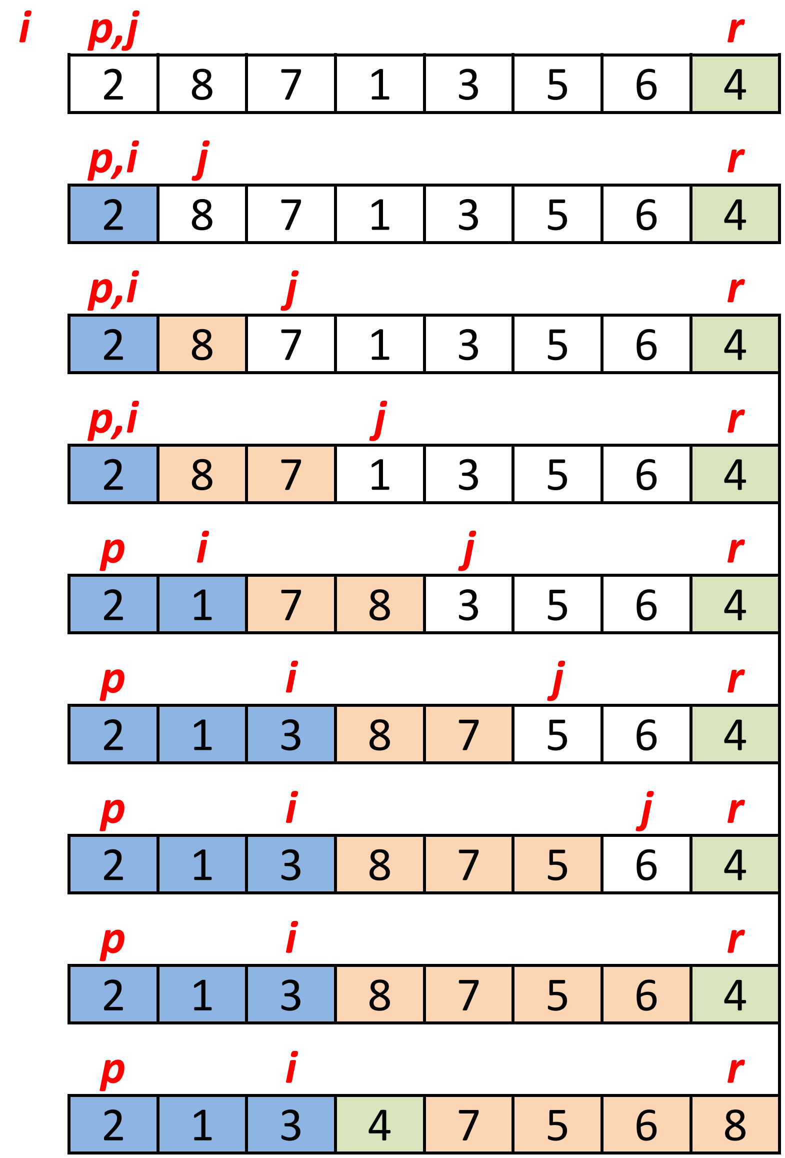
QuickSort

- Algoritmo:
  - ao final, o vetor A[Esq..Dir] está particionado de tal forma que:
    - itens em A[Esq], A[Esq+1], ..., A[j] são menores ou iguais a ***x.***
    - itens em A[i], A[i+1], ..., A[Dir] são maiores ou iguais a ***x.***

QuickSort

- Algoritmo Cormen:

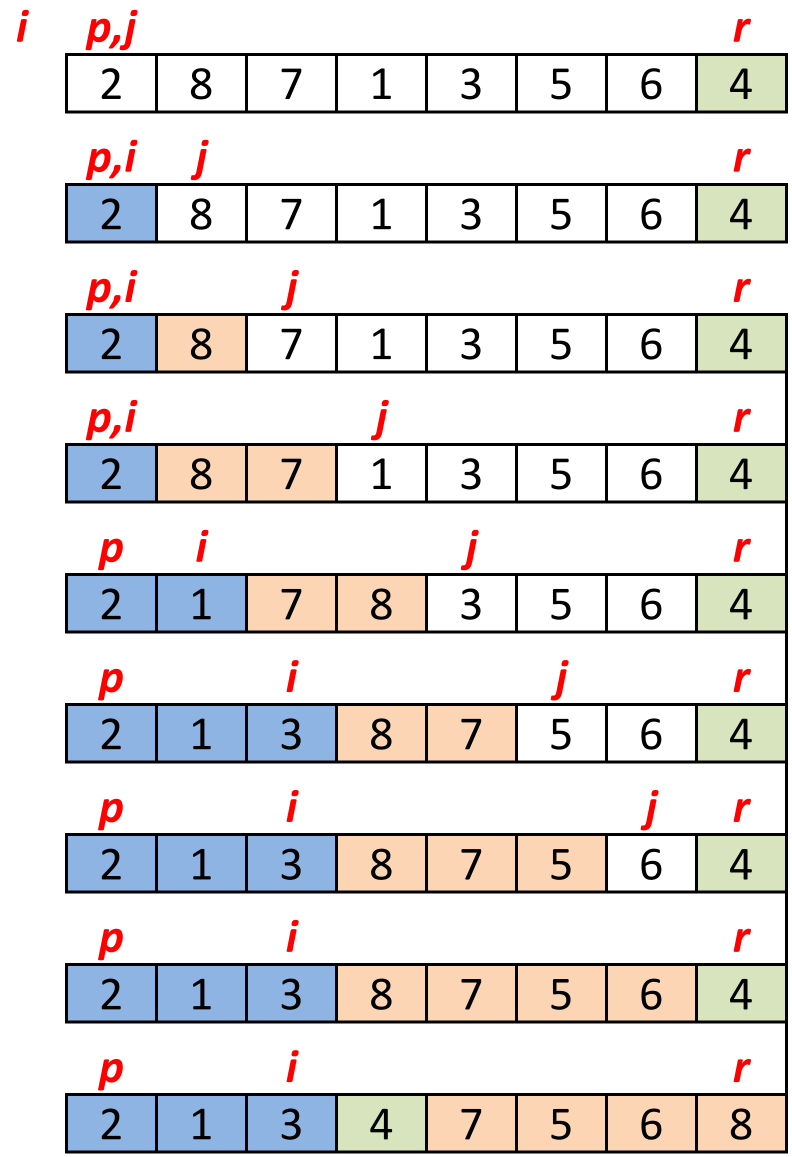
```
Particao(A[], p, r)
x ← A[r]
i ← p - 1
para j ← p até r-1 faça
   if A[j] ≤ x
       i ← i + 1
       trocar A[i] ← A[j]
fim para
trocar A[i+1] ↔ A[r]
retorna i + 1
```



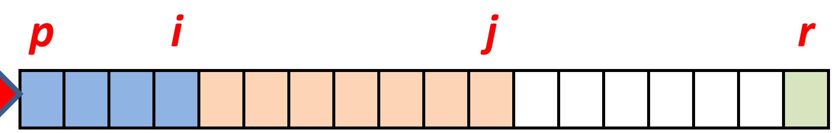
QuickSort

- 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
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
tro    A[i+1] ↔ A[r]
ret        + 1
```

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*

i	p,j							r
	2	8	7	1	3	5	6	4

p,i	j						r
2	8	7	1	3	5	6	4

p,i		j					r
2	8	7	1	3	5	6	4

p,i			j				r
2	8	7	1	3	5	6	4

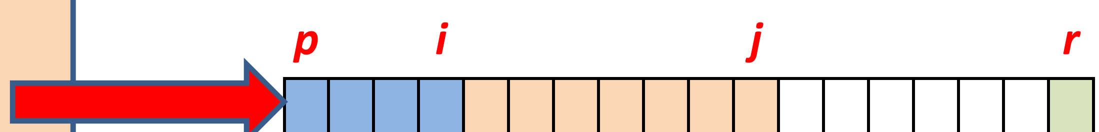
p	i			j			r
2	1	7	8	3	5	6	4

p		i			j		r
2	1	3	8	7	5	6	4

p		i				j	r
2	1	3	8	7	5	6	4

p		i					r
2	1	3	8	7	5	6	4

p		i					r
2	1	3	4	7	5	6	8



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]
ret        + 1
```

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

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

Quarta área: *elementos ainda não classificados.*

i	p,j							r
	2	8	7	1	3	5	6	4

p,i	j						r
2	8	7	1	3	5	6	4

p,i		j					r
2	8	7	1	3	5	6	4

p,i			j				r
2	8	7	1	3	5	6	4

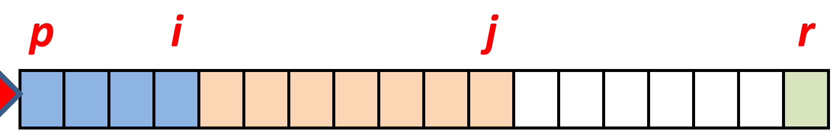
p	i			j			r
2	1	7	8	3	5	6	4

p		i			j		r
2	1	3	8	7	5	6	4

p		i				j	r
2	1	3	8	7	5	6	4

p		i					r
2	1	3	8	7	5	6	4

p		i					r
2	1	3	4	7	5	6	8



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:

O(n), onde n = r – p + 1

QuickSort

- Agora que já vimos o algoritmo do método que faz a partição, podemos definir o algoritmo completo do QuickSort:

```
quickSort (A,p,r)

se p < r

    q ← particao(A,p,r)

    quickSort(A, p, q-1)

    quickSort(A, q+1, r)
```

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)
```

SISTEMAS DE INFORMAÇÃO

EACH

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)
```

SISTEMAS DE INFORMAÇÃO

EACH

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?
    - 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?
      - procedimento de partição = **O(n)**
      - chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**

QuickSort

- Analisando o particionamento do QuickSort
  - 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).**

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

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

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?
    - procedimento de partição = **O(n)**
    - chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**

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

    - se somarmos os custos envolvidos em cada recursão, teremos uma série aritmética;
    - podemos calcular a complexidade expandindo a equação de recorrência...

QuickSort

- Analisando o particionamento do QuickSort
  - 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

- Analisando o particionamento do QuickSort
  - Pior caso?

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

$$T(n) = T(n-2) + n - 1 + n$$

$$T(n) = T(n-3) + n - 2 + n - 1 + n$$

$$\ldots$$

$$T(n) = T(n-k) + nk - \sum_{i=0}^{k-1} i$$

$$T(n) = T(n-k) + nk - k\left(\frac{k-1}{2}\right)$$

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

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

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

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

$$T(n) = O(n^2)$$

QuickSort

- Analisando o particionamento do QuickSort
  - Pior caso?

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

- procedimento de partição = **O(n)**
- chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**

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

- se somarmos os custos envolvidos em cada recursão, teremos uma série aritmética;
- podemos calcular a complexidade expandindo a equação de recorrência...

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

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?
    - procedimento de partição = **O(n)**
    - chamada recursiva a um arranjo de tamanho ***0:*** **T(0)=O(1).**

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

    - Se somarmos os custos envolvidos em cada recursão, teremos uma série aritmética – podemos calcular a complexidade por substituição...

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

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(n^2).**

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

```
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?

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?

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

SISTEMAS DE INFORMAÇÃO

EACH

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?

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

    - Aplicando caso 2 do Teorema Mestre:

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

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

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

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

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

QuickSort

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

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

    - Expandindo-se esta equação chegaríamos também a:

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

(ver árvore de recursão página 122 do Cormen *et al.*)

QuickSort

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

QuickSort

- Analisando o particionamento do QuickSort
  - Intuição para caso médio:
    - Não podemos garantir a condição anterior (divisão sempre balanceada), 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 balanceada), 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***:
      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: $T(0)+T\left(\frac{n-1}{2}\right)+T\left(\frac{n-1}{2}\right)$

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

```
particao(int[] vetor, int ini, int fim)
{
   int pivo, i, j, temp;
   pivo = vetor[fim];
   i = ini;   j = fim - 1;

   while (i <= j)
   {
      if (vetor[i] <= pivo) { i++;}
      else
         if (vetor[j] > pivo) {j--;}
         else
         {
            temp = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = temp;
            i++;
            j--;
         }
   }
  vetor[fim] = vetor[i];
  vetor[i] = pivo;
  return i;
 }
```

```
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

```
void quickSort(int[] vetor, int ini, int fim)
{
  if (ini < fim)
  {
    int q = particao(vetor, ini, fim);
    quickSort(vetor, ini, q - 1);
    quickSort(vetor, q + 1, fim);
    }
}
```

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 nosso algoritmo anterior, 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
```

SISTEMAS DE INFORMAÇÃO

EACH

QuickSort

- Implementação com escolha aleatória do pivô:

```
int particaoAleatoria(int[] A, int ini, int fim)
{
  int i, temp;
  double f;

  // Escolhe um numero aleatorio entre ini e fim

  f = java.lang.Math.random(); // retorna um real f tal que 0 <= f < 1
  i = (int) (ini + (fim - ini) * f); // i eh tal que ini <= i < fim

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

  return particaoAleatoria(A, ini, fim);
 }
}
```

QuickSort

- Implementação com escolha aleatória do pivô:

```
void quickSortAleatorio(int[] vetor, int ini, int fim)
{
  if (ini < fim)
  {
    int q = particaoAleatoria(vetor, ini, fim);
    quickSortAleatorio(vetor, ini, q - 1);
    quickSortAleatorio(vetor, q + 1, fim);
    }
}
```

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?

SISTEMAS DE INFORMAÇÃO

EACH

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.

QuickSort

- Resumindo:
  - Ordenação sem usar arranjo auxiliar;
  - Usa quantidade de memória constante, além do tamanho do próprio arranjo:
  - Complexidade do pior caso: $O(n^2)$
  - Complexidade do pior caso: $O(n \log n)$
  - Complexidade do caso médio: $O(n \log n)$

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002.
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004 (texto base)
- Notas de aula – Prof. Delano Beder – EACH-USP

Algoritmos de Ordenação

QuickSort

Professora:

Fátima L. S. Nunes