

# *Quicksort*

Algoritmos e Estruturas de Dados  
Prof. Dr. Luciano Demétrio Santos Pacífico  
{luciano.pacifico@ufrpe.br}



# Conteúdo

- **Introdução**
- ***Quicksort***

---

# Introdução



UNIVERSIDADE  
FEDERAL RURAL  
DE PERNAMBUCO

# Introdução

- Vimos que o *Mergesort* usava a abordagem dividir-para-conquistar, porém fazia a divisão de forma trivial, levando a muito trabalho na fase de conquistar.
- Será que poderíamos encontrar um esquema onde a divisão fosse feita com mais cuidado, de forma a tornar a conquista mais rápida?

# Introdução

- Uma outra abordagem que segue o paradigma dividir-para-conquistar e que possui bom desempenho no caso médio é o algoritmo *Quicksort*.
- Embora a complexidade de pior caso do *Quicksort* seja da ordem de  $O(n^2)$ , tal algoritmo é uma escolha bastante frequente para a execução da tarefa de ordenação.
- Isso se deve ao fato de que o número operações de custo constante são bem menores durante sua execução, o que torna sua complexidade de caso médio da ordem de  $\theta(n \log n)$ .

---

# ***Quicksort***



UNIVERSIDADE  
FEDERAL RURAL  
DE PERNAMBUCO

# *Quicksort*

- O principal procedimento do *Quicksort* é o de particionamento.
- O vetor  $A[p..r]$  é rearranjado de acordo com a escolha de um ponto arbitrário  $q$ , chamado de **pivô**.
- O vetor  $A$  é particionado em duas partes:
  - Parte esquerda: chaves  $\leq q$ ;
  - Parte direita: chaves  $\geq q$ .

# Quicksort

- Os três passos da execução do algoritmo (seguindo o paradigma dividir-para-conquistar) podem ser definidos como segue:
  - Dividir: Particionar o vetor  $A[p..r]$  em dois (possivelmente vazios) subvetores  $A[p..q-1]$  e  $A[q+1..r]$  de forma que cada elemento de  $A[p..q-1]$  seja menor que ou igual a  $A[q]$ , que por sua vez é menor ou igual a cada elemento de  $A[q+1..r]$ . Calcule o índice  $q$  como parte do processo de particionamento.



# Quicksort

- Os três passos da execução do algoritmo (seguindo o paradigma dividir-para-conquistar) podem ser definidos como segue: (Cont.)
- Conquistar: Ordene os subvetores  $A[p..q-1]$  e  $A[q+1..r]$  através de chamadas recursivas ao algoritmo *Quicksort*.
- Combinação: Nem um trabalho adicional é necessário para a combinação dos subvetores, tendo em vista que os mesmos já foram previamente ordenados.

# Quicksort

- Para os exemplos a seguir, não haverá necessidade da definição de novas estruturas básicas para a execução do *Quicksort*, pois precisamos apenas de um vetor numérico, que será representado por um **Array**<inteiro>.
- Para os problemas de ordenação numérica, será permitida a inserção de valores iguais no vetor (**Array**<inteiro>) a ser ordenado.
- No exemplo resolvido:
  - Todas as posições do vetor a ser ordenado já estarão preenchidas.
  - A chamada ao procedimento *Quicksort* deve ser feita com valor de  $p$  igual à primeira posição válida do vetor, e  $r$  igual à última posição válida do mesmo (ou seja, seu tamanho).

# Quicksort

```
1. //A -> Array<inteiro> contendo os dados
2. //p -> índice da posição mais à esquerda considerada
3. //r -> índice da posição mais à direita considerada
4. procedimento quicksort(A, p, r)
5.     se p < r
6.         q = particionar(A, p, r)
7.         quicksort(A, p, q - 1)
8.         quicksort(A, q + 1, r)
```

```
1. //A -> Array<inteiro> contendo os dados
2. //p -> índice da posição mais à esquerda considerada
3. //r -> índice da posição mais à direita considerada
4. procedimento particionar(A, p, r)
5.     x = A[r]
6.     i = p - 1
7.     para j = p até r - 1
8.         se A[j] <= x
9.             i = i + 1
10.            trocar(A[i], A[j]) //intercambia o conteúdo das posições do vetor
11.            trocar(A[i + 1], A[r])
12.            retorne i + 1
```

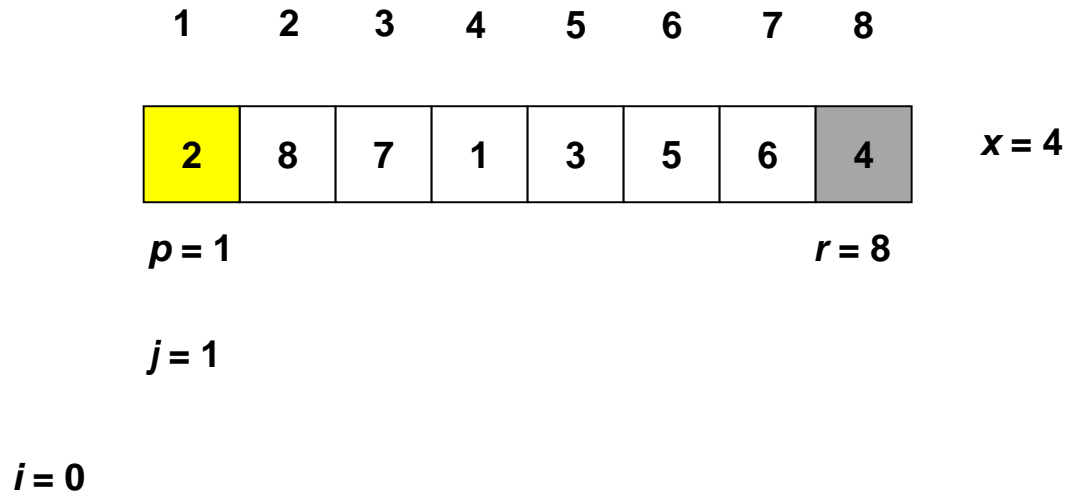
# Quicksort

- Exemplo: ordenar o vetor  $A = \{2, 8, 7, 1, 3, 5, 6, 4\}$  usando o algoritmo *Quicksort* com **pivô igual ao último elemento de cada partição.**

1	2	3	4	5	6	7	8
2	8	7	1	3	5	6	4

# Quicksort

- Particionar(A, 1, 8):

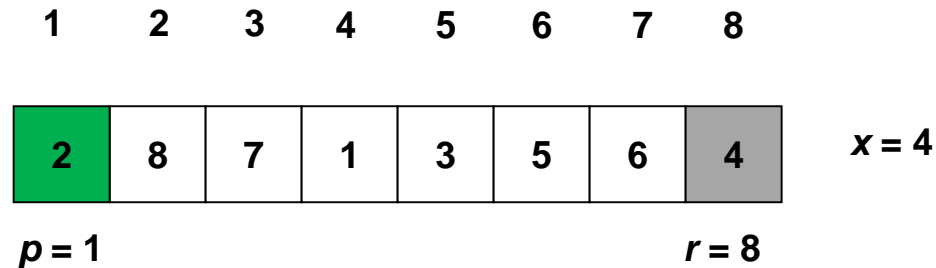


$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 1$

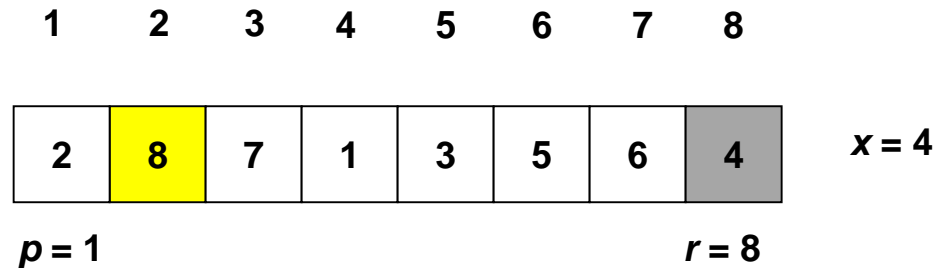
$i = 1$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 2$

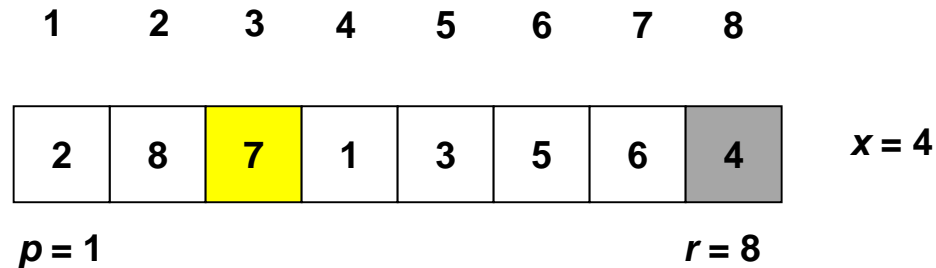
$i = 1$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 3$

$i = 1$

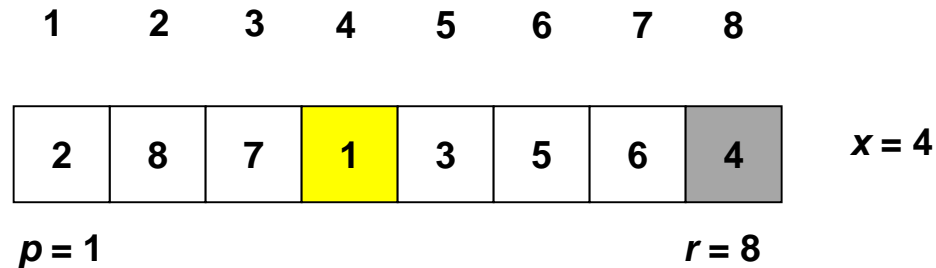
$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )



# Quicksort

- Particionar(A, 1, 8):



$j = 4$

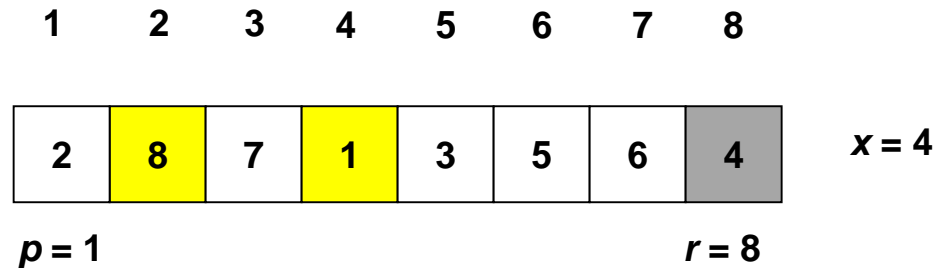
$i = 1$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 4$

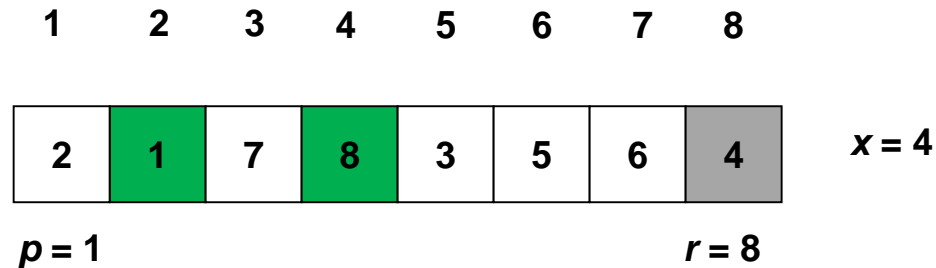
$i = 2$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 4$

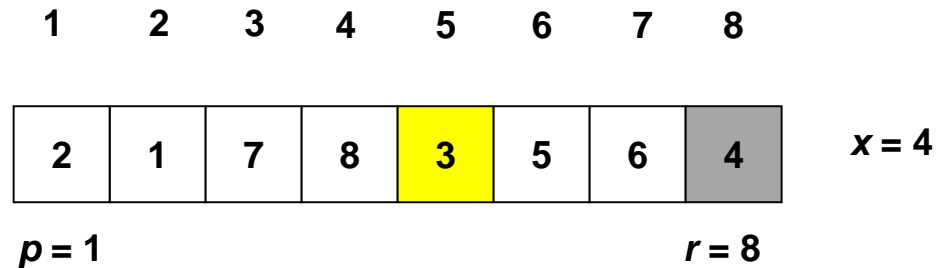
$i = 2$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 5$

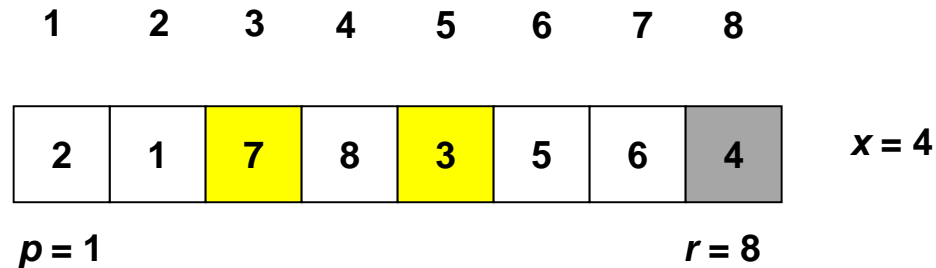
$i = 2$

$A[j] \leq x?$

$trocar(A[i], A[j])$

# Quicksort

- Particionar(A, 1, 8):



$j = 5$

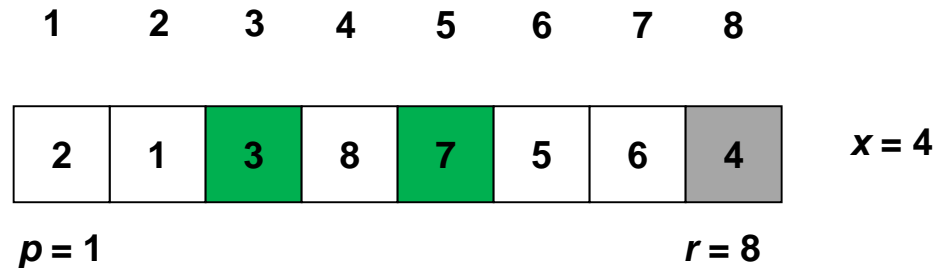
$i = 3$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 5$

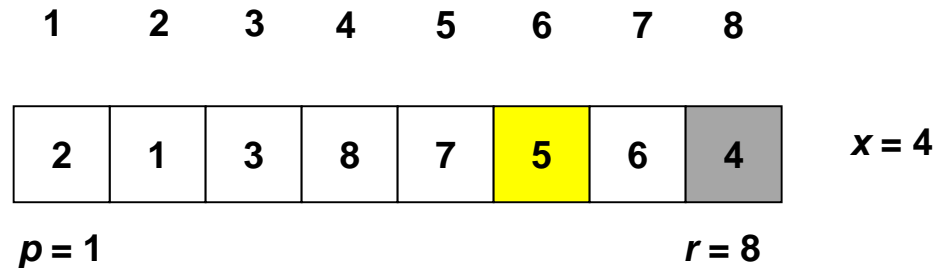
$i = 3$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 6$

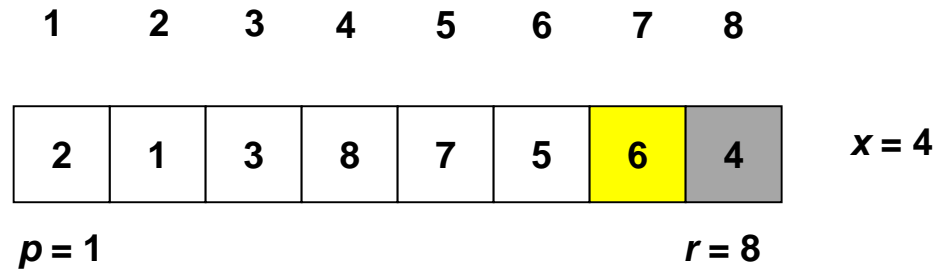
$i = 3$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 8):



$j = 7$

$i = 3$

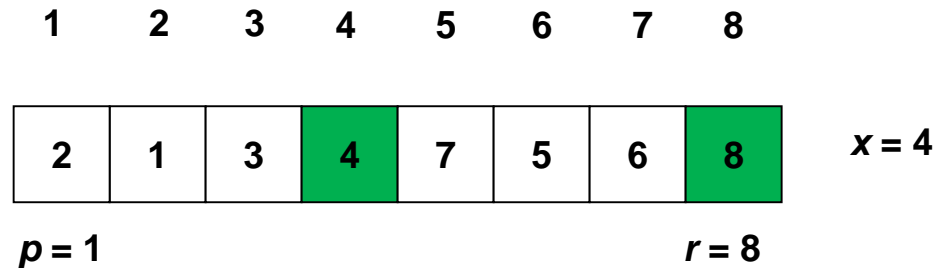
$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )



# Quicksort

- Particionar(A, 1, 8):



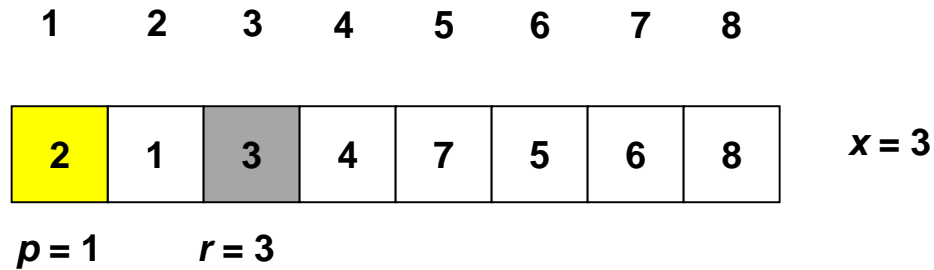
$i = 3$

*trocar*(A[i+1], A[r])

*retornar*  $i + 1$

# Quicksort

- Particionar(A, 1, 3):



$j = 1$

$i = 0$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 3):

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8

$x = 3$

$p = 1$

$r = 3$

$j = 1$

$i = 1$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 3):

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8

$x = 3$

$p = 1$

$r = 3$

$j = 1$

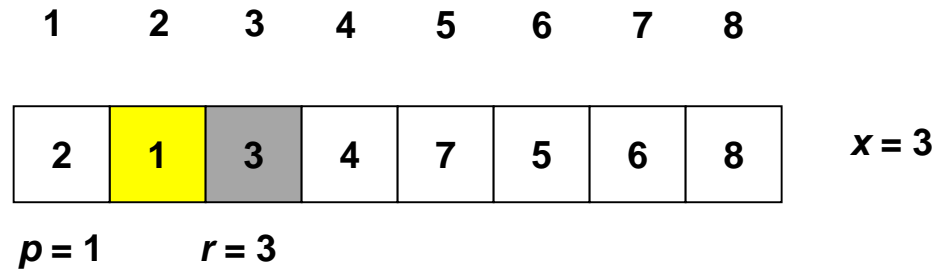
$i = 1$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 3):



$j = 2$

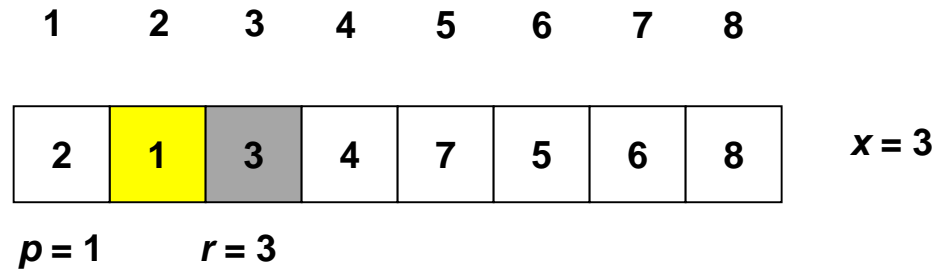
$i = 1$

$A[j] \leq x?$

*trocar*(A[i], A[j])

# Quicksort

- Particionar(A, 1, 3):



$j = 2$

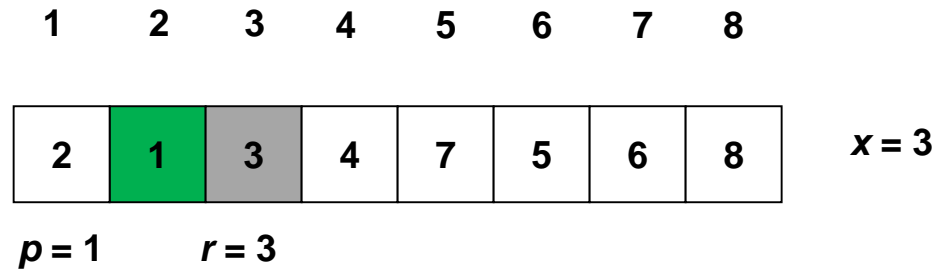
$i = 2$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 3):



$j = 2$

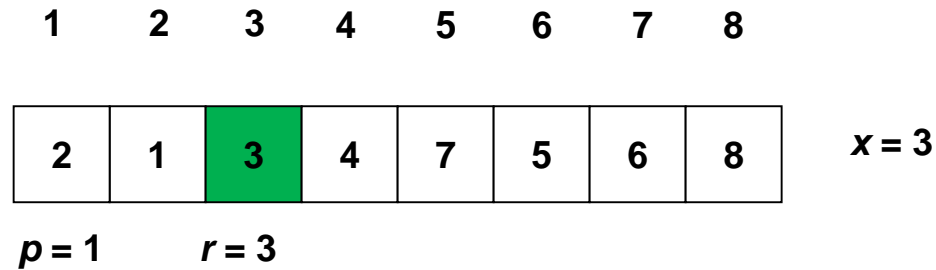
$i = 2$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 3):



$i = 2$

*trocar*(A[i+1], A[r])

*retornar*  $i + 1$



# Quicksort

- Particionar(A, 1, 2):

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8

$x = 1$

$p = 1$   $r = 2$

$j = 1$

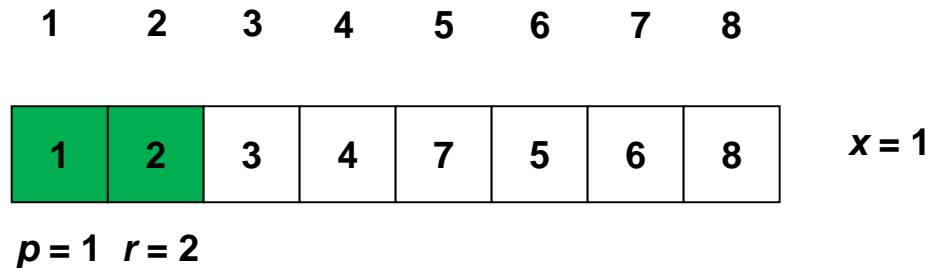
$i = 0$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 1, 2):



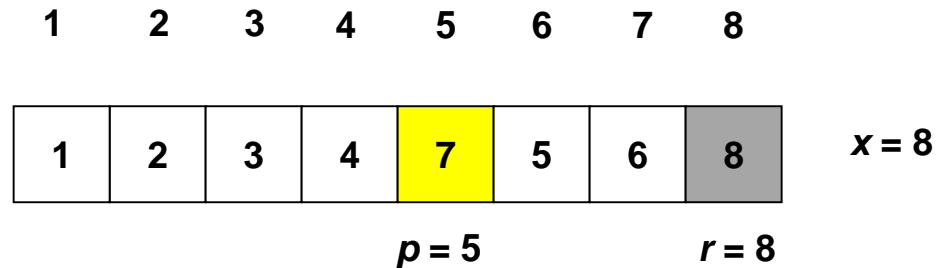
$i = 0$

*trocar*(A[i+1], A[r])

*retornar*  $i + 1$

# Quicksort

- Particionar(A, 5, 8):



$j = 5$

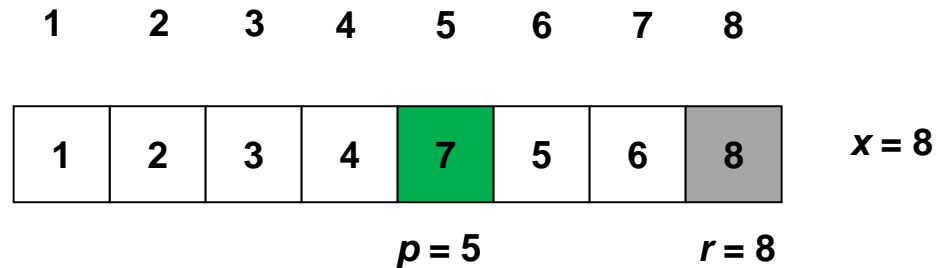
$i = 4$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 5, 8):



$j = 5$

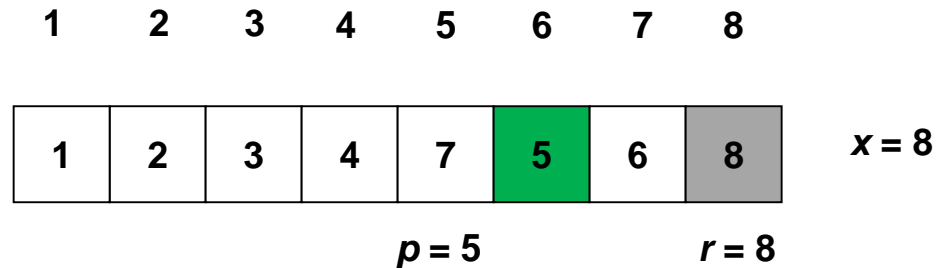
$i = 5$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 5, 8):



$j = 6$

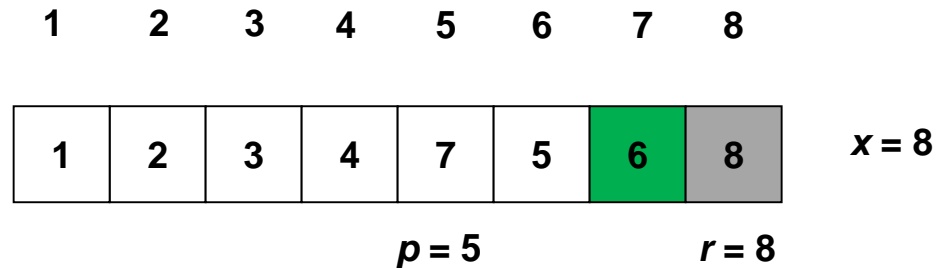
$i = 6$

$A[j] \leq x?$

*trocar*(A[i], A[j])

# Quicksort

- Particionar(A, 5, 8):



$j = 7$

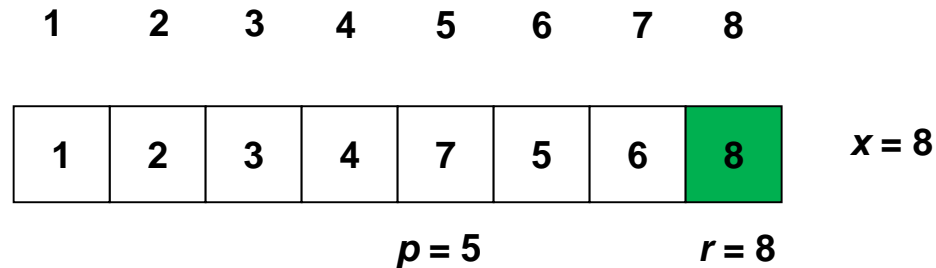
$i = 7$

$A[j] \leq x?$

*trocar*(A[i], A[j])

# Quicksort

- Particionar(A, 5, 8):



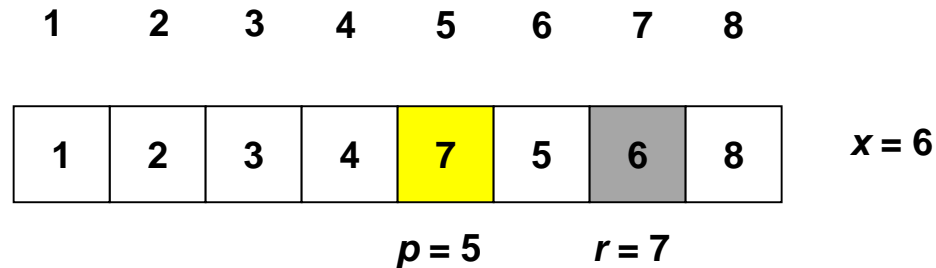
$i = 7$

*trocar*(A[i+1], A[r])

*retornar*  $i + 1$

# Quicksort

- Particionar(A, 5, 7):



$j = 5$

$i = 4$

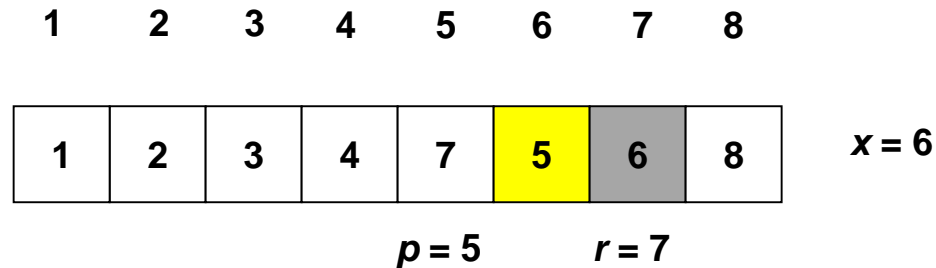
$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )



# Quicksort

- Particionar(A, 5, 7):



$j = 6$

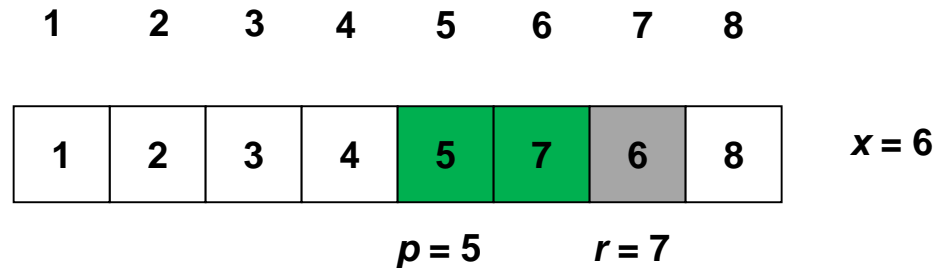
$i = 4$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 5, 7):



$j = 6$

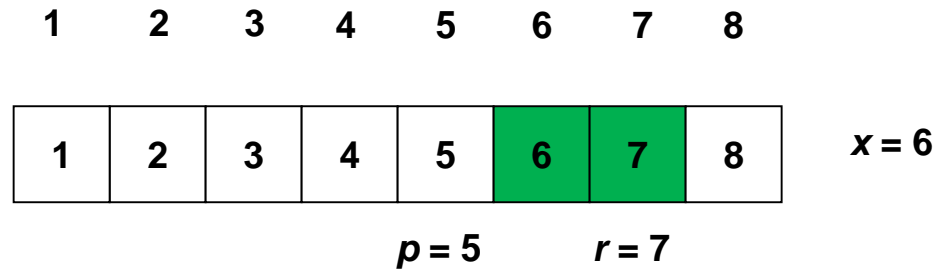
$i = 5$

$A[j] \leq x?$

*trocar*( $A[i]$ ,  $A[j]$ )

# Quicksort

- Particionar(A, 5, 7):



$i = 5$

*trocar*(A[i+1], A[r])

*retornar*  $i + 1$

# *Quicksort*

- Não há mais nenhuma chamada recursiva.

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

# *Quicksort*

- Considerações de implementação:
  - Quando a quantidade de elementos é pequena, é melhor usar um outro algoritmo na chamada recursiva.
    - Na verdade, quando  $n$  é pequeno, quanto mais simples melhor.

# *Quicksort*

- Considerações de implementação: (Cont.)
  - Como a escolha do pivô é crucial para um bom desempenho do algoritmo, as implementações usam métodos mais sofisticados para a sua escolha, sendo os mais populares:
    - Mediana ( $A[p]$ ,  $A[(p+r)/2]$ ,  $A[r]$ )
    - Aleatório ( $p..r$ )

# Quicksort

- Complexidade: Caso desfavorável
- Se a posição do pivô no vetor ordenado está sempre ou muito próximo do início ou muito próximo do final, então o tempo de execução de *Quicksort* é alto.
- Por exemplo, se o vetor de entrada está ordenado, o pivô é sempre o menor elemento da sequência então:
  - A primeira partição gasta  $(n-1)$  comparações e deixa  $n-1$  chaves por serem ordenadas.

$$\begin{aligned} Q(n) &= (n-1) + (n-2) + (n-3) + \dots + 1 \\ &= n(n-1) / 2 = \Theta(n^2) \end{aligned}$$

# Quicksort

- Complexidade: Caso favorável
- Designaremos por  $Q(n)$  o número de comparações de chaves que *Quicksort* faz para uma entrada de tamanho  $n$ .
- 
- Se o pivô sempre particionasse as chaves em partes iguais, então o número de comparações seria aproximadamente:

$$(1 \times n) + (2 \times n/2) + (4 \times n/4) + (8 \times n/8) + \dots + (n \times 1)$$

- E a relação de recorrência seria:

$$Q(n) = 2Q(n/2) + O(n), \quad Q(1) = 1$$

- Neste caso, já sabemos (*Mergesort*) que  $Q(n) = \Theta(n \log n)$ .



# Referências

- CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms, 3rd ed., *Boston: MIT Press*, 2009.
- FEOFILOFF, Paulo. Algoritmos em Linguagem C. Editora Campus/Elsevier, 2009.

# *Quicksort*

Algoritmos e Estruturas de Dados  
Prof. Dr. Luciano Demétrio Santos Pacífico  
{luciano.pacifico@ufrpe.br}



UNIVERSIDADE  
FEDERAL RURAL  
DE PERNAMBUCO