5. Algoritmos de Ordenação

Prof. Renato Tinós

Depto. de Computação e Matemática (FFCLRP/USP)

Principais Tópicos

5.1. Ordenação por Inserção
5.2. Ordenação por Seleção
5.3. Método da Bolha
5.4. Ordenação por Fusão
5.5. Heapsort
5.6. Quicksort
    5.6.1. Algoritmo de Partição
    5.6.2. Ordenação por Quicksort
5.7. Considerações sobre o Problema de Ordenação
5.8. Ordenação em Tempo Linear

5.6. Quicksort

- Desenvolvido por C. A. R. Hoare em 1960
- É um algoritmo mais rápido, em média, que se conhece para diversas categorias de problemas
  - É recursivo, mas os fatores constantes são pequenos quando comparado com outros métodos
- Como outros algoritmos de ordenação, existem versões diferentes
- Princípios
  - Dividir para conquistar
    - Dividir o problema de ordenação de *N* registros em problemas menores e então combinar as soluções parciais
  - Permutações devem ser preferencialmente empregadas para pares de elementos que guardem entre si distâncias grandes, com a finalidade de se conseguir uma maior eficiência

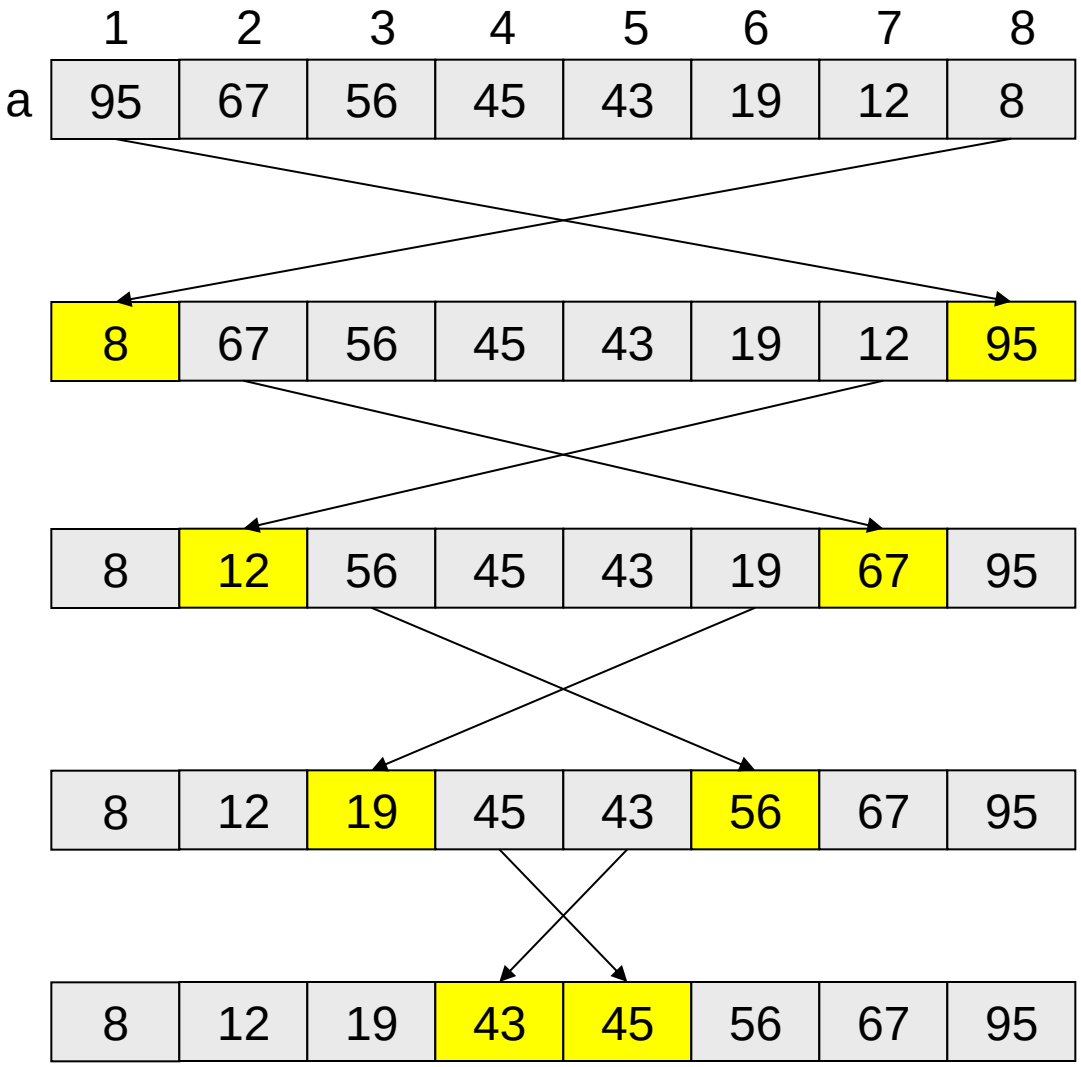
5.6. Quicksort

- Se os *N* registros estão na ordem inversa de suas chaves, é possível ordená-los com apenas *N*/2 permutações tomando-se primeiramente os elementos das extremidades à direita e à esquerda e convergindo gradualmente para o centro, pelos dois lados

- Obviamente, isto é possível se os elementos estiverem exatamente na ordem inversa

5.6. Quicksort

Exemplo 5.6.1.



5.6. Quicksort

Exercício 5.6.1. Escreva um algoritmo para inverter a ordem dos elementos de um vetor de *N* elementos

5.6. Quicksort

Exercício 5.6.1. Escreva um algoritmo para inverter a ordem dos elementos de um vetor de *N* elementos

```
...
para i ← 1 até i ← N/2
        x ← a[i]
        a[i] ← a[N-i+1]
        a[N-i+1]← x
fim para
...
```

5.6.1. Algoritmo de Partição

- Algoritmo de partição
  - escolha-se arbitrariamente um elemento *x* do vetor **a**
  - o vetor é varrido da esquerda para a direita, até que seja encontrado um elemento *a[i] > x*
  - após isso, o vetor é varrido da direita para a esquerda até que seja encontrado um elemento *a[j] < x*
  - nesta ocasião, os dois elementos serão permutados, e este processo de *varredura e de permutação* continua até que os dois deslocamentos se encontrem em algum ponto intermediário do vetor

- O resultado desta prática é um vetor particionado, no qual a partição esquerda contém apenas chaves cujos valores são menores (ou iguais) a *x* e a partição direita, apenas chaves cujos valores são maiores (ou iguais) a *x*

5.6.1. Algoritmo de Partição

- Seja $x = 43$
  - 45 56 12 43 95 8 19 67
  - 45 56 12 43 95 8 19 67
  - 19 56 12 43 95 8 45 67
  - 19 56 12 43 95 8 45 67
  - 19 8 12 43 95 56 45 67

Chaves menores ou iguais a **x**

Chaves maiores ou iguais a **x**

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
...
```

O(N)

O(1)

Portanto, o algoritmo de Partição é O(N)

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

	i							j
	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

	i						j	
	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

	i						j	
	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

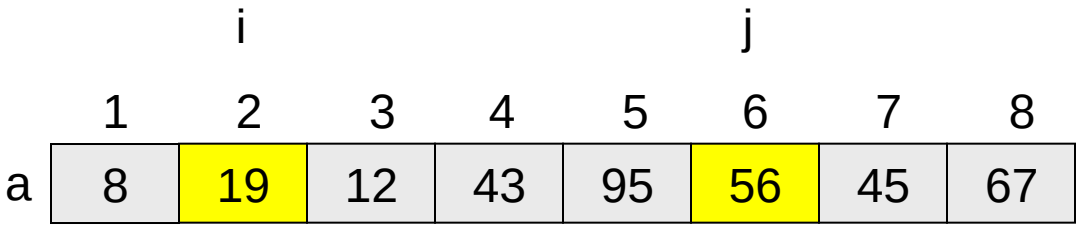
		i					j		
	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8



5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

			i			j		
	1	2	3	4	5	6	7	8
a	8	19	12	43	95	56	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

				i		j		
	1	2	3	4	5	6	7	8
a	8	19	12	43	95	56	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

j

i

	1	2	3	4	5	6	7	8
a	8	19	12	43	95	56	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

j

i

	1	2	3	4	5	6	7	8
a	8	19	12	43	95	56	45	67

5.6.1. Algoritmo de Partição

```
...
i ← 1
j ← N
x ← selecionar um elemento
      aleatoriamente do vetor a
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
```

Exemplo 5.6.2. Partição para $x = 43$

x = 43 N = 8

			j		i			
	1	2	3	4	5	6	7	8
a	8	19	12	43	95	56	45	67

5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Particione o vetor abaixo utilizando o algoritmo descrito para

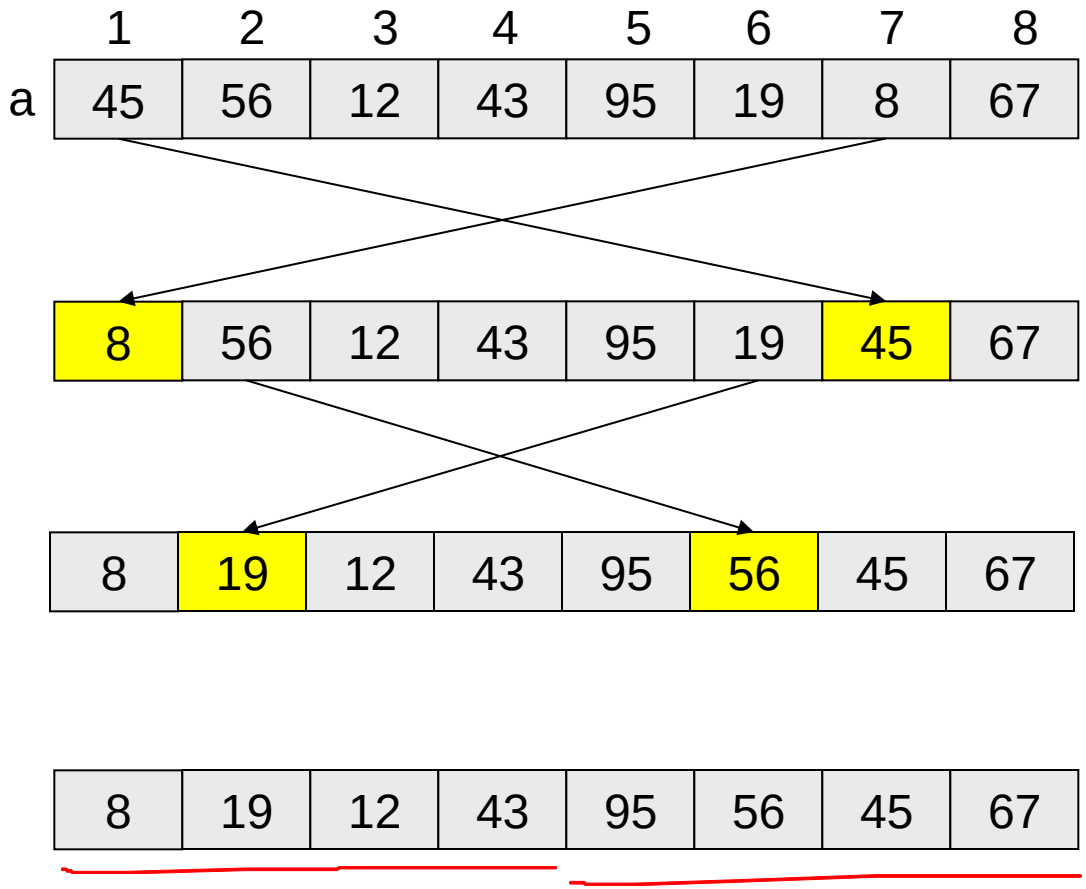
- $x = 45$
- $x = 56$
- $x = 12$
- $x = 95$
- $x = 19$
- $x = 8$
- $x = 67$

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

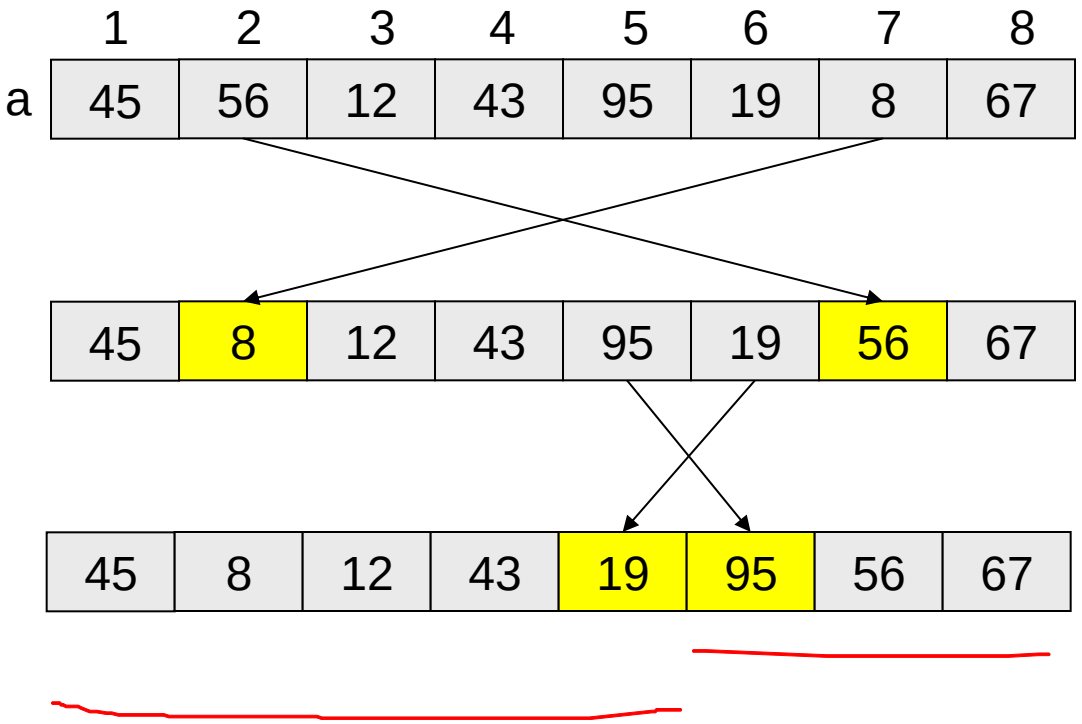
- $x = 45$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

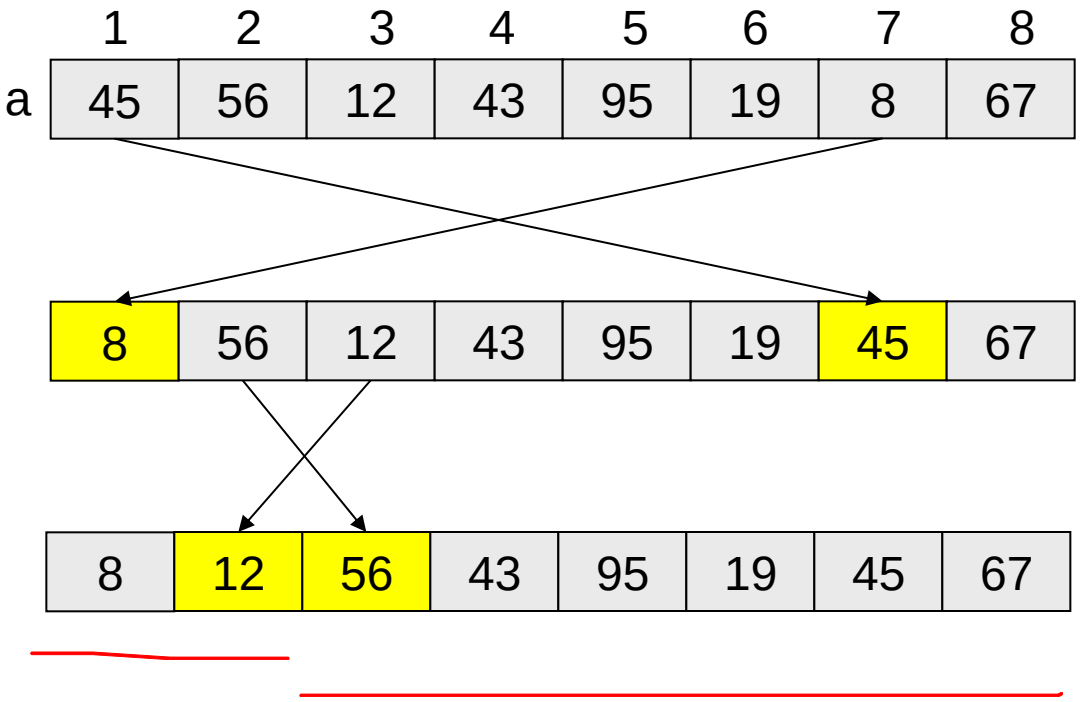
- $x = 56$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

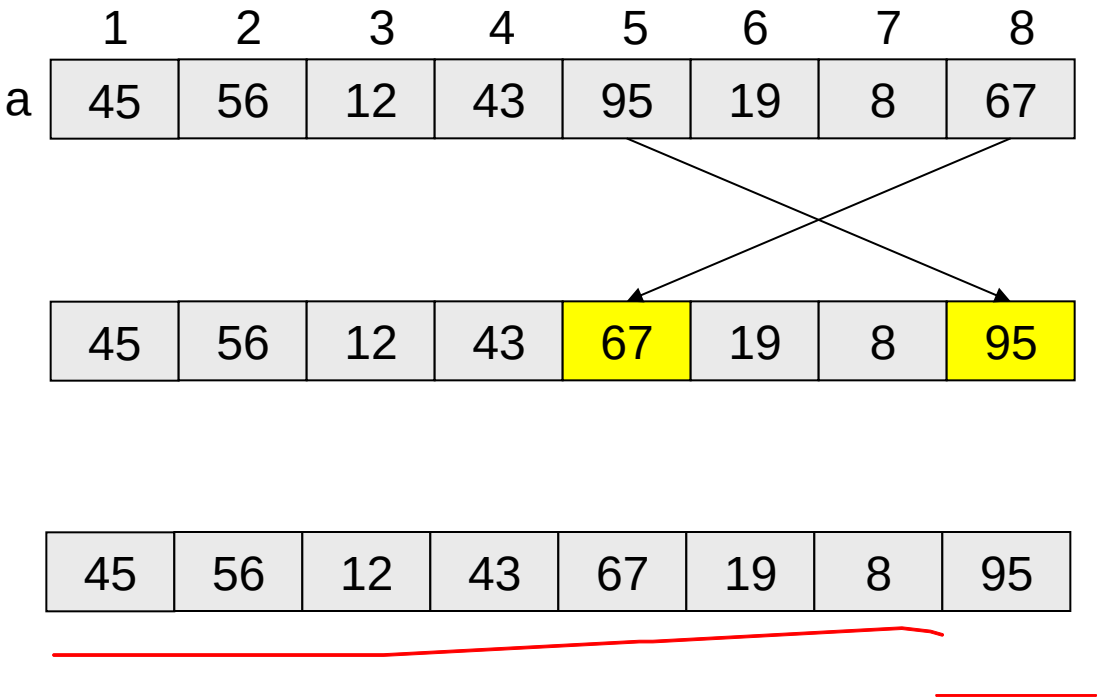
- $x = 12$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

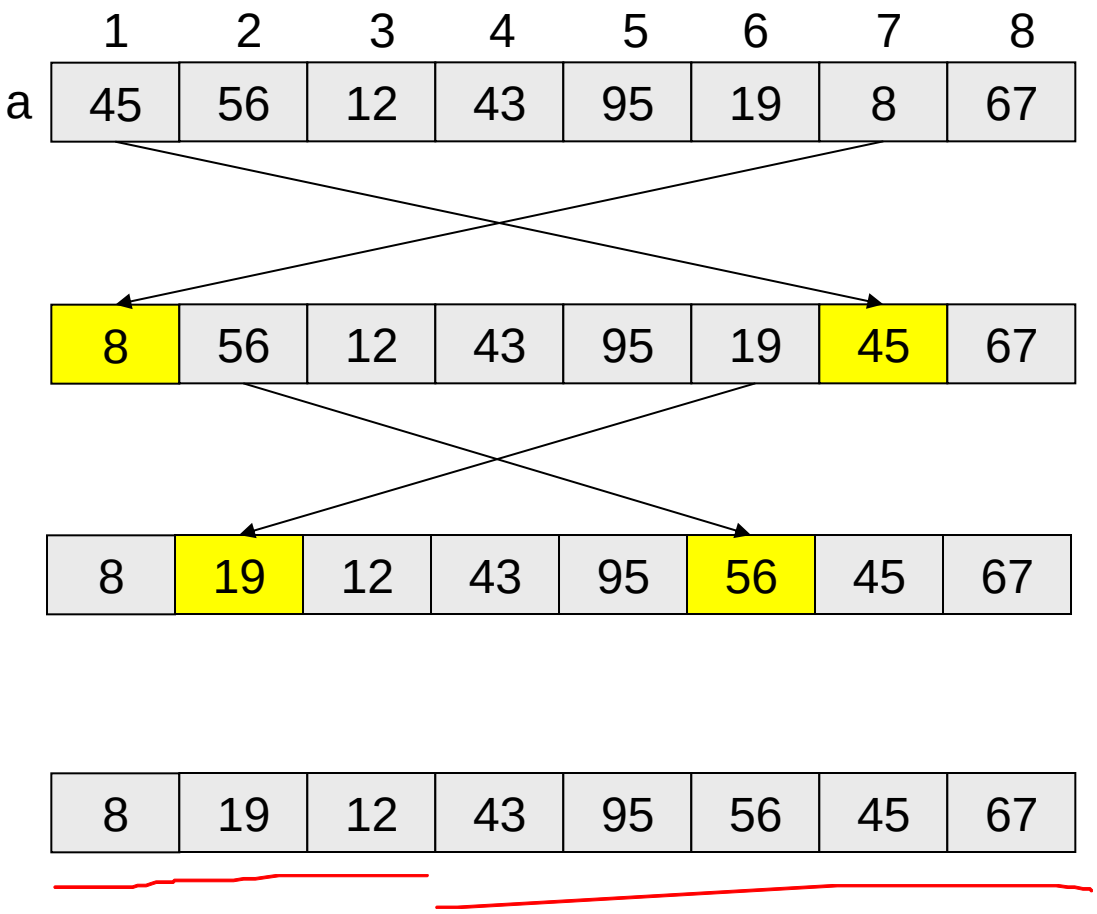
- $x = 95$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

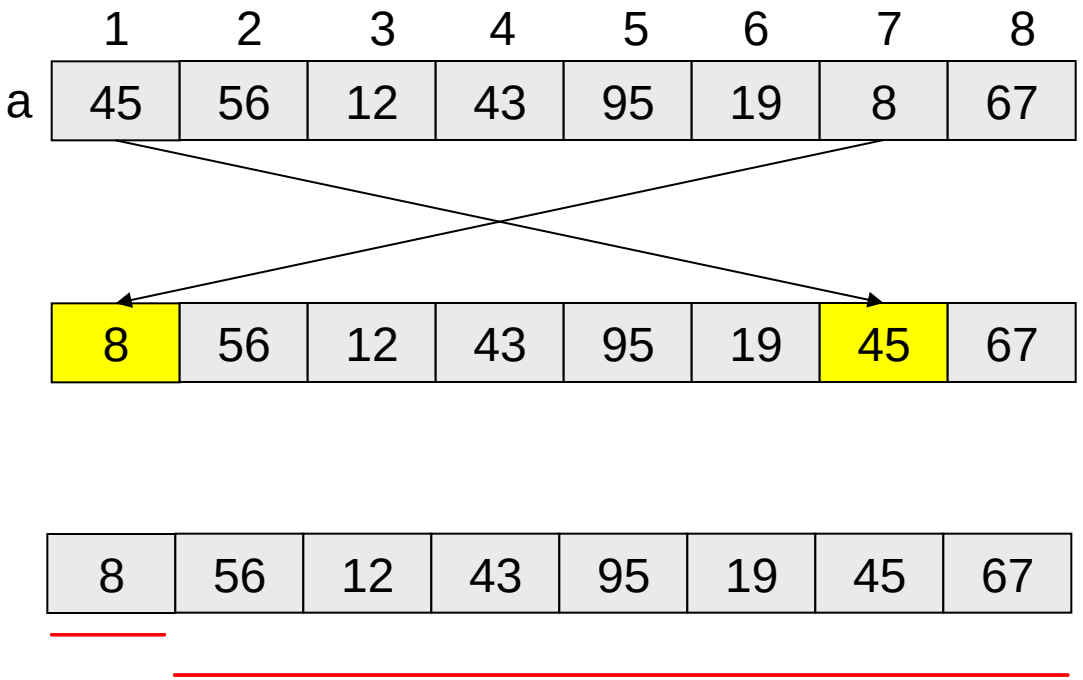
- $x = 19$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

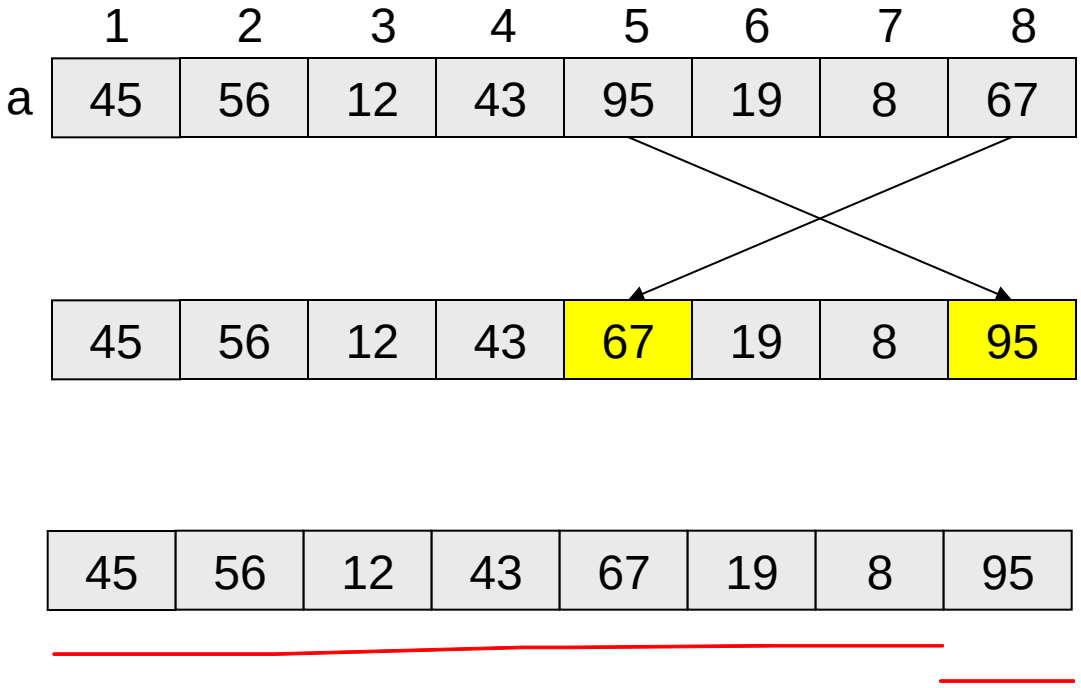
- $x = 8$



5.6.1. Algoritmo de Partição

Exemplo 5.6.2. Solução

- $x = 67$



5.6.1. Algoritmo de Partição

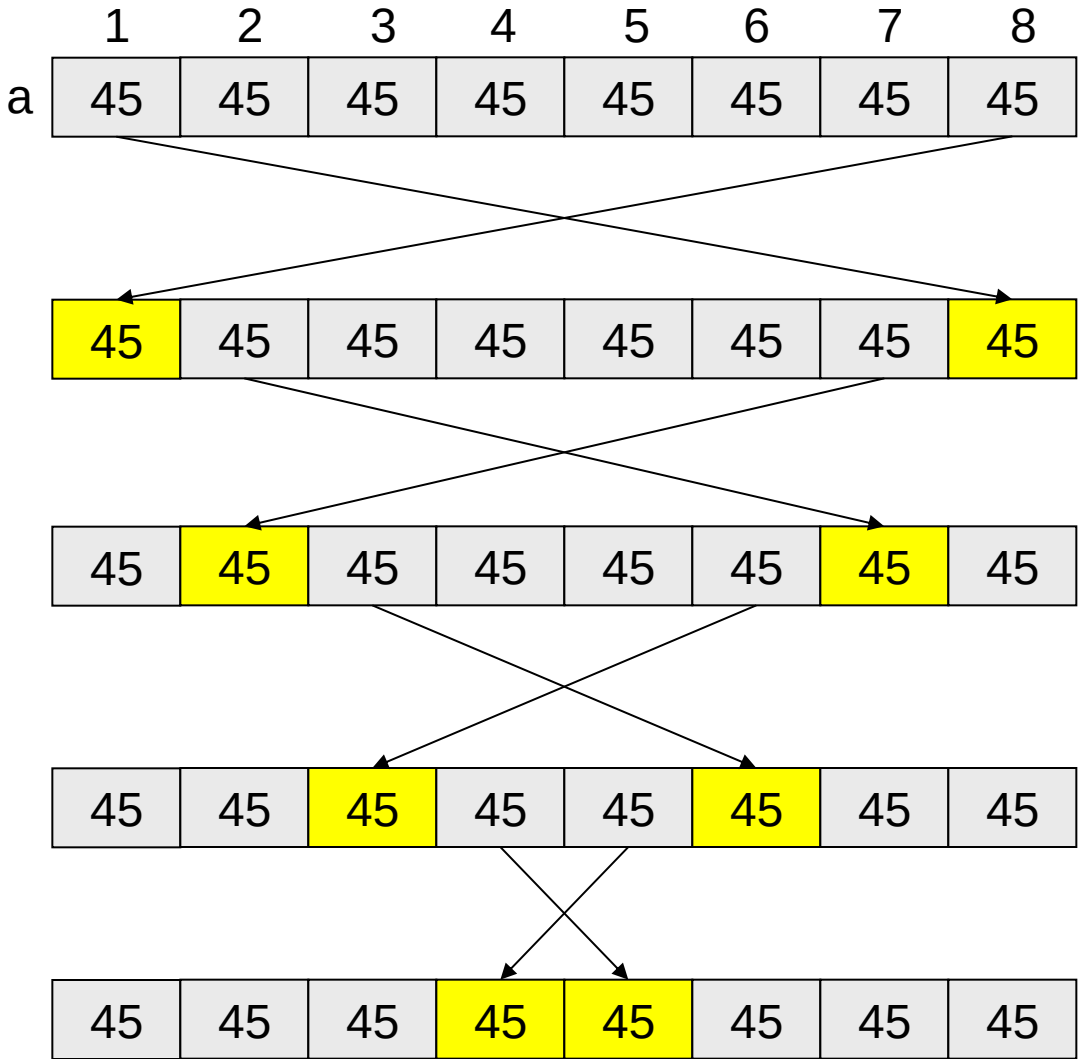
Exemplo 5.6.3. Particione o vetor abaixo utilizando o algoritmo descrito para

- $x = 45$

	1	2	3	4	5	6	7	8
a	45	45	45	45	45	45	45	45

5.6.1. Algoritmo de Partição

Exemplo 5.6.3. Solução



5.6.1. Algoritmo de Partição

- Este algoritmo é bastante direto e eficiente
  - Entretanto no caso das *N* chaves idênticas são necessárias *N/2* permutações
- Permutações desnecessárias podem ser eliminadas trocando-se os comandos de varredura para

```
enquanto ( a[ i ] ≤ x )
        i ← i + 1
fim enquanto
enquanto ( x ≤ a[ j ] )
        j ← j -1
fim enquanto
```

- Entretanto, um vetor que possuísse chaves iguais poderia provocar uma varredura para além da extensão do vetor,
  - a menos que venham a ser utilizadas condições de término mais complexas
- A simplicidade das condições empregadas no algoritmo padrão certamente compensam permutações extras, que dificilmente ocorrem de fato na média dos casos reais de aplicação

5.6.2. Ordenação por Quicksort

- É necessário lembrar que o objetivo almejado não é só o de encontrar partições do vetor original, mas também ordená-lo

- Entretanto é simples o passo que leva à ordenação a partir do particionamento
  - após ter sido particionado o vetor, aplica-se o mesmo processo para ambas as partições
  - em seguida, para as partições oriundas de cada uma das partições obtidas
  - e assim por diante, até que todas as partições consistam de apenas um único elemento.

5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
        i ← L
        j ← R
        x ← a [  floor( (L + R ) / 2 ) ]
        faça {
                enquanto ( a[ i ] < x )
                        i ← i + 1
                fim enquanto
                enquanto ( x < a[ j ] )
                        j ← j -1
                fim enquanto
                se ( i ≤ j )
                        w ← a[ i ]
                        a[ i ] ← a[ j ]
                        a[ j ] ← w
                        i ← i + 1
                        j ← j - 1
                fim se
        } enquanto ( i ≤ j )
        se ( L < j )
                qsort( a , L, j )
        fim se
        se ( i < R )
                qsort( a , i, R )
        fim se
```

```
Algoritmo quicksort( a[ ] , N )
        qsort( a , 1, N )
```

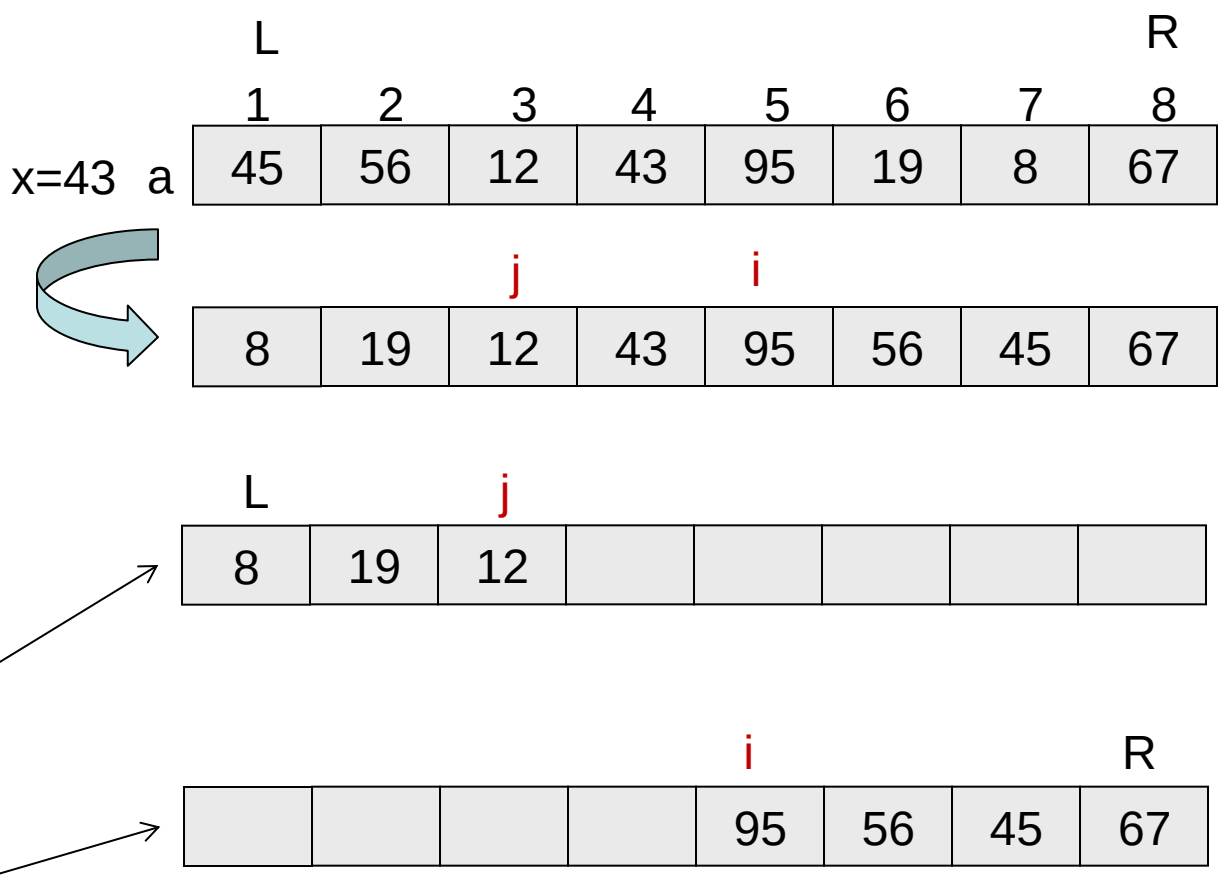
- Note que o procedimento *qsort* utiliza recursão

5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
i ← L
j ← R
x ← a [ floor( (L + R ) / 2 ) ]
faça {
        enquanto ( a[ i ] < x )
                i ← i + 1
        fim enquanto
        enquanto ( x < a[ j ] )
                j ← j -1
        fim enquanto
        se ( i ≤ j )
                w ← a[ i ]
                a[ i ] ← a[ j ]
                a[ j ] ← w
                i ← i + 1
                j ← j - 1
        fim se
} enquanto ( i ≤ j )
se ( L < j )
        qsort( a , L, j )
fim se
se ( i < R )
        qsort( a , i, R )
fim se
```

O(R-L)

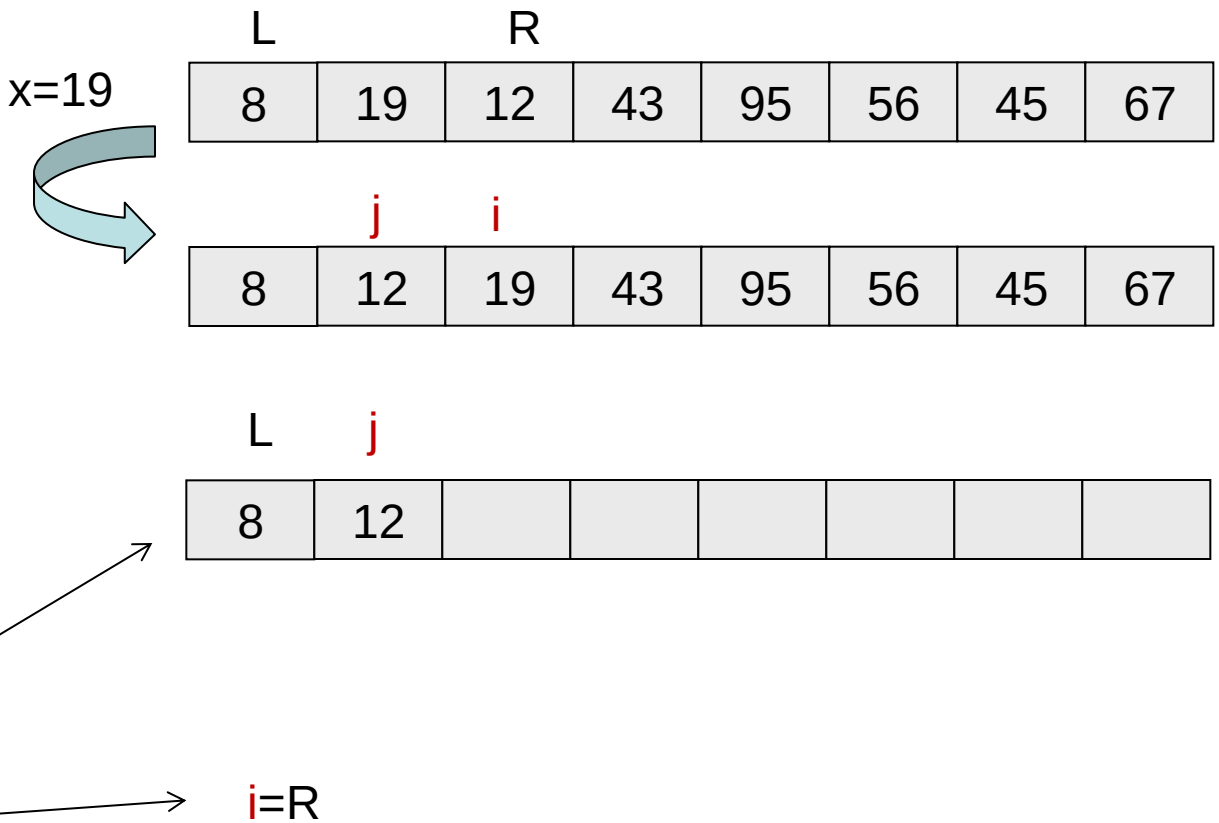
```
Algoritmo quicksort( a[ ] , N )
 qsort( a , 1, N )
```



5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
 i ← L
 j ← R
 x ← a [  floor( (L + R ) / 2 ) ]
 faça {
            enquanto ( a[ i ] < x )
                        i ← i + 1
            fim enquanto
            enquanto ( x < a[ j ] )
                        j ← j -1
            fim enquanto
            se ( i ≤ j )
                        w ← a[ i ]
                        a[ i ] ← a[ j ]
                        a[ j ] ← w
                        i ← i + 1
                        j ← j - 1
            fim se
 } enquanto ( i ≤ j )
 se ( L < j )
            qsort( a , L, j )
 fim se
 se ( i < R )
            qsort( a , i, R )
 fim se
```

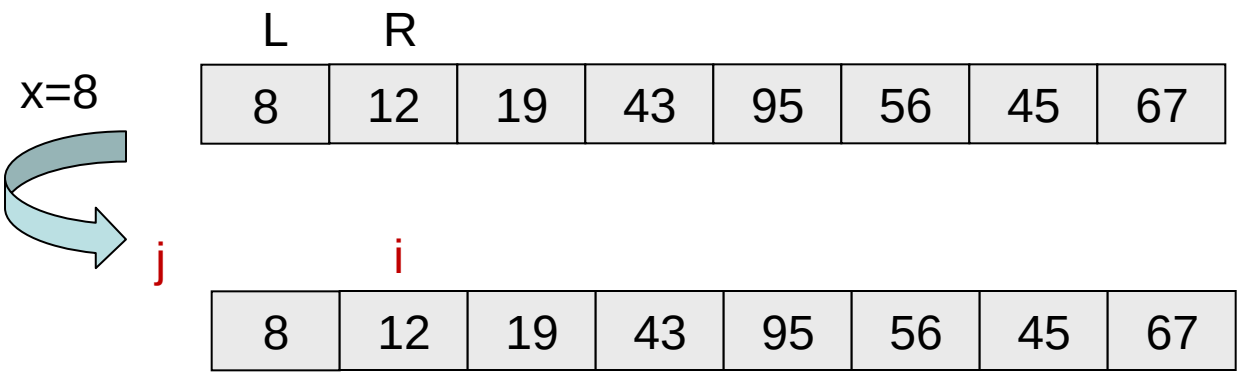
```
Algoritmo quicksort( a[ ] , N )
 qsort( a , 1, N )
```



5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
 i ← L
 j ← R
 x ← a [  floor( (L + R ) / 2 ) ]
 faça {
            enquanto ( a[ i ] < x )
                        i ← i + 1
            fim enquanto
            enquanto ( x < a[ j ] )
                        j ← j -1
            fim enquanto
            se ( i ≤ j )
                        w ← a[ i ]
                        a[ i ] ← a[ j ]
                        a[ j ] ← w
                        i ← i + 1
                        j ← j - 1
            fim se
 } enquanto ( i ≤ j )
 se ( L < j )
            qsort( a , L, j )
 fim se
 se ( i < R )
            qsort( a , i, R )
 fim se
```

```
Algoritmo quicksort( a[ ] , N )
 qsort( a , 1, N )
```



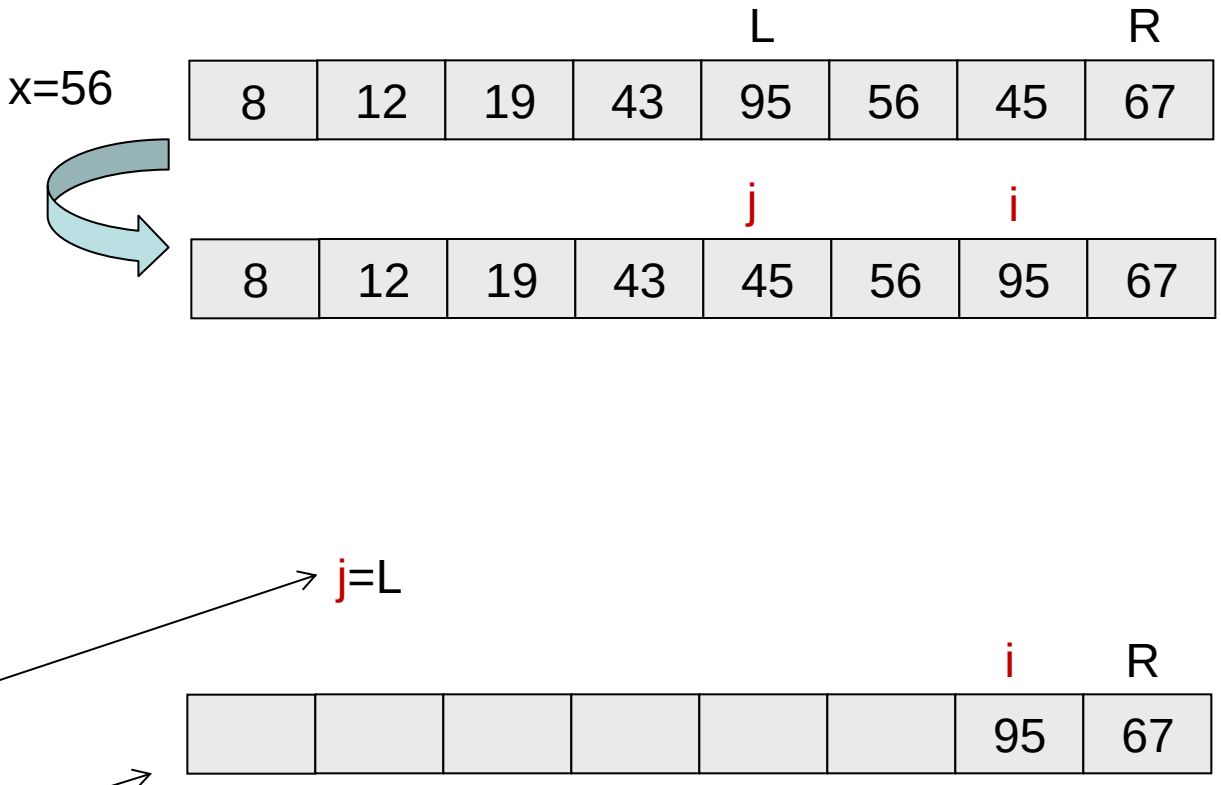
j<L

i=R

5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
 i ← L
 j ← R
 x ← a [  floor( (L + R ) / 2 ) ]
 faça {
            enquanto ( a[ i ] < x )
                        i ← i + 1
            fim enquanto
            enquanto ( x < a[ j ] )
                        j ← j -1
            fim enquanto
            se ( i ≤ j )
                        w ← a[ i ]
                        a[ i ] ← a[ j ]
                        a[ j ] ← w
                        i ← i + 1
                        j ← j - 1
            fim se
 } enquanto ( i ≤ j )
 se ( L < j )
            qsort( a , L, j )
 fim se
 se ( i < R )
            qsort( a , i, R )
 fim se
```

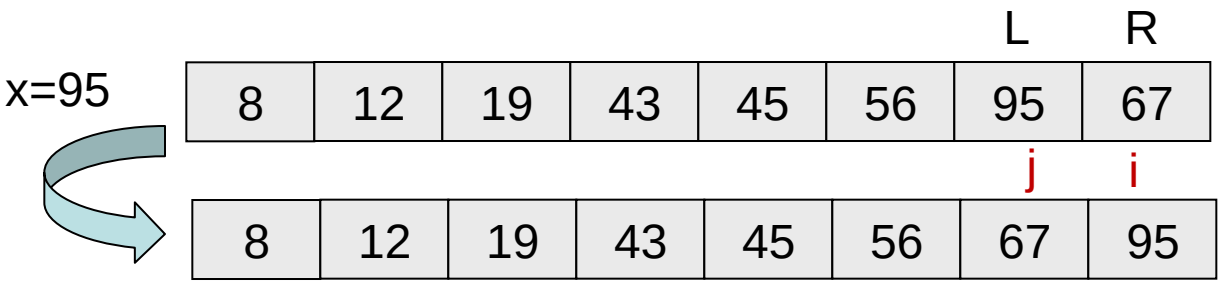
```
Algoritmo quicksort( a[ ] , N )
 qsort( a , 1, N )
```



5.6.2. Ordenação por Quicksort

```
Algoritmo qsort( a[ ] , L , R )
 i ← L
 j ← R
 x ← a [  floor( (L + R ) / 2 ) ]
 faça {
            enquanto ( a[ i ] < x )
                        i ← i + 1
            fim enquanto
            enquanto ( x < a[ j ] )
                        j ← j -1
            fim enquanto
            se ( i ≤ j )
                        w ← a[ i ]
                        a[ i ] ← a[ j ]
                        a[ j ] ← w
                        i ← i + 1
                        j ← j - 1
            fim se
 } enquanto ( i ≤ j )
 se ( L < j )
            qsort( a , L, j )
 fim se
 se ( i < R )
            qsort( a , i, R )
 fim se
```

```
Algoritmo quicksort( a[ ] , N )
 qsort( a , 1, N )
```



j=L

i=R

5.6.2. Ordenação por Quicksort

Quick-sort with Hungarian (Küküllőmenti legényes) folk dance

https://youtu.be/ywWBy6J5gz8?t=2

IMPORTANTE: O ALGORITMO DEMONSTRADO É UMA VARIAÇÃO DO APRESENTADO AQUI

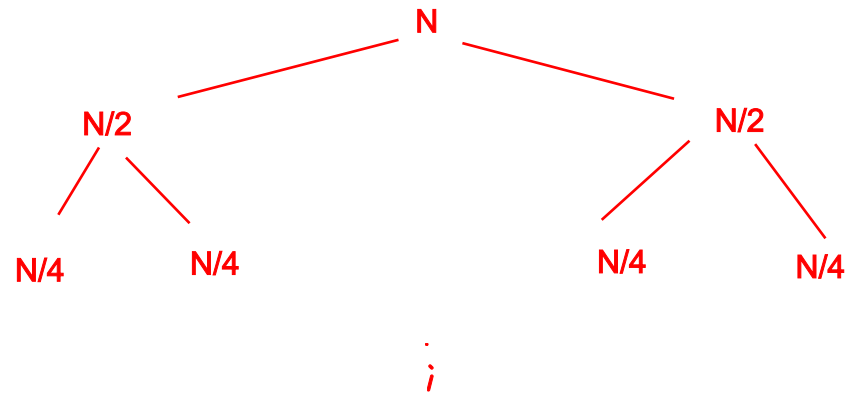
5.6.2. Ordenação por Quicksort

- O algoritmo é recursivo
  - A análise pode ser feita utilizando a árvore de recursão

- Melhor caso: $O(N \log(N))$
  - Admitindo-se que sempre ocorra o melhor caso (o limite escolhido é o ponto médio da partição), então em cada particionamento, divide-se os elementos analisados na recursão em duas metades, o que resulta em uma árvore com altura $O(\log(N))$
  - Os números de movimentações e comparações entre chaves em cada nível são proporcionais a N
    - Ou seja, $O(N)$ em cada nível
  - Portanto, a complexidade da ordenação será $O(N \log(N))$

5.6.2. Ordenação por Quicksort

Melhor Caso: sempre dividindo em 2
Árvore de recursão



Altura

Complexidade por nível

0 O(N)

1 O(N)

2 O(N)

h=O(log_2 N) O(N)

1

Soma: O(N * log_2 N)

5.6.2. Ordenação por Quicksort

- Caso Médio: O(*N* log(*N*))
  - Está complexidade também é válida para o caso médio, já que, em média, as divisões no vetor são equilibradas se considerarmos a distribuição uniforme de elementos
  - Mesmo considerando divisões bastante desequilibradas, ainda sim a complexidade é O(*N* log(*N*))
    - Exemplo:

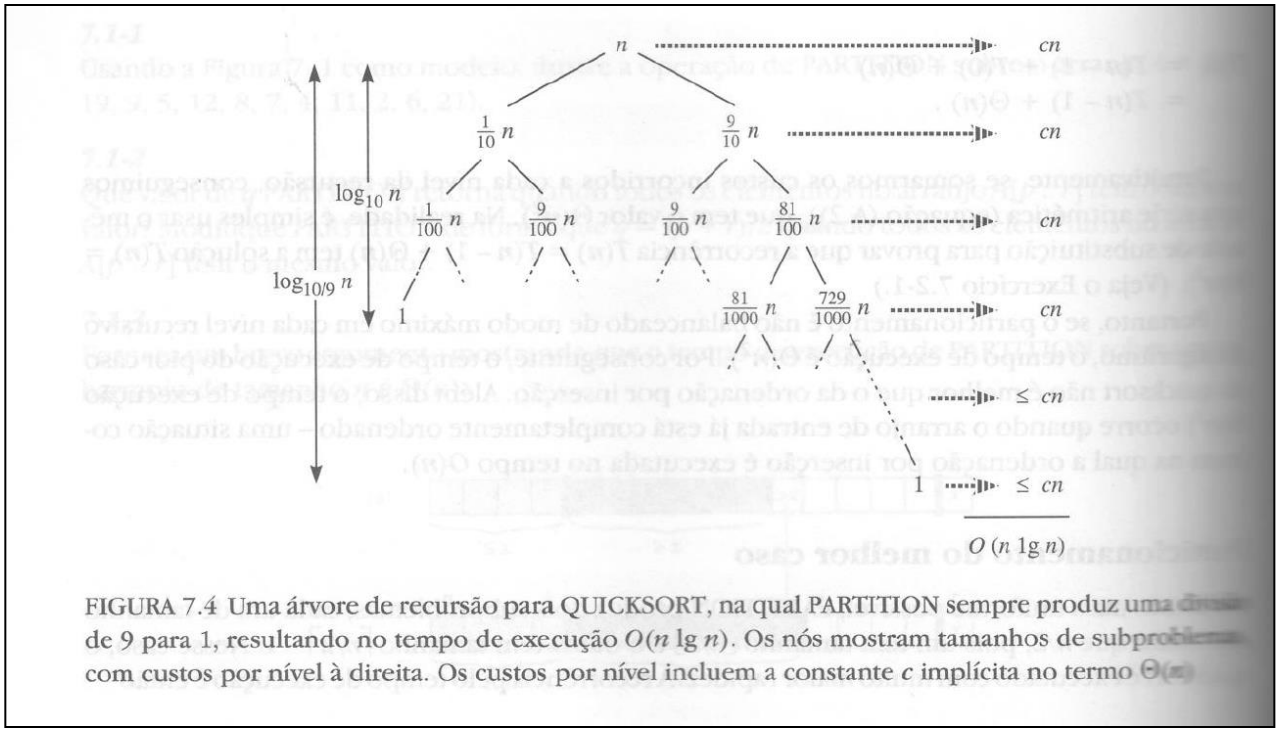


FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo $\Theta(n)$.

Lembre que:

$$\log_b a = \frac{\log_c a}{\log_c b}$$

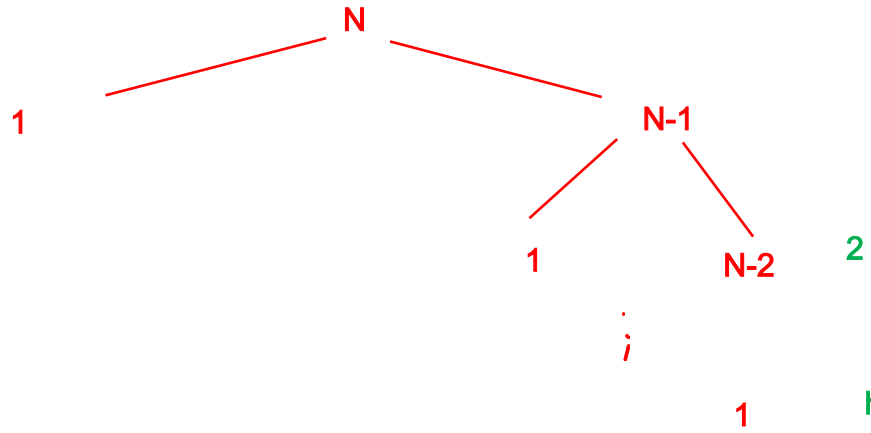
5.6.2. Ordenação por Quicksort

- **Pior Caso:** $O(N^2)$

  - Ocorre quando o elemento *x* de comparação é sempre o maior ou menor dos valores de uma partição.
  - Então, em cada passo, um segmento de *N* elementos será dividido em uma partição com *N-1* elementos e uma partição com um único elemento.
  - O resultado é que são necessários $O(N)$ (ao invés de $O(\log(N))$) divisões
    - Altura da árvore é $O(N)$
  - A complexidade neste caso é $O(N^2)$

5.6.2. Ordenação por Quicksort

Pior Caso: sempre dividindo e restando 1
Árvore de recursão



Altura	Complexidade por nível
0	O(N)
1	O(N)
2	O(N)
h=O(N)	O(N)

Soma: O(N^2)

5.6.2. Ordenação por Quicksort

Exercício 5.6.1. Utilizando ordenação pelo método quicksort, obtenha o número de comparações e movimentações em cada passo (*i* e *j*) para os seguintes vetores

- [45 56 12 43 95 19 8 67]
- [8 12 19 43 45 56 67 95]
- [95 67 56 45 43 19 12 8]
- [19 12 8 45 43 56 67 95]

5.6.2. Ordenação por Quicksort

Exercício 5.6.1. Solução

i	j	Ci	Mi	45	56	12	43	95	19	8	67
5	3	9	10	8	19	12	43	95	56	45	67
3	2	3	4	8	12	19	43	95	56	45	67
2	0	3	4	8	12	19	43	95	56	45	67
7	5	5	7	8	12	19	43	45	56	95	67
8	7	2	4	8	12	19	43	45	56	67	95
		22	29								

i	j	Ci	Mi	8	12	19	43	45	56	67	95
5	3	9	4	8	12	19	43	45	56	67	95
3	2 1	4	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		21	16								

i	j	Ci	Mi	19	12	8	45	43	56	67	95
5	4	8	4	19	12	8	43	45	56	67	95
3	1	5	7	8	12	19	43	45	56	67	95
4	2	3	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		24	23								

i	j	Ci	Mi	95	67	56	45	43	19	12	8
5	4	8	13	8	12	19	43	45	56	67	95
3	1	5	4	8	12	19	43	45	56	67	95
4	2	3	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		24	29								

Comentários

- **Agradecimentos**

  - Parte do material desta apresentação foi obtida através de slides da disciplina de Introdução à Computação II ministrada pelo Prof. José Augusto Baranauskas