CC BY

"Diga-me como me medes te direi como me comportarei [te engano]" (Goldratt, E.).

Quicksort

Paulo Ricardo Lisboa de Almeida

Particionar

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6			

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16		

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	1

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	1

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2

Teste de Mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
				4

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	42	23	8	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
				4

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	23	42	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
				4

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	23	42	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	23	42	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
				5

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	23	42	4	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
				5

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	4	42	23	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
				5

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	4	42	23	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
			4	5

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	4	42	23	16

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
			4	5
				6

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
			4	5
				6

Teste de Mesa

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

a	b	x	m	i
1	6	16	1	2
			2	3
			3	4
			4	5
				6

Pergunta

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

O que o algoritmo faz?

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Pergunta

i	1	2	3	**4**	5	6
v[i]	15	8	4	**16**	23	42

O que o algoritmo faz?

Dado o pivô, o algoritmo coloca todos os elementos

- Menores que o pivô à sua esquerda
- Maiores que o pivô à sua direita

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Faça você mesmo

i	1	2	3	4	5	6
v[i]	27	81	56	16	3	1

Execute o teste de mesa para o vetor acima

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Faça você mesmo

i	1	2	3	4	5	6
v[i]	27	81	56	16	3	1

Execute o teste de mesa para o vetor acima

i	1	2	3	4	5	6
v[i]	1	81	56	16	3	27

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Pergunta

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

O que o algoritmo faz?

Dado o pivô, o algoritmo coloca todos os elementos

- Menores que o pivô à sua esquerda
- Maiores que o pivô à sua direita

Como o algoritmo faz?

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Pergunta

i	1	2	3	**4**	5	6
v[i]	15	8	4	**16**	23	42

O que o algoritmo faz?

Dado o pivô, o algoritmo coloca todos os elementos

- Menores que o pivô à sua esquerda
- Maiores que o pivô à sua direita

Como o algoritmo faz?

Os elementos menores que o pivô são alocados nas posições de *a* até *m-1*

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Pergunta

i	1	2	3	**4**	5	6
v[i]	15	8	4	**16**	23	42

O que o algoritmo faz?

Dado o pivô, o algoritmo coloca todos os elementos

Menores que o pivô à sua esquerda

Maiores que o pivô à sua direita

Como o algoritmo faz?

Os elementos menores que o pivô são alocados nas posições de *a* até *m-1*

Para um vetor de tamanho *n*, quantas comparações com elementos do vetor são feitas?

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Pergunta

i	1	2	3	**4**	5	6
v[i]	15	8	4	**16**	23	42

O que o algoritmo faz?

Dado o pivô, o algoritmo coloca todos os elementos

Menores que o pivô à sua esquerda

Maiores que o pivô à sua direita

Como o algoritmo faz?

Os elementos menores que o pivô são alocados nas posições de *a* até *m-1*

Para um vetor de tamanho *n*, quantas comparações com elementos do vetor são feitas?

n-1

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

Quicksort

```
função quickSort(v,a,b)
entrada: vetor v, indexado por [a..b]
saída: o vetor v modificado de forma que v[a..b] é um vetor ordenado.
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

Teste de mesa

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

quickSort		
a	b	m
1	6	

i	1	2	3	4	5	6
v[i]	42	15	23	8	4	16

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```

quickSort		
a	b	m
1	6	4

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

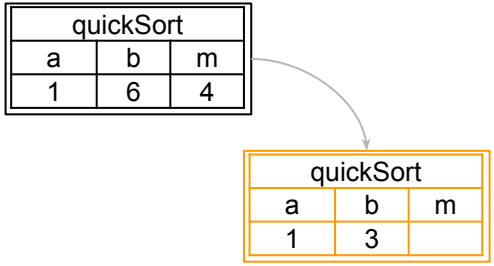
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```

quickSort		
a	b	m
1	6	4

i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

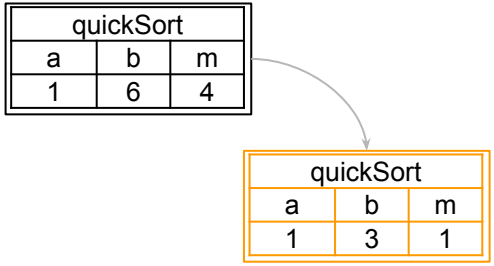
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	15	8	4	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

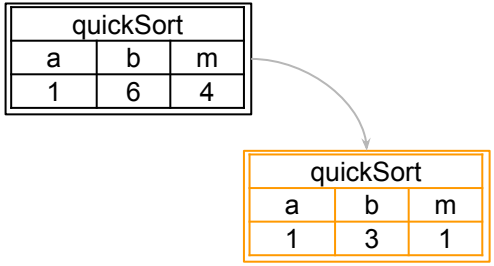
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

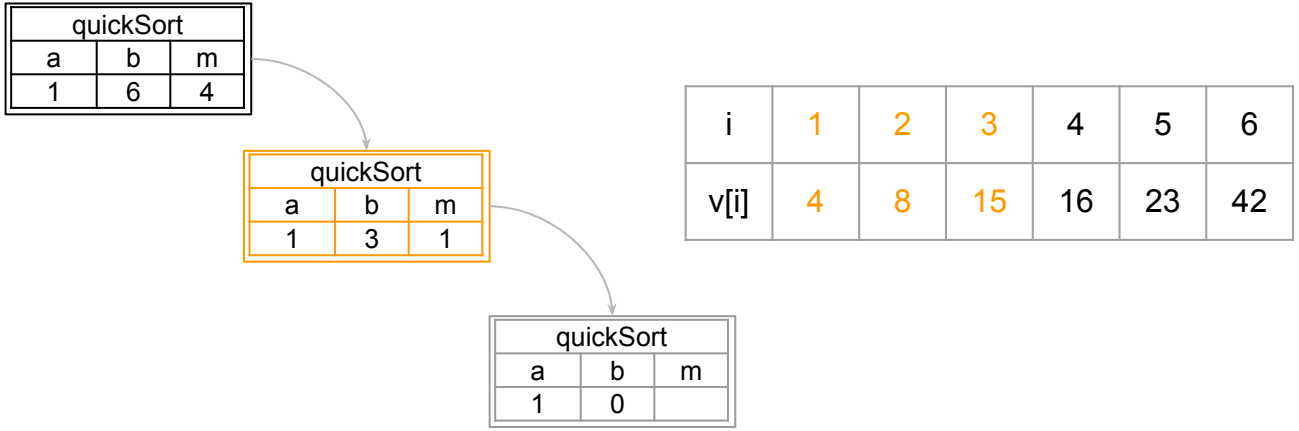
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

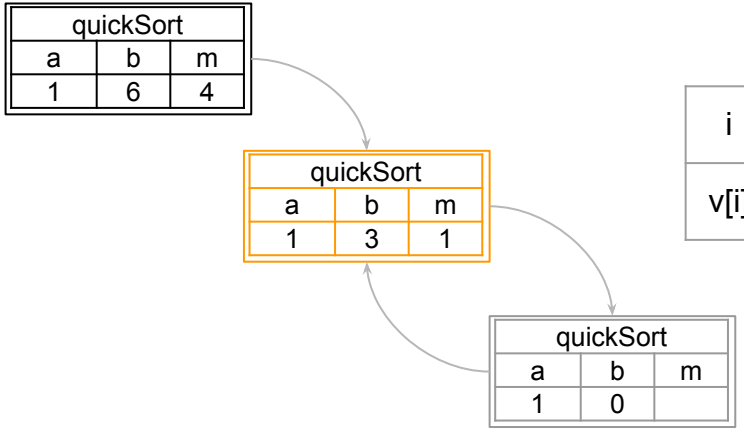
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

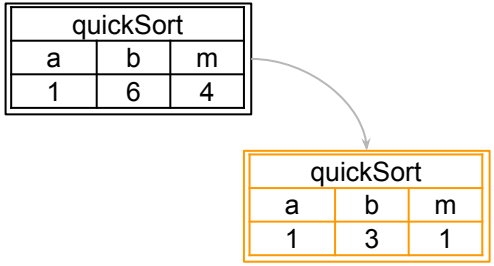
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
    retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

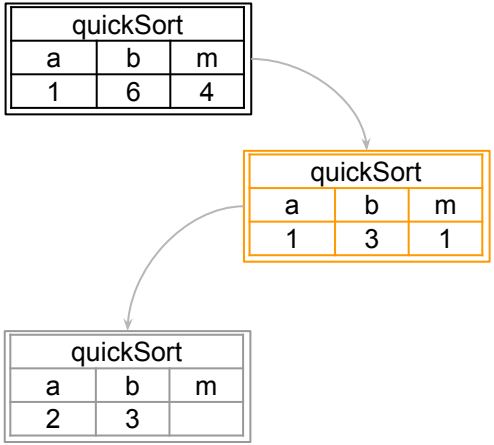
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

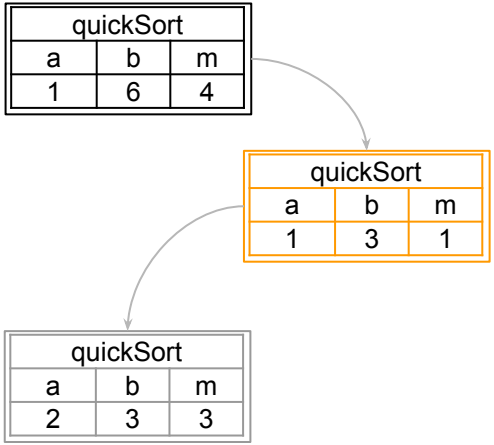
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

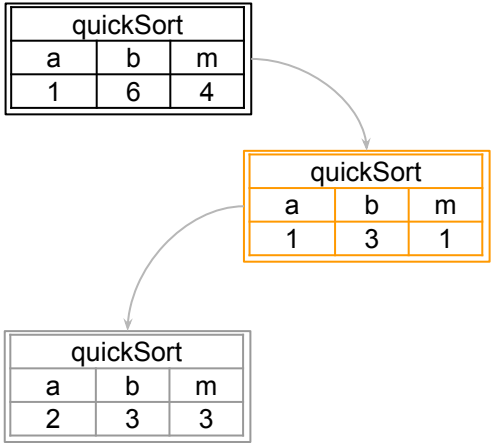
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

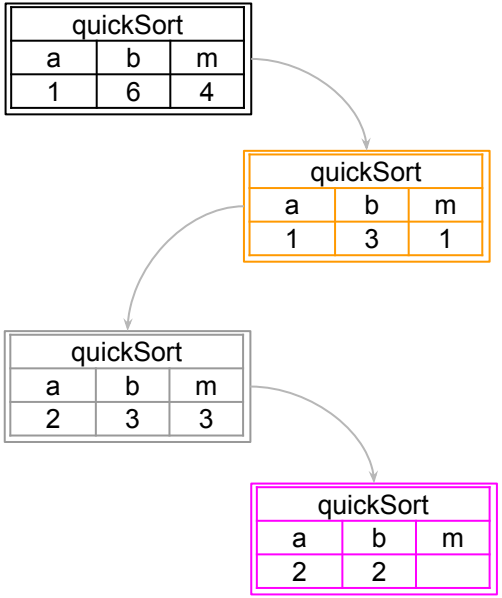
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

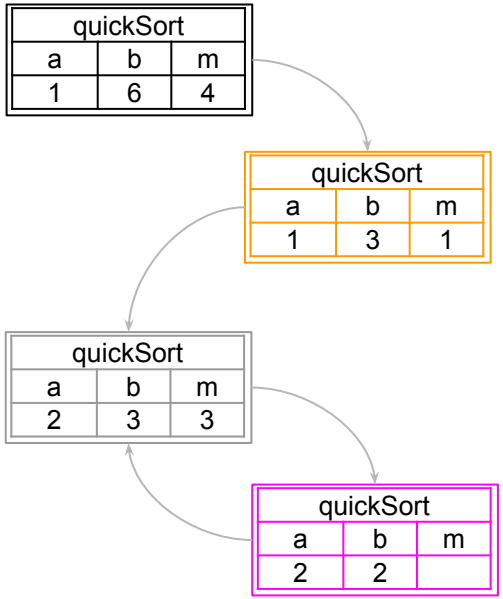
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

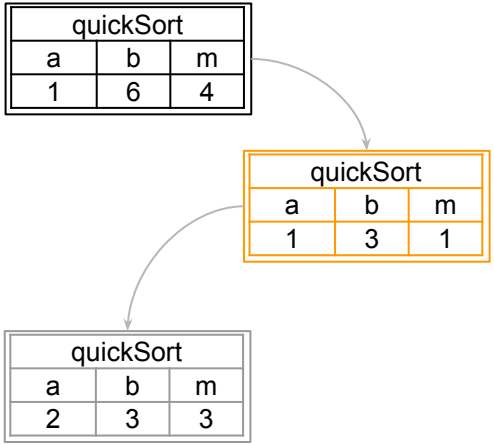
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
    retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

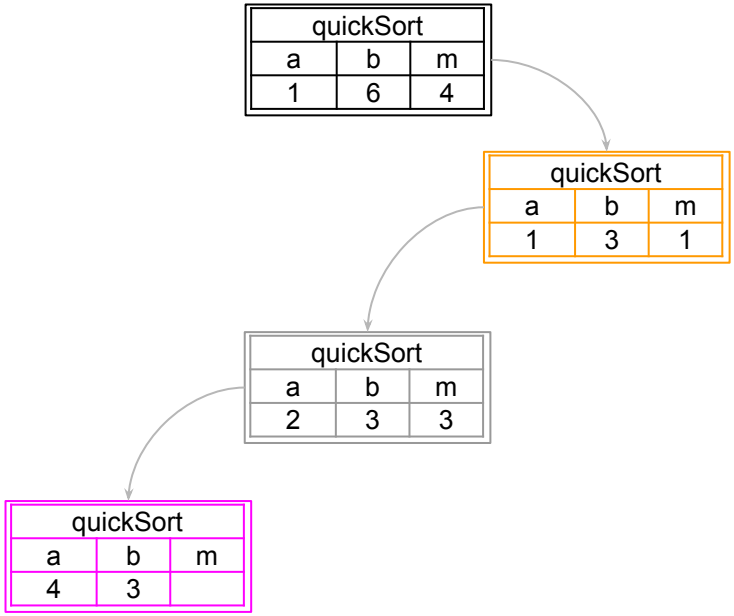
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

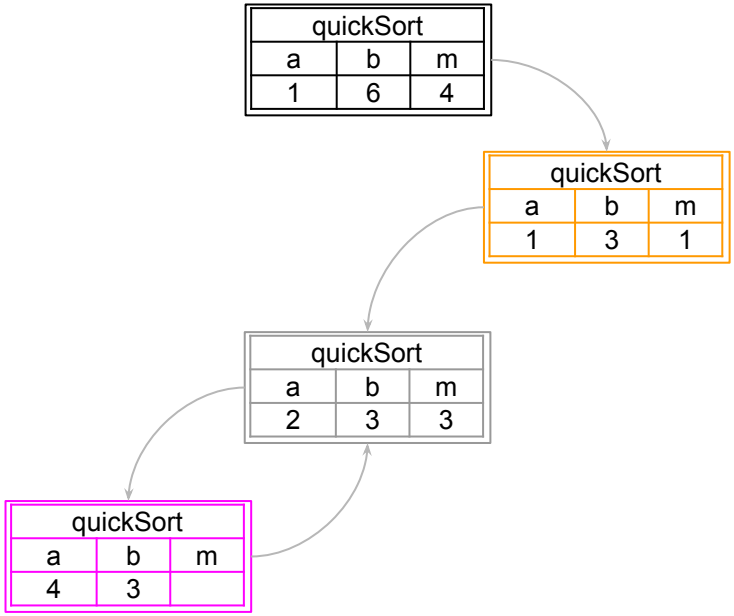
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

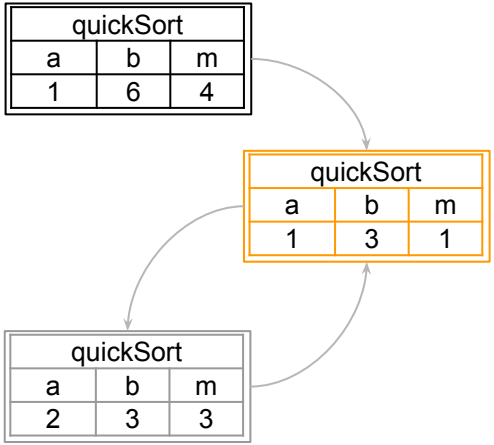
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
    retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

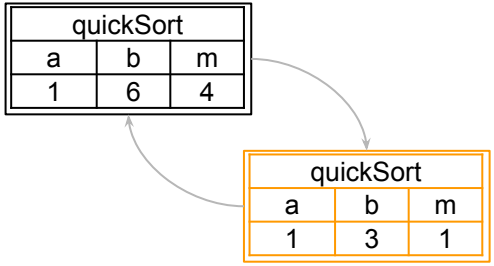
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

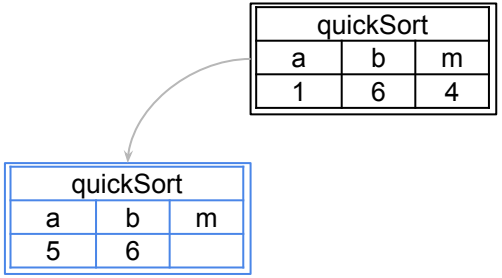
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```

quickSort		
a	b	m
1	6	4

i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
      retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

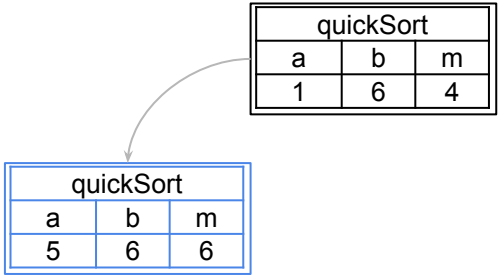
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

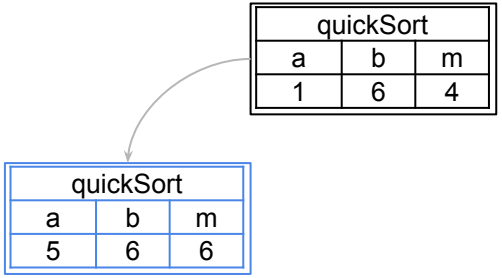
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

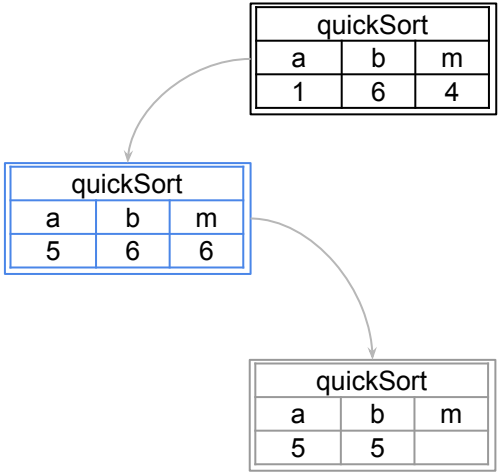
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

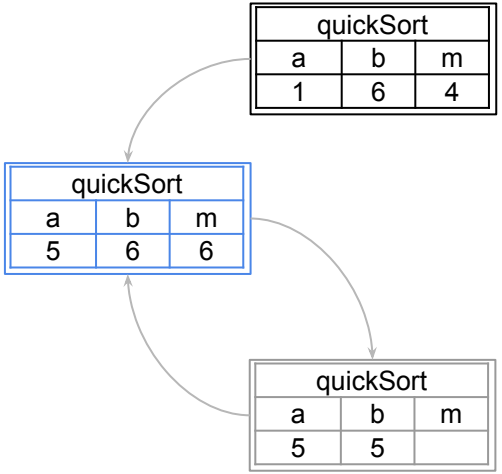
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

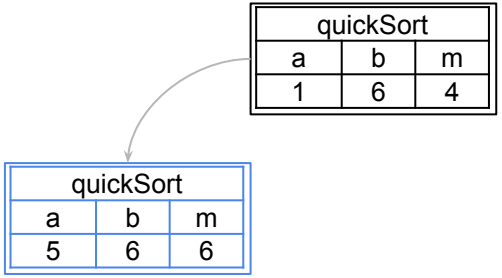
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
    retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

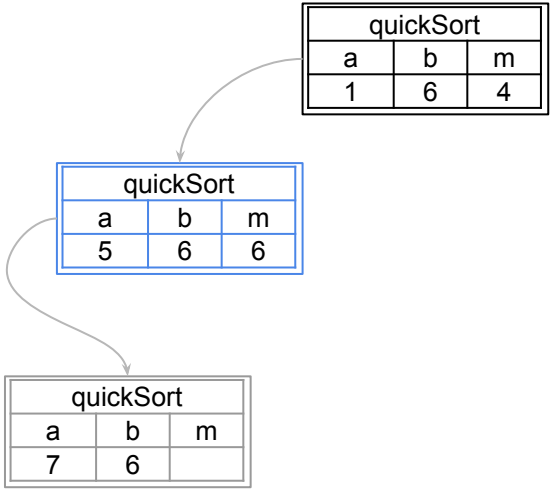
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

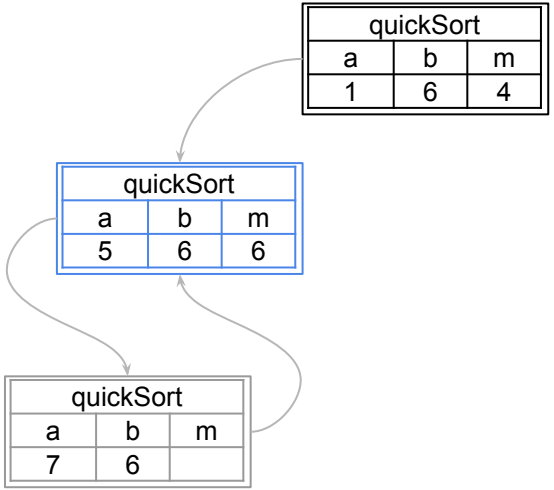
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

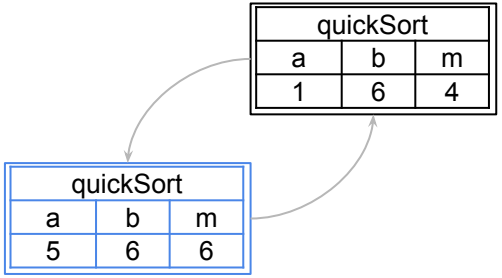
```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
    retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
      se v[i] ≤ x
            trocar(v,m,i)
            m ← m + 1
trocar(v,m,b)
retorne m
```



i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```

quickSort		
a	b	m
1	6	4

i	1	2	3	4	5	6
v[i]	4	8	15	16	23	42

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```

Análise

- Temos melhor e pior caso?

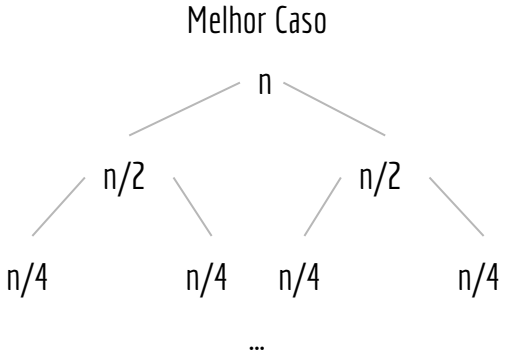
```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

Análise

- Temos melhor e pior caso?
- Tudo depende de *m*.
  - *m* pode dividir o vetor em aproximadamente 2, e cada chamada recursiva vai lidar com *n/2* elementos
  - *m* pode dividir de maneira desbalanceada
    - Uma chamada lida com *n-1* elementos e outra lida com 0 elementos

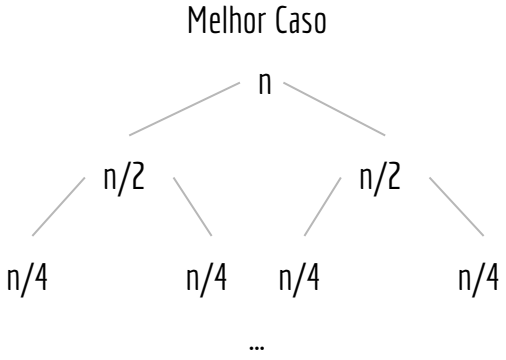
```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

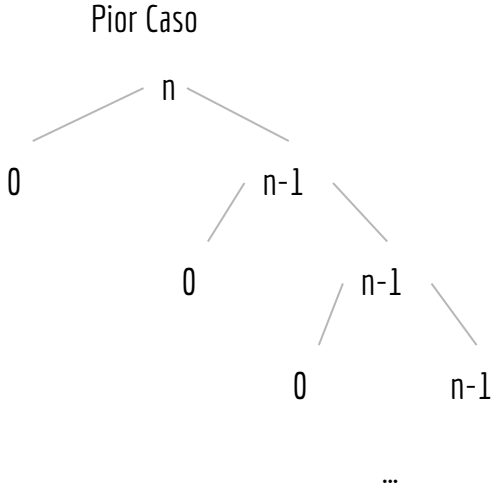
Análise




```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

Análise






```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

Análise

- Considerando o número de comparações

$$C^+(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^+(n-1) + C^+(0) + C_p(n), & \text{se } n > 1 \end{cases}$$

```
função quickSort(v,a,b)
se a ≥ b
     retorne
m ← particionar(v,a,b)
quickSort(v,a,m-1)
quickSort(v,m+1,b)
retorne
```

```
função particionar(v,a,b)
x ← v[b] //pivô
m ← a
para i ← a até b-1
     se v[i] ≤ x
          trocar(v,m,i)
          m ← m + 1
trocar(v,m,b)
retorne m
```

Análise

- Considerando o número de comparações

$$C^+(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^+(n-1) + C^+(0) + C_p(n), & \text{se } n > 1 \end{cases}$$

Custo para uma entrada de tamanho 0 é zero.

Custo de particionar. Para **simplificar**, vamos considerar *n*.

Análise

$$C^+(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^+(n-1) + n, & \text{se } n > 1 \end{cases}$$

Análise

$$C^+(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^+(n-1) + n, & \text{se } n > 1 \end{cases}$$

$$C^+(n) = \frac{n^2 + n - 2}{2} \approx \frac{n^2}{2}$$

Análise

$$C^-(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^-(\lfloor \frac{n}{2} \rfloor) + C^-(\lfloor \frac{n}{2} \rfloor) + C_p(n), & \text{se } n > 1 \text{ e } n \text{ impar} \\ C^-(\frac{n}{2}) + C^-(\frac{n}{2} - 1) + C_p(n), & \text{se } n > 1 \text{ e } n \text{ par} \end{cases}$$

Mais simplificações

$$C^-(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ C^-(\lfloor \frac{n}{2} \rfloor) + C^-(\lfloor \frac{n}{2} \rfloor) + C_p(n), & \text{se } n > 1 \text{ e } n \text{ impar} \\ C^-(\frac{n}{2}) + C^-(\frac{n}{2} - 1) + C_p(n), & \text{se } n > 1 \text{ e } n \text{ par} \end{cases}$$

De maneira semelhante ao Mergesort, para simplificar a análise, vamos considerar $\lfloor \frac{n}{2} \rfloor \approx \frac{n}{2} - 1 \approx \frac{n}{2}$

Então

$$C^{-}(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ 2C^{-}(\frac{n}{2}) + n, & \text{se } n > 1 \end{cases}$$

Então

$$C^-(n) = \begin{cases} 0, & \text{se } n \leq 1, \\ 2C^-(\frac{n}{2}) + n, & \text{se } n > 1 \end{cases}$$

$$C^-(n) = n \log_2 n$$

Teorema

Para uma entrada de tamanho *n*, o custo *C(n)* de comparações para o algoritmo Quicksort é

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Pense nisso

Parece que não ganhamos nada com o Quicksort, já que a análise de custo de comparações mostra que ele não é melhor que algoritmos básicos

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Mas o Quicksort é um dos algoritmos mais usados. Motivos?

Pense nisso

Parece que não ganhamos nada com o Quicksort, já que a análise de custo de comparações mostra que ele não é melhor que algoritmos básicos

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Mas o Quicksort é um dos algoritmos mais usados. Motivos:

- O caso médio (tópico para análise de algoritmos) é igual ao melhor caso

Pense nisso

Parece que não ganhamos nada com o Quicksort, já que a análise de custo de comparações mostra que ele não é melhor que algoritmos básicos

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Mas o Quicksort é um dos algoritmos mais usados. Motivos:

- O caso médio (tópico para análise de algoritmos) é igual ao melhor caso
- Você precisa “dar muito azar” para cair no pior caso

Pense nisso

Parece que não ganhamos nada com o Quicksort, já que a análise de custo de comparações mostra que ele não é melhor que algoritmos básicos

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Mas o Quicksort é um dos algoritmos mais usados. Motivos:

- O caso médio (tópico para análise de algoritmos) é igual ao melhor caso
- Você precisa “dar muito azar” para cair no pior caso
- O algoritmo é *in-place* (não precisa de vetores auxiliares) - Diferente do Mergesort

Pense nisso

Parece que não ganhamos nada com o Quicksort, já que a análise de custo de comparações mostra que ele não é melhor que algoritmos básicos

$$n \log_2 n \leq C(n) \leq \frac{n^2 + n - 2}{2}$$

Mas o Quicksort é um dos algoritmos mais usados. Motivos:

- O caso médio (tópico para análise de algoritmos) é igual ao melhor caso
- Você precisa "dar muito azar" para cair no pior caso
- O algoritmo é *in-place* (não precisa de vetores auxiliares) - Diferente do Mergesort
- O algoritmo se mantém concentrado em regiões pequenas do vetor (evita ficar percorrendo tudo o tempo todo)
    - Melhor localidade da memória na maioria dos computadores "convencionais" - Tópico de Arquitetura

Variantes

Tudo depende da escolha do pivô

Existem variantes do Quicksort que tentam escolher o pivô de forma a evitar o desbalanceamento da árvore

- Exemplos:
  - Cálculo da mediana
  - Estimativa da mediana (amostra de 3 ou 5 elementos)
  - Sorteio do pivô

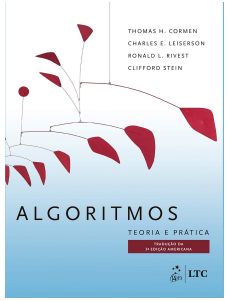
Exercícios

1. Crie um vetor de pelo menos 6 posições que force o pior caso do Quicksort.
2. Implemente o Quicksort em C.
3. Demonstre (passo a passo) o pior e o melhor casos para número de comparações do algoritmo Quicksort.
4. Demonstre o pior e o melhor casos para o número de trocas para o algoritmo Quicksort.

Referências

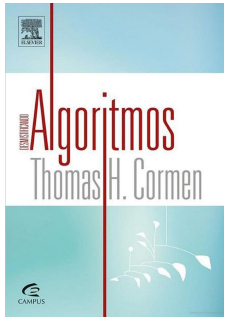
Renato Carmo. Algoritmos e Estruturas de Dados. www.inf.ufpr.br/renato

T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e Prática. 3a ed. 2012

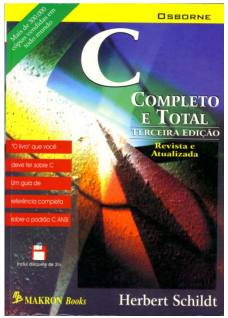


R. Sedgewick, K. Wayne. Algorithms Part I. 4a ed. 2014

T. Cormen. Desmistificando algoritmos. 2017.



H. Schildt. C completo e total. 1996



Licença

Este obra está licenciado com uma Licença Creative Commons Atribuição 4.0 Internacional.