Quicksort

Letícia Rodrigues Bueno

UFABC

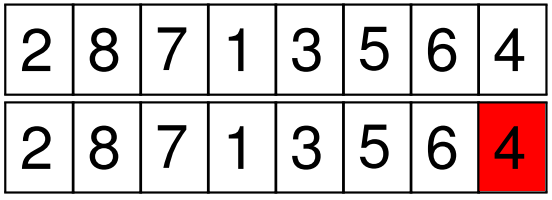
Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

2	8	7	1	3	5	6	4

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:



Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

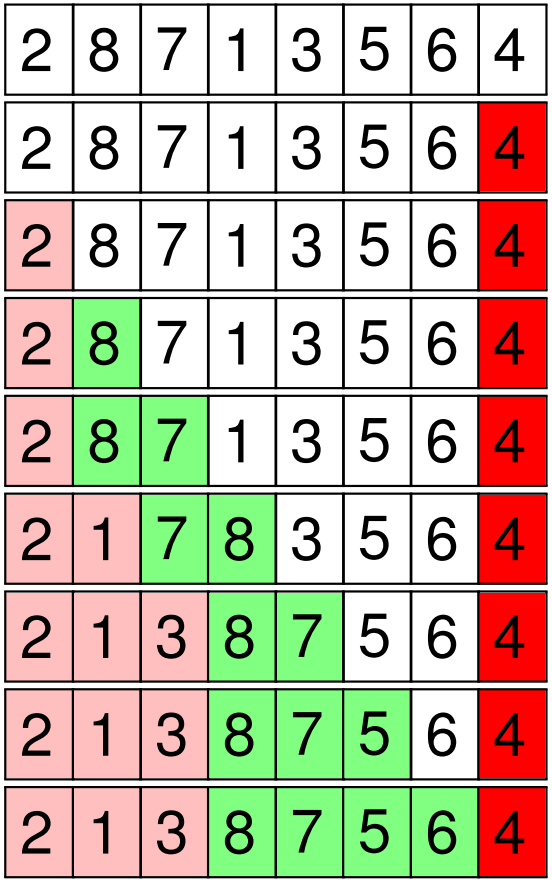
Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:

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

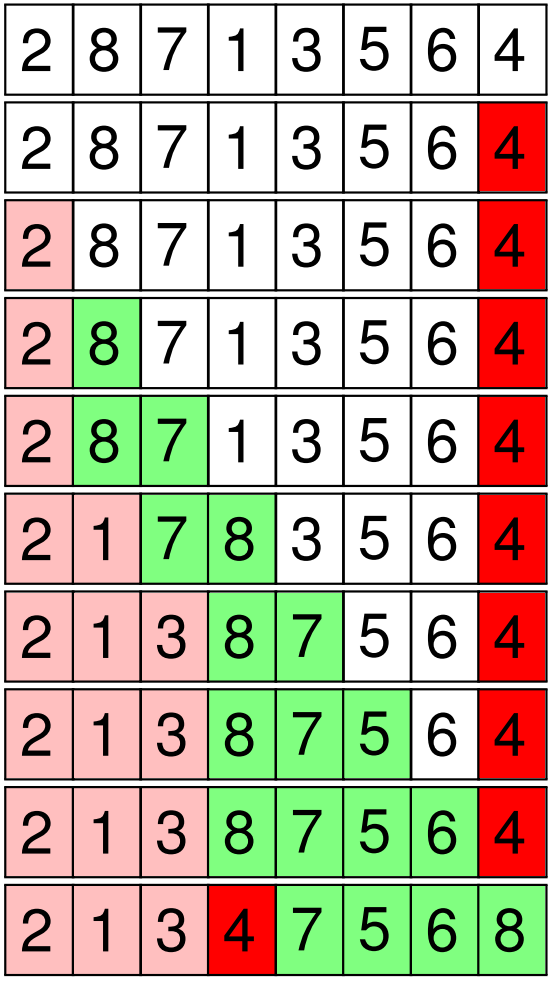
Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:



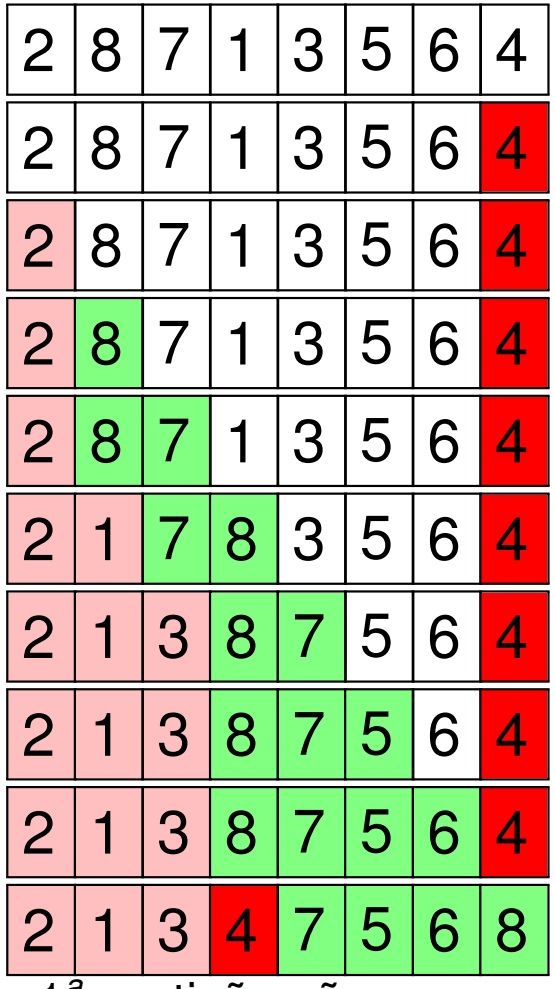
Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:



Quicksort

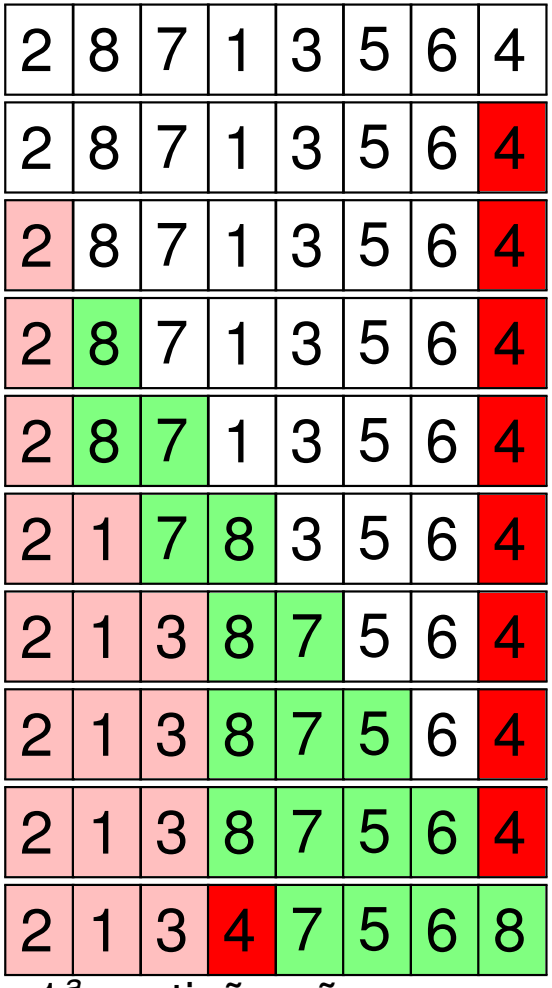
Legenda: pivô; 1ª partição: ; 2ª partição:



todos elementos da 1ª partição são menores ou iguais ao pivô

Quicksort

Legenda: pivô; 1ª partição: ; 2ª partição:



todos elementos da 1ª partição são menores ou iguais ao pivô

todos elementos da 2ª partição são maiores que o pivô

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

Chamada externa:

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

Chamada externa:
quicksort(A,1,length(A))

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

Quicksort - Complexidade no Pior Caso

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;
- **Complexidade no pior caso:**

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;
- **Complexidade no pior caso:**

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ \underbrace{T(n-1) + T(0)}_{\textit{dividir}} + \underbrace{\Theta(n)}_{\textit{conquistar}} = T(n-1) + \Theta(n), \text{se } n > 1. \end{cases}$$

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;
- **Complexidade no pior caso:**

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ \underbrace{T(n-1) + T(0)}_{\textit{dividir}} + \underbrace{\Theta(n)}_{\textit{conquistar}} = T(n-1) + \Theta(n), \text{se } n > 1. \end{cases}$$

Pelo Método de Substituição:

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;
- **Complexidade no pior caso:**

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ \underbrace{T(n-1) + T(0)}_{\textit{dividir}} + \underbrace{\Theta(n)}_{\textit{conquistar}} = T(n-1) + \Theta(n), \text{se } n > 1. \end{cases}$$

Pelo Método de Substituição:

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

Quicksort - Complexidade no Pior Caso

- **Dividir:** problema é particionado em dois subproblemas;
- **Conquistar:** subproblemas são ordenados;
- **Combinar:** subproblemas são ordenados localmente, então não é necessário combinar;
- **Complexidade no pior caso:**

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ \underbrace{T(n-1) + T(0)}_{\textit{dividir}} + \underbrace{\Theta(n)}_{\textit{conquistar}} = T(n-1) + \Theta(n), \text{se } n > 1. \end{cases}$$

Pelo Método de Substituição:

$$T(n) = T(n-1) + \Theta(n) = \Theta(\mathbf{n^2})$$

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

- Nesta versão do Quicksort, dependendo da entrada, a pilha de execução pode crescer muito.

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

- Nesta versão do Quicksort, dependendo da entrada, a pilha de execução pode crescer muito.
- Isso não afeta o desempenho do algoritmo, mas pode esgotar o espaço de memória disponível.

Quicksort

```
1  quicksort(A, p, r):
2      se (p < r) então
3          q = partição(A,p,r)
4          quicksort(A,p,q − 1)
5          quicksort(A,q + 1,r)
```

```
1  partição(A, p, r):
2      pivo = A[r]
3      i = p − 1
4      para (j = p; j ≤ r − 1; j + +) faça
5          se (A[j] ≤ pivo) então
6              i = i + 1
7              A[i] ⇔ A[j])
8      A[i + 1] ⇔ A[r])
9      retorne i + 1
```

- Nesta versão do Quicksort, dependendo da entrada, a pilha de execução pode crescer muito.
- Isso não afeta o desempenho do algoritmo, mas pode esgotar o espaço de memória disponível.
- Como resolver?

Quicksort Randomizado

```
1  quicksortAleat(A, p, r):
2      se (p < r) então
3          q = partiçãoAleat(A,p,r)
4          quicksortAleat(A,p,q − 1)
5          quicksortAleat(A,q + 1,r)
```

Quicksort Randomizado

```
1  quicksortAleat(A, p, r):
2      se (p < r) então
3          q = partiçãoAleat(A,p,r)
4          quicksortAleat(A,p,q − 1)
5          quicksortAleat(A,q + 1,r)
```

```
1  partiçãoAleat(A, p, r):
2      i = RANDOM(p, r)
3      A[r] ⇔ A[i])
4      retorne partição(A, p, r)
```

Quicksort Randomizado

```
1  quicksortAleat(A, p, r):
2      se (p < r) então
3          q = partiçãoAleat(A,p,r)
4          quicksortAleat(A,p,q − 1)
5          quicksortAleat(A,q + 1,r)
```

Chamada externa:

```
1  partiçãoAleat(A, p, r):
2      i = RANDOM(p, r)
3      A[r] ⇔ A[i])
4      retorne partição(A, p, r)
```

Quicksort Randomizado

```
1  quicksortAleat(A, p, r):
2      se (p < r) então
3          q = partiçãoAleat(A,p,r)
4          quicksortAleat(A,p,q − 1)
5          quicksortAleat(A,q + 1,r)
```

Chamada externa:
quicksortAleat(A,1,length(A))

```
1  partiçãoAleat(A, p, r):
2      i = RANDOM(p, r)
3      A[r] ⇔ A[i])
4      retorne partição(A, p, r)
```

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *quicksort* é estável? Exemplifique.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *quicksort* é estável? Exemplifique.
2. Modifique o *quicksort* para fazer ordenação decrescente.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *quicksort* é estável? Exemplifique.
2. Modifique o *quicksort* para fazer ordenação decrescente.
3. Forneça um breve argumento mostrando que o tempo de execução de partição() sobre um subarranjo de tamanho n é $\Theta(n)$.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *quicksort* é estável? Exemplifique.
2. Modifique o *quicksort* para fazer ordenação decrescente.
3. Forneça um breve argumento mostrando que o tempo de execução de partição() sobre um subarranjo de tamanho n é $\Theta(n)$.
4. Que valor de q partição() retorna quando todos os elementos no arranjo $A[p..r]$ têm o mesmo valor? Modifique partição() de forma que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p..r]$ têm o mesmo valor.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *quicksort* é estável? Exemplifique.
2. Modifique o *quicksort* para fazer ordenação decrescente.
3. Forneça um breve argumento mostrando que o tempo de execução de partição() sobre um subarranjo de tamanho n é $\Theta(n)$.
4. Que valor de q partição() retorna quando todos os elementos no arranjo $A[p..r]$ têm o mesmo valor? Modifique partição() de forma que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p..r]$ têm o mesmo valor.
5. Qual o melhor caso do *quicksort*? Qual seu custo neste caso?

Bibliografia Utilizada

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

SZWARCFITER, J. L. e MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994.

ZIVIANI, N. Projeto de Algoritmos: com implementações em Pascal e C, 2ª edição, Cengage Learning, 2009.