

Ordenação por troca

Programação de computadores II

Prof. Renan Augusto Starke

Instituto Federal de Santa Catarina – IFSC
Campus Florianópolis
`renan.starke@ifsc.edu.br`

2 de junho de 2017



INSTITUTO FEDERAL
SANTA CATARINA

Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
INSTITUTO FEDERAL DE SANTA CATARINA

Tópicos da aula

- 1 Introdução
- 2 Ordenação por troca
- 3 Exercícios

1 Introdução

2 Ordenação por troca

3 Exercícios

- ▶ Entender alguns fundamentos matemáticos relacionados com algoritmos de ordenação
- ▶ Aprender as ordenações por troca
- ▶ Conhecer o *Quicksort*
- ▶ Aplicar ordenação nas estruturas de dados conhecidas

1 Introdução

2 Ordenação por troca

3 Exercícios

Ordenação por troca

Compreende em algoritmos onde a ordenação é realizada por *trocas entre pares* de elementos.

- ▶ Troca entre elementos adjacentes
- ▶ Troca entre elementos mais distantes

Algoritmos mais conhecidos:

- ▶ *Bubble Sort*
- ▶ *Quick Sort*

Quicksort

O *Quicksort* ou ordenação rápida é um algoritmo no estilo “Dividir e conquistar”. Este estilo de algoritmo resolve um problema dividindo-o em dois ou mais subproblemas, resolvendo-os cada um destes e combinando a solução no final.

Para ordenar uma sequência $S = \{s_1, s_2, \dots, s_n\}$, o Quicksort realiza os seguintes passos:

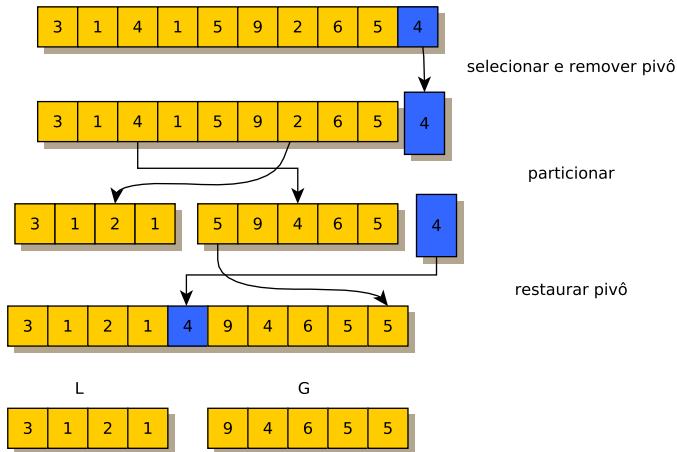
- 1 Seleciona-se um dos elementos de S . Este elemento é chamado de pivô p .
- 2 Remove-se p de S e particiona-se o elementos restantes de S in duas sequencias distintas:
 - L : todos os elementos de L são iguais ou menores do que p
 - G : todos os elementos de G são iguais ou maiores do que p
 - Em geral, L e G não estão ordenados.

- 1 Rearranja-se os elementos da sequência da seguinte forma:

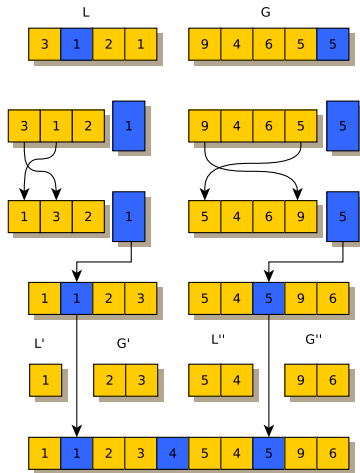
$$S = \{\underbrace{l_1, l_2, \dots, l_{|L|}}_L, \underbrace{p}_{\text{pivô}}, \underbrace{g_1, g_2, \dots, g_{|G|}}_G\} \quad (1)$$

- 2 Note que p já encontra-se no lugar adequado no vetor ordenado.
- 3 Continua-se chamando Quicksort recursivamente nas sequências não ordenadas G e L .

Quicksort



Quicksort

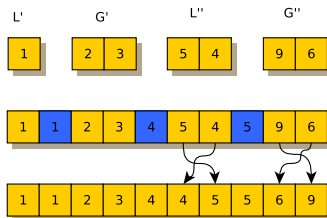


selecionar e remover pivô

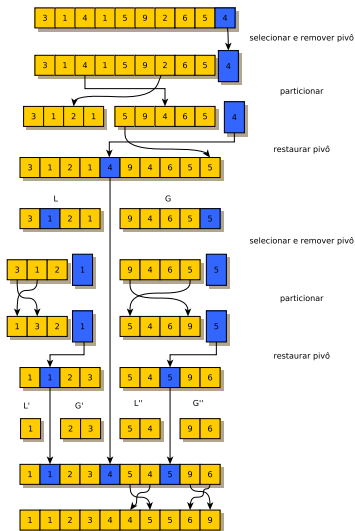
particionar

restaurar pivô

Quicksort



Quicksort



- ▶ A seleção do pivô impacta somente no tempo de execução
- ▶ Se houver uma seleção pobre, o tempo de execução será pobre
- ▶ O número de chamadas recursivas deve ser limitado:
 - Para-se de dividir quando $|L| = 2$ ou $|G| = 2$
 - Nesta situação, ordena-se manualmente os dois últimos elementos

- ▶ Pior caso: $O(n^2)$
- ▶ Caso médio: $O(n \log n)$
- ▶ Melhor médio: $O(n \log n)$

Lembrem-se, o *Bubble Sort* é sempre $O(n^2)$

- ▶ As análises do tempo de execução demonstram a importância da seleção do pivô
- ▶ Seleções pobres fazem com que se tenha $O(n^2)$
- ▶ Porém, a análise de caso médio demonstra que se qualquer elemento de uma sequência pode ser selecionado como pivô, o tempo de execução é $O(n \log n)$
- ▶ Logo, simplesmente podemos selecionar um pivô aleatoriamente

- ▶ Se esperamos que a sequência esteja sempre desordenada, pode-se selecionar qualquer elemento como pivô, pois garante-se a aleatoriedade e, conseqüentemente, $O(n \log n)$.
- ▶ Na prática, as sequências geralmente estão “quase” ordenadas
- ▶ Caso obtivermos sequências já ordenadas crescentemente, selecionando o primeiro elemento garantiremos $O(n^2)$
- ▶ Caso obtivermos sequências já ordenadas decrescentemente, selecionando o último elemento garantiremos $O(n^2)$
- ▶ Baseando-se nisso, é muito interessante utilizarmos a *mediana* como pivô

Seleção do Pivô

- ▶ Para esperarmos $O(n \log n)$, devemos selecionar a mediana em, no mínimo em $O(n)$
- ▶ Como encontrar a mediana? Não podemos ordenar, obviamente.
- ▶ Uma maneira eficiente é utilizar a *mediana de três* como pivô:

```
1  mediana-de-tres(array, esq, direi):
2
3      med = (esq + direi)/2
4
5      if array[direi] < array[esq]:
6          swap(array, esq, direi)
7
8      if array[med] < array[esq]:
9          swap(array, med, esq)
10
11     if array[direi] < array[med]:
12         swap(array, direi, med)
13
14     return med
```

```
1  quicksort(A, esquerda, direita){
2      if esquerda < direita then
3          p = particionar(A, esquerda, direita)
4          quicksort(A, esquerda, p)
5          quicksort(A, p + 1, direita)
6      }

1  partition(A, esquerda, direita) {
2
3      med = mediana-de-tres(array, esquerda, direita):
4
5      pivot = A[med]
6      i = esquerda - 1
7      j = direita + 1
8
9      for(;;) {
10         do
11             i = i + 1
12             while A[i] < pivot
13
14         do
15             j = j - 1
16             while A[j] > pivot
17
18         if i >= j
19             return j
20
21         swap(A[i],A[j])
22     }
23 }
```

1 Introdução

2 Ordenação por troca

3 Exercícios

- ▶ Implemente o *Quicksort* para um vetor de inteiros.
 - Teste seu algoritmo para um vetor de 100.000 de elementos alocados dinamicamente.
 - Inicialize-o com números aleatórios.
 - Meça o tempo de execução para 20 execuções do *Quicksort*.
 - Calcule o tempo de execução médio.
 - **OBS: números aleatórios gerados por:** `srand()` alimentando a semente com `srand (getpid() ^time(NULL))`;
- ▶ Estenda a implementação da lista duplamente encadeada com uma função de ordenação por *Quicksort*.
- ▶ Compare com o tempo de execução do *BubbleSort*