

## SME0827 - Estruturas de Dados

### Ordenação Aula 07

Professor: André C. P. L. F. de Carvalho, ICMC-USP  
PAE: Moisés Rocha dos Santos  
Monitor:

© André de Carvalho - ICMC/USP

1

1

## Ordenação por *Quicksort*

- A maioria dos algoritmos de ordenação é baseada no *Quicksort*
- Como a ordenação por combinação, utiliza a estratégia de divisão e conquista
  - Começa percorrendo a lista, quando transfere
    - Itens pequenos para o início da lista
    - itens grandes para o final
    - O que significa pequeno e grande?

© André de Carvalho - ICMC/USP

4

4

## Aula de hoje

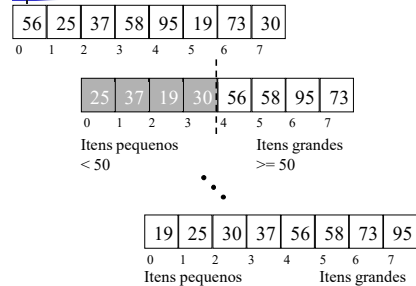
- Algoritmos de ordenação
- Quicksort
- Bubblesort
- Inserção

© André de Carvalho - ICMC/USP

2

2

## Ordenação por *Quicksort*



5

5

## Introdução

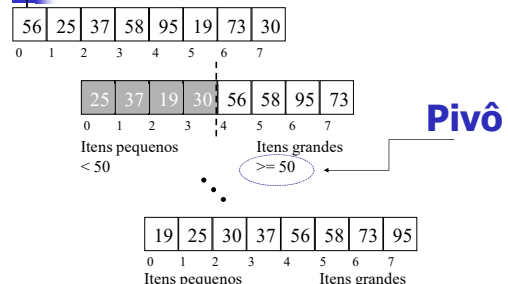
- Em muitas tarefas práticas, precisamos ordenar dados
  - Divulgar a lista de candidatos do ENEM para um dado curso por ordem da nota
- Já vimos dois algoritmos de ordenação
  - Ordenação por seleção (selection sort)
    - $O(n^2)$
  - Ordenação por combinação (merge sort)
    - $O(n \log n)$

© André de Carvalho - ICMC/USP

3

3

## Ordenação por *Quicksort*



6

6

## Como escolher o pivô

- Varias opções:
  - Item escolhido aleatoriamente
    - Aumenta risco de uma das sublistas ter 1 item
  - Primeiro (último) item da lista
    - Geralmente suficiente
    - Ruim se a lista estiver parcialmente ordenada
  - Item do meio da lista
    - Melhor se a lista estiver parcialmente ordenada
  - Escolher item da mediana (primeiro, meio e último)
    - Melhor alternativa

© André de Carvalho - ICMC/USP 7

7

## Ordenação por Quicksort

- Quando os itens coincidem (início = fim)
  - Se posição deles contém um valor grande
    - Pivô deve ser o menor item da lista (fronteira está na posição 0)
    - Põe na lista ordenada
  - Se a posição contém um valor pequeno
    - Trocar o valor da posição com o do item pivô, no início da lista
    - Geralmente, o item onde *início* e *fim* coincidem é aquele que tem o valor pequeno mais a direita da sublista1
      - Sublista1: itens menores que o pivô

10

10

## Ordenação por Quicksort

- Caso simples
  - Lista com 0 ou 1 items
- Parte recursiva
  - Escolhe um item pivô para dividir uma lista em duas sublistas
  - Rearranja os itens da lista em:
    - Sublista 1: itens menores que o pivô
    - Sublista 2: itens maiores ou iguais que o pivô
  - Ordena as sublistas

© André de Carvalho - ICMC/USP 8

8

## Ordenação por Quicksort

11

11

## Ordenação por Quicksort

- Partição da lista
  - Ignorando o pivô, faz um índice *início* (*fim*) apontar para o primeiro (último) item da lista
  - Repetir até índices *início* e *fim* coincidirem
    - Mover *início* para a direita
      - Até coincidir com *fim* ou apontar para um item grande
    - Mover *fim* para a esquerda
      - Até coincidir com *início* ou apontar para um item pequeno
    - Se *início* ≠ *fim*, trocar os valores apontados por eles

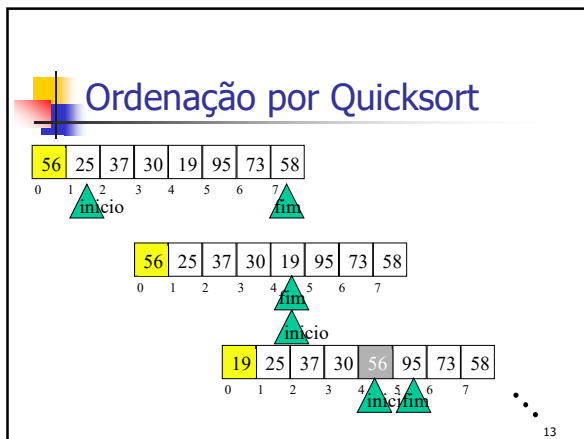
© André de Carvalho - ICMC/USP 9

9

## Ordenação por Quicksort

12

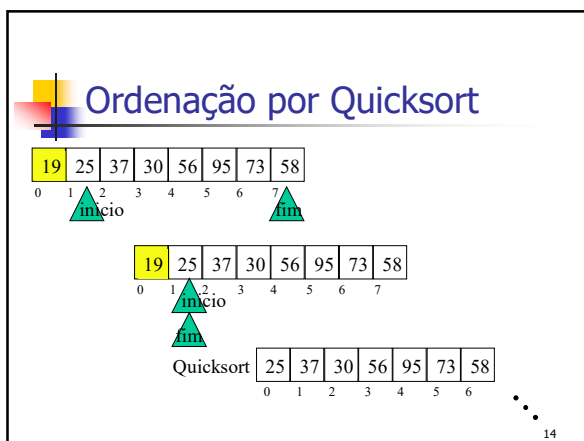
12



## Ordenação por Quicksort

```
def particao (lista1, inicio, fim):
    centro = (inicio + fim)/2
    if lista1[inicio] > lista1[centro]:
        lista1[inicio], lista1[centro] = lista1[centro],
    lista1[inicio]
    if lista1[inicio] > lista1[fim]:
        lista1[inicio], lista1[fim] = lista1[fim],
    lista1[inicio]
    if lista1[centro] > lista1[fim]:
        lista1[centro], lista1[fim] = lista1[fim],
    lista1[centro]
    lista1[centro], lista1[inicio] = lista1[inicio],
    lista1[centro]
    centro = inicio
    i = inicio + 1
    j = fim
    while True:
        while i <= fim and lista1[i] <=
            lista1[centro]:
            i += 1
        while j >= inicio and lista1[j] >
            lista1[centro]:
            j -= 1
        if i > j:
            break
        else:
            lista1[i], lista1[j] = lista1[j],
            lista1[i]
        lista1[j], lista1[centro] = lista1[centro],
        lista1[j]
    return j
```

16



## Ordenação por Quicksort

N	Ord. Quicksort	Ord. Combinação
10	0.10 mseg.	0.54 mseg.
20	0.26 mseg.	1.17 mseg.
40	0.52 mseg.	2.54 mseg.
100	1.76 mseg.	6.90 mseg.
200	4.04 mseg.	14.84 mseg.
400	8.85 mseg.	31.25 mseg.
1000	26.04 mseg.	84.38 mseg.
2000	56.25 mseg.	179.17 mseg.
4000	129.17 mseg.	383.33 mseg.
10000	341.67 mseg.	997.67 mseg.

17

## Ordenação por Quicksort

```
def quicksort(lista1):
    quicksort_rec (lista1, 0, len(lista1)-1)
    return (lista1)

def quicksort_rec (lista1, inicio, fim):
    if (fim > inicio):
        front = particao (lista1, inicio, fim)
        quicksort_rec (lista1, inicio, front-1)
        quicksort_rec (lista1, front+1, fim)
```

15

- ## Ordenação por Quicksort
- Desempenho do *Quicksort*
    - Em um vetor definido pela mediana *Quicksort* tende a ter complexidade  $O(N \log n)$
    - Pior caso ocorre quando o vetor já está ordenado, quando a complexidade é de  $O(n^2)$ 
      - Apesar disto, na prática, é muito mais rápido que a maioria dos algoritmos
      - Escolha padrão para os procedimentos de ordenação
      - Existem várias estratégias para melhorar desempenho
- 18

## Pausa



© André de Carvalho - ICMC/USP

19

## Ordenação por Bubblesort

```
def bubblesort(lista1):
    for j in range(len(lista1)-1):
        for k in range(len(lista1)-1-j):
            if lista1[k] > lista1[k+1]:
                aux = lista1[k]
                lista1[k] = lista1[k+1]
                lista1[k+1] = aux
    return lista1
```

*Listaz = [3, 5, 1, 4, 5]*  
 print (bubblesort(Listaz))  
 [1, 3, 4, 5, 5]

22

## Ordenação por Bubblesort

- Algoritmo de ordenação mais simples e, provavelmente, menos eficiente
- Passa sequencialmente pela lista várias vezes
  - Em cada passada compara um item com seu sucessor
    - Trocas suas posições se não estiverem na ordem correta
    - Iteração  $i$  coloca o item  $n-i$  na posição correta

20

## Ordenação por Bubblesort

```
def bubblesort(lista1):
    for j in range(len(lista1)-1):
        for k in range(len(lista1)-1-j):
            if lista1[k] > lista1[k+1]:
                lista1[k], lista1[k+1] = lista1[k+1], lista1[k]
    return lista1
```

*Listaz = [3, 5, 1, 4, 5]*  
 print (bubblesort(Listaz))  
 [1, 3, 4, 5, 5]

23

## Ordenação por Bubblesort

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

25	56	37	58	95	19	73	30
0	1	2	3	4	5	6	7

25	37	56	58	95	19	73	30
0	1	2	3	4	5	6	7

25	37	56	58	19	30	73	95
0	1	2	3	4	5	6	7

21

## Ordenação por Bubblesort

- Observações
  - Complexidade:  $O(n^2)$
- Vantagens
  - Requer pouca memória adicional
    - Apenas uma variável para guardar o valor da variável *aux* (ou nem ela)
  - Para arquivo ordenado, complexidade  $O(n)$

24

## Ordenação por inserção

- Nome original: *insertion sort*
- Como algoritmo funciona:
  - Assim como ordenação por seleção e *bubblesort*, visita um a um os itens da lista
  - Procura garantir que o item atual está na ordem certa em relação aos outros já visitados
  - Inserir itens em uma lista já ordenada

© André de Carvalho - ICMC/USP 25

25

## Ordenação por inserção

- Observações
  - Complexidade:  $O(n^2)$ 
    - Provar
- Vantagens
  - Requer pouca memória adicional
    - Apenas uma variável para guardar o valor da variável *aux* (ou nem ela)
  - Geralmente melhor que o *bubblesort*

28

28

## Ordenação por inserção

na ordem

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

na ordem

25	56	37	58	95	19	73	30
0	1	2	3	4	5	6	7

na ordem

25	37	56	58	95	19	73	30
0	1	2	3	4	5	6	7

...

26

26

## Conclusão

- Análise de algoritmos
- Algoritmos de Ordenação
  - Seleção, Combinação, *Quicksort*, Inserção e *Bubblesort*
  - Complexidade

29

29


## Ordenação por inserção

```
def insercao (lista1):
    for i in range (1, len(lista1)):
        aux = lista1[i]
        j = i
        while j > 0 and lista1[j-1] > aux:
            lista1[j] = lista1[j-1]
            j -= 1
        lista1[j] = aux
```

27

27

## Perguntas



30

30