

# Classificação e Pesquisa de Dados

Aulas 06 e 07  
Classificação de dados por Troca : QuickSort

UFRGS

INF01124

1

## Quicksort Método da Partição e Troca

- ◆ Comparado com os demais métodos, apresenta, em média, o menor tempo de classificação
- ◆ Baseia-se numa técnica fundamental para solução de problemas conhecida como Dividir para Conquistar
- ◆ O problema de ordenar um vetor de  $n$  elementos é subdividido em dois problemas menores e independentes, cujos resultados podem ser facilmente combinados
- ◆ O processo de subdivisão/cominação é aplicado recursivamente

2

## Quicksort Princípio de Classificação

- ◆ Inicialmente, o vetor de chaves  $C$  é particionado em três segmentos  $S_1$ ,  $S_2$  e  $S_3$
- ◆  $S_2$  conterá apenas uma chave denominada particionadora ou pivô
- ◆  $S_1$  conterá todas as chaves cujos valores são menores ou iguais ao pivô. Esse segmento está posicionado à esquerda de  $S_2$
- ◆  $S_3$  conterá todas as chaves cujos valores são maiores do que o pivô. Esse segmento está posicionado à direita de  $S_2$

3

## Quicksort

- ◆ Esquema conceitual do particionamento

Vetor Inicial :  $C[1 .. n]$



Vetor Particionado



Onde:  $C[i] \leq C[k]$ , para  $i = 1, \dots, k-1$   
 $C[i] > C[k]$ , para  $i = k+1, \dots, n$

4

## Quicksort Princípio de Classificação (Cont.)

- ◆ O particionamento é reaplicado aos segmentos  $S_1$  e  $S_3$  e a todos os segmentos correspondentes daí resultantes com comprimento  $\geq 1$
- ◆ Quando não restarem segmentos a serem particionados, o vetor estará ordenado
- ◆ Perguntas:
  - Qual é a chave particionadora ideal?
  - Como escolher essa chave ?

5

## Quicksort Escolha do pivô

- ◆ A chave particionadora ideal é aquela que produz segmentos  $S_1$  e  $S_3$  com tamanhos (aproximadamente) iguais: chave de valor mediano
- ◆ A identificação do pivô ideal requer a varredura de todo o vetor (o benefício não justifica o custo)
- ◆ Deseja-se um critério de escolha simples e rápido
- ◆ Sem conhecimento prévio sobre a distribuição de valores das chaves, supõe-se que qualquer uma pode ser o pivô e arbitra-se a primeira chave
- ◆ Caso o vetor já se encontre parcialmente ordenado, pode-se utilizar o elemento médio

6

## Quicksort Particionamento

- ◆ O algoritmo executa o particionamento em  $n$  passos ( $n$  = número de chaves)
- ◆ Nos primeiros  $n-1$  passos:
  - ◆ Chaves menores ou iguais ao pivô são deslocadas para o lado esquerdo do vetor
  - ◆ Chaves maiores que o pivô são deslocadas para o lado direito do vetor
- ◆ No último passo o pivô é inserido em sua posição definitiva

7

## Quicksort Exemplo de Particionamento

**Vetor Original:** [ 9 25 10 18 5 7 15 3 ]  
Particionamento no primeiro nível da recursão: **pivô = 9**

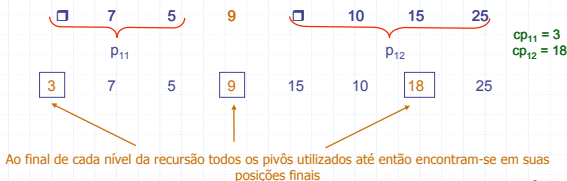
1.		25	10	18	5	7	15	3	esquerda
2.	3	25	10	18	5	7	15	f	direita
3.	3	10	18	5	7	15	25	f	esquerda
4.	3	10	18	5	7	15	25	f	esquerda
5.	3	7	10	18	5	15	25	f	direita
6.	3	7	10	18	5	15	25	f	esquerda
7.	3	7	5	18	10	15	25	f	direita
8.	3	7	5	9	18	10	15	25	8

## Quicksort Exemplo de Particionamento (Cont.)

Resultado do particionamento no primeiro nível da recursão:

3 7 5 9 18 10 15 25

Particionamento no segundo nível da recursão:



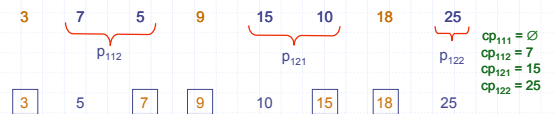
9

## Quicksort Exemplo de Particionamento (Cont.)

Resultado do particionamento no segundo nível da recursão:

3 7 5 9 15 10 18 25

Particionamento no terceiro nível da recursão:



10

## Quicksort Exercício

Suponha que se deseja classificar o seguinte vetor:

O R D E N A

Assuma que a chave particionadora está na posição inicial do vetor e simule as iterações necessárias para a classificação, segundo o algoritmo apresentado.

11

## Procedimento Partição

```

Proc partição (c, i, f, k); /* k = posição ocupada pela chave particionadora */
begin
    i1 ← i; f1 ← f; cp ← c[i1]; esq ← true;
    while i1 < f1 do
        if esq then
            if cp ≥ c[f1] then begin /* esquerda vaga */
                c[i1] ← c[f1]; /* transfere para s1 */
                i1 ← i1 + 1;
                esq ← false;
            end
        else f1 ← f1 - 1;
        if cp < c[i1] then begin /* direita vaga */
            c[f1] ← c[i1]; /* transfere para s3 */
            f1 ← f1 - 1;
            esq ← true;
        end
        else i1 ← i1 + 1;
    k ← i1; c[k] ← cp /* ou k := f1, já que neste ponto i1 = f1 */
end
    
```

12

## Procedimento Quicksort

```

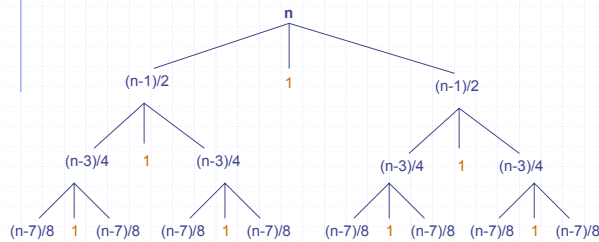
Proc quicksort ( c, i, f )
/* i: posição inicial do segmento; f: posição final do segmento */
begin
  if f > i
    then begin
      partição ( c, i, f, k ); /* particiona */
      quicksort ( c, i, k - 1 ); /* ordena segmento da esquerda */
      quicksort ( c, k + 1, f ); /* ordena segmento da direita */
    end
  end
end

```

13

## Quicksort Análise de Desempenho

◆ **Melhor caso:** subdivisão produz segmentos com mesmo tamanho



14

## Quicksort Análise de Desempenho

◆ **Melhor caso (Cont.)**

Nível recursão	Segmentos	Comparações
0	1	$n - 1$
1	2	$n - 3 = (((n - 1) / 2) - 1) * 2$
2	4	$n - 7 = (((n - 3) / 4) - 1) * 4$
3	8	$n - 15 = (((n - 7) / 8) - 1) * 8$
...	...	...

Total:  $(n - 1) + (n - 3) + (n - 7) + (n - 15) + \dots \lfloor \log_2 n \rfloor$  vezes

$$Total = \sum_{i=1}^{\log_2 n} (n - (2^i - 1)) = n \log_2 n - \sum_{i=1}^{\log_2 n} (2^i - 1) = n \log_2 n + \log_2 n - \sum_{i=1}^{\log_2 n} 2^i$$

15

## Quicksort Análise de Desempenho

◆ **Melhor caso (Cont.)**

$$Total = n \log_2 n + \log_2 n - \sum_{i=1}^{\log_2 n} 2^i$$

◆ Soma dos termos de uma PG  $\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}$

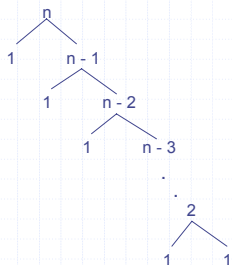
$$\sum_{i=1}^{\log_2 n} 2^i = \frac{2^{\log_2 n + 1} - 1}{2 - 1} - 2^0 = 2(2^{\log_2 n} - 1) = 2(n - 1)$$

$$Total = n \log_2 n + \log_2 n - 2(n - 1) = \Omega(n \log_2 n)$$

16

## Quicksort Análise de Desempenho

◆ **Pior caso:** quando o pivô é a menor (ou a maior) de todas as chaves e esta situação se repete para todos os níveis de subdivisão.



Número de Comparações:

$$T(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 1$$

$$T(n) = (n^2 - n) / 2$$

17

## Quicksort Análise de Desempenho

◆ **Pior caso (Cont.)**

◆ Ocorrerá sempre que o vetor já estiver ordenado e escolhermos a menor (ou maior) chave como particionadora!!!

◆ Para estes casos, o algoritmo por inserção direta apresenta melhor desempenho assintótico

18

## Quicksort na Prática

- ◆ Apesar do seu desempenho no pior caso ser  $n^2$ , *Quicksort* costuma ser, na prática, a melhor escolha:
  - ◆ Na média, sua performance é excelente
  - ◆ O tempo de execução esperado é  $n \log n$ , sendo que a constante associada ao termo mais significativo é bem pequena
  - ◆ Realiza classificação local
  - ◆ Executa eficientemente mesmo em ambientes com memória virtual

19

## Quicksort Análise de Desempenho

### ◆ Tempo de Execução do Caso Médio

- ◆ Muito mais próximo do melhor caso do que do pior caso
- ◆ Por exemplo, suponha que o particionamento em todos os níveis ocorra na proporção 1 para 9 (i.e., não balanceado)

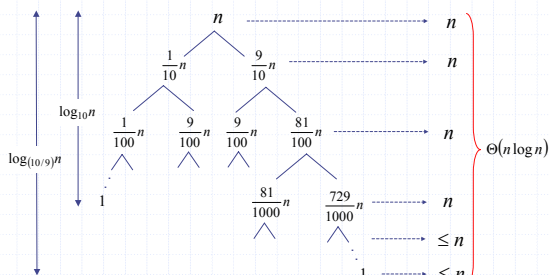
$$T(n) = \underbrace{T(n/10)}_{\text{Tempo para ordenar toda a sequência}} + \underbrace{T(9n/10)}_{\text{Tempo para ordenar } S_3} + \underbrace{n}_{\text{Tempo para particionar}}$$

Função Recorrente: definida em termos da própria função aplicada à entradas menores

20

## Quicksort Análise de Desempenho

### ◆ Particionamento Desbalanceado (1 para 9)



21

## Quicksort Análise de Desempenho

### ◆ Particionamento Desbalanceado (1 para 9)

$$\log_{10} n = \Theta(\log n)$$

- ◆ Note que

$$\begin{aligned} \log_b n &= \frac{\log_a n}{\log_a b} \\ \log_a n &= (\log_a b) \log_b n \\ \log_a n &= k \log_b n \end{aligned}$$

- ◆ Logo

$$\log_a n = \Theta(\log_b n)$$

22

## Quicksort Análise de Desempenho

### ◆ Particionamento Desbalanceado (Cont.)

- ◆ Qualquer subdivisão com proporcionalidade constante produz uma árvore de recursão com profundidade  $\Theta(\log n)$
- ◆ Por exemplo, mesmo com uma subdivisão de 1 para 99, o tempo de execução do *Quicksort* é  $O(n \log n)$  (desde que a seqüência já não se encontre ordenada)
- ◆ No caso médio, pode-se esperar uma mistura de subdivisões boas ( $S_1$  e  $S_3$  não vazios) e más ( $S_1$  ou  $S_3$  vazio). Neste caso,  $T(n) = \Theta(n \log n)$

23

## Quicksort “Randomizado”

- ◆ Antes de iniciar o procedimento de ordenação, permuta alguns elementos randomicamente
  - ◆ Tenta garantir que todas as permutações são igualmente prováveis
  - ◆ A ocorrência do pior caso não é eliminada, mas apenas ocorre se houver a coincidência do gerador de números aleatórios produzir uma tal permutação
- ◆ Uma alternativa é permutar apenas o primeiro elemento da seqüência com outro, cuja posição é escolhida aleatoriamente
  - ◆ Pergunta: Qual a desvantagem desta abordagem em relação a anterior acima?

24

## Quicksort x Randomized Quicksort

### Quicksort

```
Proc quicksort ( c, i, f )
/* i: início do segmento; f: final do segmento */
begin
  if f > i
  then begin
    partição ( c, i, f, k );
    quicksort ( c, i, k - 1 );
    quicksort ( c, k + 1, f );
  end
end
```

### Randomized Quicksort

```
Proc quicksort_randomizado ( c, i, f )
/* i: início do segmento; f: final do segmento */
begin
  if f > i
  then begin
    partição_randomizada ( c, i, f, k );
    quicksort ( c, i, k - 1 );
    quicksort ( c, k + 1, f );
  end
end

Proc partição_randomizada ( c, i, f, k )
begin
  r ← random(i+1, f);
  exchange ( c[i], c[r] );
  return partição ( c, i, f, k );
end
```