07 – Ordenação em Memória Interna (parte 2) — **quicksort**

SCC201/501 - Introdução à Ciência de Computação II

Prof. Moacir Ponti Jr.
www.icmc.usp.br/~moacir

Instituto de Ciências Matemáticas e de Computação – USP

2010/2

Sumário

História

- **Quicksort** é um algoritmo recursivo atribuído a **Sir Charles Antony Richard Hoare**,
- C.A.R. Hoare nasceu em Colombo no Ceilão (hoje Sri Lanka), de pais britânicos.
- Graduou-se na Universidade de Oxford em 1956.

- Estudou tradução computacional de linguagens humanas em visita à Universidade de Moscou, URSS.
  - durante os estudos, foi preciso realizar a ordenação de palavras a serem traduzidas.
  - o *quicksort* foi o algoritmo desenvolvido por Hoare para ordenar as palavras, em 1960, aos 26 anos.

História: C.A.R. Hoare

- Recebeu o Prêmio Turing da ACM de 1980, por "suas contribuições fundamentais para a definição e projeto de linguagens de programação".
- Em 2009, desculpou-se por ter inventado a referência nula.
- É atualmente pesquisador sênior da Microsoft Research em Cambridge, England e professor emérito da Universidade de Oxford.

- "There are two ways of constructing a software design:
  - One way is to make it so simple that there are obviously no deficiencies, and
  - the other way is to make it so complicated that there are no obvious deficiencies.
- The first method is far more difficult."

Características e estratégia

- **Quicksort** é um algoritmo recursivo que utiliza a estratégia da divisão e conquista
- Não estável.
- Considerada a mais rápida ordenação baseada em comparações em arranjos.
  - Na prática, se bem implementado, executa quase sempre em $\Theta(n \log n)$.
  - No pior caso pode executar em tempo $\Theta(n^2)$.
- O núcleo do método está na **partição** realizada em uma lista a ser ordenada.
  - Essa partição rearranja os elementos de uma lista `A[p..r]` e devolve um índice `i` em `p..r` tal que `A[p..i-1] < A[i] < A[i+1..r]`.
  - O elemento `v = A[i]` é chamado de **pivô**.

Algoritmo

1. Iniciar com uma lista L de n itens
2. Escolher um item pivô v, de L
3. **Particionar** L em duas listas não ordenadas, $L1$ e $L2$
   1. $L1$: conterá todas as chaves menores que v
   2. $L2$: conterá todas as chaves maiores que v
   3. Itens com a mesma chave que v podem fazer parte de $L1$ ou $L2$
   4. O pivô v não faz parte de nenhuma das duas listas
4. **Ordenar**:
   1. $L1$ recursivamente, obtendo a lista ordenada $S1$
   2. $L2$ recursivamente, obtendo a lista ordenada $S2$
5. Concatenar $S1$, v, $S2$ — produzindo a lista ordenada S

Exemplo 1

- O pivô será sempre o primeiro elemento da lista.
- Na fase de partição, formaremos duas sub-listas, $L1$ e $L2$

```
    | 4 | 7 | 1 | 5 | 9 | 3 | 0 |
L1  | 1 | 3 | 0 |
L2  | 7 | 5 | 9 |
```

- $L1$ será ordenada recursivamente.
- como foi alcançado o caso base, as listas serão concatenadas

```
     | 1 | 3 | 0 |
L11  | 0 |
L12  | 3 |
 S1  | 0 | 1 | 3 |
```

Exemplo 1

```
    | 4 | 7 | 1 | 5 | 9 | 3 | 0 |
S1  | 0 | 1 | 3 |
L2  | 7 | 5 | 9 |
```

- *L*2 será ordenada recursivamente.
- como foi alcançado o caso base, as listas serão concatenadas

```
     | 7 | 5 | 9 |
L21  | 5 |
L22  | 9 |
 S2  | 5 | 7 | 9 |
```

Exemplo 1

```
   | 4 | 7 | 1 | 5 | 9 | 3 | 0 |
S1 | 0 | 1 | 3 |
S2 | 5 | 7 | 9 |
```

- após ordenar as sub-listas, é feita a concatenação final

```
   | 0 | 1 | 3 | 4 | 5 | 7 | 9 |
```

Exemplo 2

- Um arranjo de números ordenados

```
    | 0 | 1 | 3 | 4 | 5 | 7 | 9 |
L1  |
L2  | 1 | 3 | 4 | 5 | 7 | 9 |
```

- Desenvolvimento na lousa...
- Quando a entrada já está ordenada o tempo de execução é $\Theta(n^2)$,
- escolher o primeiro item como pivô é uma má escolha para esse caso.

Escolha do pivô

- É crucial para o bom desempenho do método, já que a fase de partição é a parte crítica do algoritmo..
- Há várias estratégias possíveis.

Elemento do meio

- Intuitivamente poderia ser uma boa escolha.
- No entanto, não funciona bem em alguns casos, levando o algoritmo à complexidade quadrática.

Exemplo 3: estratégia de escolha do pivô como elemento do meio

```
    | 1 | 2 | 3 | 4 | 3 | 2 | 1 |
L1  | 1 | 2 | 3 | 3 | 2 | 1 |
L2  |

    L11 | 1 | 2 | 2 | 1 |
    L12 | 3 |

        L111 | 1 | 1 |
        L112 | 2 |

            L1111 | 1 |
            L1112 |
```

Exemplo 3: estratégia de escolha do pivô como elemento do meio

```
      L1111 | 1 |
      L1112 |

    S111 | 1 | 1 |
    S112 | 2 |

  S11 | 1 | 1 | 2 | 2 |
  S12 | 3 |

S1 | 1 | 1 | 2 | 2 | 3 | 3 |
S2 |

   | 1 | 1 | 2 | 2 | 3 | 3 | 4
```

Escolha do pivô

- Há outras opções como a escolha do elemento correspondente à mediana da lista, ou ainda o mais próximo da média.
- Para o caso em que se conhece a **distribuição** dos dados, podemos utilizar a estratégia de escolha do pivô mais adequada àquela distribuição.
- Quando não há conhecimento...

Escolha aleatória

- Escolher aleatoriamente um item da lista L como pivô
- Na média teremos uma partição da lista na proporção: $\frac{1}{4}$ e $\frac{3}{4}$.
- É possível provar que, se a partição da lista ocorrer pelo menos metade das vezes nessa proporção, o **tempo de execução esperado** é $O(n \log n)$.

Escolha do pivô

Mediana de três

1. Escolher três elementos aleatoriamente,
2. Utilizar como pivô o elemento correspondente à mediana dos três.
    - Essa estratégia aumenta ainda mais as chances de se obter caso esperado de $O(n \log n)$.
    - Como há um maior custo em se obter três elementos aleatórios e obter a mediana, essa estratégia é utilizada apenas em listas grandes. Quando a lista a ser ordenada tem tamanhos menores, utiliza-se a escolha aleatória simples.

Exemplo 4: escolha do pivô de forma aleatória

```
| 0 | 1 | 3 | 4 | 5 | 7 | 9 |

| 1 | 0 | 4 | 3 | 5 | 9 | 7 |

| 1 | 2 | 3 | 4 | 3 | 2 | 1 |
```

Sumário

Quicksort em listas ligadas

- Nesse caso é interessante tratar o problema da partição como sendo a partição em **3 listas**:
  - $L1$ contendo chaves menores que o pivô.
  - $L2$ contendo chaves maiores que o pivô.
  - L_v contendo chaves iguais ao pivô.
- A ordenação é realizada apenas em $L1$ e $L2$ e não em L_v.
- A concatenação é realizada na forma: $S1$, L_v, $L2$.

```
   | 5 | 7 | 5 | 0 | 6 | 5 | 5 |
L1 | 0 |
L2 | 7 | 6 |
Lv | 5 | 5 | 5 | 5 |
```

Quicksort em arranjos

- Quicksort é considerado rápido para realizar ordenação *in-place*, ou seja, que utiliza apenas movimentações dentro do próprio arranjo, sem uso de memória auxiliar.
- É importante prestar atenção à implementação para evitar casos de execução quadrática.
- Mesmo alguns livros fornecem algoritmos que podem ser lentos em alguns casos.
- Um algoritmo possível será apresentado a seguir.

Quicksort em arranjos: algoritmo

- Considere um arranjo `A`
- Ordenar os itens de `A[p]` até `A[r]`
- Ao escolher um pivô `v`, substitua-o pelo último item (`A[r]`).
- Vamos utilizar duas variáveis de controle, `i = p-1` e `j = r`:

```
   | 3 | 8 | 4 | 0 | 9 | 7 | 5 |
     p       v               r

   | 3 | 8 | 5 | 0 | 9 | 7 | 4 |
 i                           j
```

- O arranjo será ordenado então para as posições maiores que `i` e menores que `j`.

Quicksort em arranjos: algoritmo

```
  | 3 | 8 | 5 | 0 | 9 | 7 | 4 |
i                           j
```

Invariantes

- No algoritmo proposto há <u>invariantes</u>:
  1. Todos os itens à esquerda de `i` tem chave **menor** ou igual ao pivô.
  2. Todos os itens à direita de `j` tem chave **maior** ou igual ao pivô.

Operações

1. Incrementar `i` até que encontre uma chave maior ou igual ao pivô
2. Decrementar `j` até que encontre uma chave menor ou igual ao pivô
3. Trocar itens `i, j`.
4. Parar quando $i \geq j$. e substituir o pivô de volta à posição inicial.

Sumário

1. História, características e estratégia

2. Algoritmo

3. Escolha do pivô

4. Implementação
   - Listas ligadas
   - Arranjos
   - Código-fonte

Código-fonte

```
int quicksort(int a[], int p, int r) {
   int t;
   if (p < r) {
      int v = (rand()%(r-p))+p; // escolhe pivo aleatoriamente
      int pivo = a[v];
      a[v] = a[r], a[r] = pivo; // troca pivo e ultimo elemento

      int i = p-1, int j = r;
      do {
          do { i ++; } while (a[i] < pivo);
          do { j -; } while ((a[j] > pivo) && (j > p));
          if (i < j)
             t = a[i], a[i] = a[j], a[j] = t; // troca i com j
      } while (i<j);

      a[r] = a[i], a[i] = pivo;
      quicksort(a, p, i-1);
      quicksort(a, i+1, r);
   }
}
```

Exercícios

1. Na página do professor (`www.icmc.usp.br/~moacir`), na seção "Teaching (aulas)", e baixe o arquivo contendo duas implementações do quicksort.
    1. entenda as duas implementações do quicksort e rode o programa para diferentes valores de `MAX` (tamanho do arranjo a ser ordenado)
2. Utilizando como base o código fonte disponível, implemente a estratégia "mediana-de-três" para a escolha do pivô em ambas as implementações do *quicksort*. Verifique se essa estratégia melhora o desempenho do algoritmo.

Bibliografia

- ZIVIANI, N. **Projeto de algoritmos**: com implementações em Pascal e C (Capítulo 4). 2.ed. Thomson, 2004.
- CORMEN, T.H. et al. **Algoritmos: Teoria e Prática** (Capítulo 7). Campus. 2002.
- FEOFILOFF, P. **Quicksort**. Disponível em: `http://www.ime.usp.br/~pf/algoritmos/aulas/quick.html`.
- SHEWCHUCK, J. **CS61B — Data Structures**. Disponível em: `http://www.cs.berkeley.edu/~jrs/61bs09/`.