Algoritmos de Ordenação: *QuickSort*

ACH2002 - Introdução à Ciência da Computação II

Delano M. Beder

Escola de Artes, Ciências e Humanidades (EACH)
Universidade de São Paulo
dbeder@usp.br

10/2008

Material baseado em slides do professor Marcos Chaim

Projeto por Indução Forte

Hipótese de indução forte

Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.

- **Caso base:** n = 1. Um conjunto de um unico elemento está ordenado.

- **Passo da Indução (Segunda Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Sejam S_1 e S_2 os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente. Ambos S_1 e S_2 possuem menos do que n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2.

  Podemos então obter S ordenado concatenando S_1 ordenado, x e S_2 ordenado ($S_1 + x + S_2$).

QuickSort

- Esta indução dá origem ao algoritmo de divisão e conquista *QuickSort*.

- Passos para ordenar um subvetor típico $V[p..r]$ são:
  - **Dividir**: o vetor $V[p..r]$ é particionado (reorganizado) em dois subvetores (possivelmente vazios)
    - $V[p..q-1]$ e $V[q+1..r]$ tais que: $V[p..q-1] \leq V[q] \leq V[q+1..r]$.
    - O índice q é calculado como parte desse particionamento.
  - **Conquistar**: os dois subvetores $V[p..q-1]$ e $V[q+1..r]$ são ordenados por chamadas recursivas ao *QuickSort*.
  - **Combinar**: Como os subvetores são ordenados localmente, não é necessário nenhum trabalho para combiná-los:
    - O vetor $V[p..r]$ no final já está ordenado.

QuickSort

```
void QuickSort(int[] A, int ini, int fim) {
  if (ini < fim) {
    int q = particao(A, ini, fim);
    QuickSort(A, ini, q - 1);
    QuickSort(A, q + 1, fim);
  }
}
```

- No *MergeSort* a operação de **dividir** era bem barata. O caro era **combinar**.
- No *QuickSort*, o caro é **dividir**, enquanto **combinar** não custa nada.

Divisão no *QuickSort*

```
int particao (int[] A, int ini, int fim) {
 int i, j, temp;
 int x = A[fim]; // pivô
 i = ini;
 j = fim - 1;
 while (i <= j) {
   if(A[i] <= x) {
     i++;
   } else if (A[j] > x) {
     j--;
   } else { // trocar A[i] e A[j]
     temp = A[i];
     A[i] = A[j];
     A[j] = temp;
     i++; j--;
   }
 }
 A[fim] = A[i];  // reposicionar o pivô
 A[i] = x;
 return i;
}
```

1. *QuickSort* (A, 0, 7)

1.1. Partição(A, 0, 7)

2	8	7	1	3	5	6	4	i = 0, j = 6
2	8	7	1	3	5	6	4	i = 1, j = 6
2	8	7	1	3	5	6	4	i = 1, j = 5
2	8	7	1	3	5	6	4	i = 1, j = 4
2	3	7	1	8	5	6	4	i = 2, j = 3
2	3	1	7	8	5	6	4	i = 3, j = 2
2	3	1	4	8	5	6	7	pivô = 3

1.2. *QuickSort* (A, 0, 2)

1.2.1 Partição(A, 0, 2)

2	3	1	4	8	5	6	7	i = 0, j = 1
2/2	3	1	4	8	5	6	7	i = 0, j = 0
2	3	1	4	8	5	6	7	i = 0, j = -1
1	3	2	4	8	5	6	7	pivô = 0

1.2.2 *QuickSort* (A, 0, -1) $\times$

1.2.3. *QuickSort* (A, 1, 2)
1.2.3.1. Partição(A, 1, 2)

1	3/3	2	4	8	5	6	7	i = 1, j = 1
1	3	2	4	8	5	6	7	i = 1, j = 0
1	2	3	4	8	5	6	7	pivô = 1

1.2.3.2. *QuickSort* (A, 1, 0) $\times$
1.2.3.3. *QuickSort* (A, 2, 2) $\times$

1.3. *QuickSort* (A, 4, 7)
1.3.1 Partição(A, 4, 7)

1	2	3	4	8	5	6	7	i = 4, j = 6
1	2	3	4	6	5/5	8	7	i = 5, j = 5
1	2	3	4	6	5	8	7	i = 6, j = 4
1	2	3	4	6	5	7	8	pivô = 6

1.3.2. *QuickSort* (A, 4, 5)

1.3.2.1. Partição(A, 4, 5)

1	2	3	4	6/6	5	7	8	i = 4, j = 4
1	2	3	4	6	5	7	8	i = 4, j = 3
1	2	3	4	$\boxed{5}$	$\boxed{6}$	7	8	pivô = 4

1.3.2.2. *QuickSort* (A, 4, 3) $\times$

1.3.2.3. *QuickSort* (A, 5, 5) $\times$

1.3.3. *QuickSort* (A, 7, 7) $\times$

1	2	3	4	5	6	7	8	ordenado

O Desempenho do *QuickSort*

- O desempenho do *QuickSort* depende se o particionamento é balanceado ou não balanceado.

- Para um dado vetor de tamanho n:

  1. Particionamento balanceado mais uniforme possível
     - Dois subproblemas de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$-1.
     - Note que uma posição foi reservada para o pivô, por isso, a soma do tamanho dos dois problema é $n - 1$.
  2. Particionamento desbalanceado, dois subproblemas, um de tamanho $n - 1$ e outro de tamanho 0. Quando isto ocorre?
     - Quando o vetor está ordenado.

- Particionamento balanceado, divisão em subproblemas entre as partições 1 e 2. Exemplo: $n - 2$ e 1.

O Desempenho do *QuickSort*

- Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto o *MergeSort*.

  - Isto é, $O(n \log n)$.
  - Vantagem adicional em relação ao *MergeSort*: é *in place*, isto é, não utiliza um vetor auxiliar.
  - Note-se que basta ser balanceado, não precisa ser o particionamento mais uniforme!

- Contudo, se o particionamento não é balanceado, ele pode ser executado tão lentamente quanto os algoritmos *InsertionSort*, *SelectionSort* e *BubbleSort*.

  - Isto é, $O(n^2)$.

O Desempenho do *QuickSort* – Pior Caso

- Pior caso do *QuickSort* ocorre quando a rotina de particionamento produz um subproblema com $n-1$ elementos e um com zero elementos.
    - O custo da partição é $\Theta(n)$. Por quê?
        - Para fazer o particionamento, serão necessárias *sempre n* comparações, ou seja, $\Theta(n)$.
- Supondo que esse particionamento não balanceado surge em cada chamada recursiva, vamos ter a seguinte a seguinte equação de recorrência.

$$T(n) = T(n-1) + T(0) + n$$

- Para $T(0) = \Theta(1)$, pois a chamada sobre um vetor de tamanho 0 simplesmente retorna.

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

Particionamento pior caso

- Expandindo a equação de recorrência, pode-se concluir que
  - $T(n)$ é $\Theta(n^2)$

$$
\begin{aligned}
&T(n) = T(n-1) + n = T(n-2) + n - 1 + n\\
&T(n) = T(n-3) + n - 2 + n - 1 + n\\
&...\\
&T(n) = T(n-k) + nk - \textstyle\sum_{i=0}^{k-1} i\\
&T(n) = T(n-k) + nk - k(\frac{k-1}{2})\\
&onde\ n - k = 0 \Rightarrow n = k
\end{aligned}
$$

- Logo, obtemos:

$$
\begin{aligned}
&T(n) = \Theta(1) + n^2 - \tfrac{n^2-n}{2}\\
&T(n) = \Theta(1) + \tfrac{2n^2-n^2+n}{2}\\
&T(n) = \Theta(1) + \tfrac{n^2}{2} + \tfrac{n}{2}\\
&T(n) = \Theta(n^2)
\end{aligned}
$$

Particionamento melhor caso

- Na divisão mais uniforme possível, o particionamento produz dois subproblemas não maiores que $n/2$:
  - Um tem tamanho $\lfloor n/2 \rfloor$ e outro tem tamanho $\lceil n/2 \rceil - 1$.
  - Neste caso *QuickSort* é executado com muito maior rapidez.
- A recorrência para o tempo de execução é:

$$T(n) \leq 2T(n/2) + \Theta(n)$$

- Essa recorrência é semelhante à do *MergeSort* e sabemos que

$$T(n) = O(n \log n)$$

Particionamento balanceado

- O tempo de execução do caso médio do *QuickSort* é muito mais próximo do melhor caso do que do pior caso.
- Para entender isto, considere um particionamento que **sempre** produza uma divisão proporcional de 9 para 1.
- Isto parece bastante desequilibrado e possui a seguinte recorrência:

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

- Esta recorrência *na verdade* possui a solução $T(n) = O(n \log n)$.
  - (Ver página 122 do Cormen [1]).

Intuição para o caso médio

- Caso médio $\Rightarrow$ todas as permutações dos números de entrada são igualmente prováveis.
- Para um vetor de entrada aleatória é pouco provável que o particionamento ocorra sempre do mesmo modo em todo nível (como suposto na análise informal anterior).
- Esperado $\Rightarrow$ algumas divisões **boas** e outras **ruins**, distribuídas aleatoriamente ao longo da árvore.

Intuição para o caso médio

- Suposição: uma divisão **ruim** (pior caso) seguida de uma **boa** (melhor caso).
  - Primeira divisão (ruim): $T(0) + T(n-1)$
  - Segunda divisão (boa): $T(\frac{n-1}{2}) + T(\frac{n-1}{2})$
  - Total: $T(0) + T(\frac{n-1}{2}) + T(\frac{n-1}{2})$
  - O custo da divisão ruim é absorvido pela divisão boa.
- Portanto, o tempo de execução do *QuickSort* quando os níveis se alternam entre divisões boas e ruins é semelhante ao custo para divisões boas sozinhas:
  - $T(n) = O(n \log n)$, mas com uma constante maior.

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis.
  - Cheques são registrados na ordem temporal (data) em que são descontados.
  - O banco pode querer uma listagem dos cheques ordenados pelo número do cheque (e.g., 86781, 86782,...).
  - Normalmente, os cheques são emitidos e descontados na ordem do talão, mas nem sempre. Às vezes, o comerciante pode dar um prazo a mais (cheque pré-datado) e demorar para descontar o cheque.
  - Em resumo: a seqüência temporal dos cheques está quase ordenada por número de cheque.
- Nesta situação, o *QuickSort* vai ter um desempenho ruim. É então necessário aproximar-se o caso médio. Como?

Aproximando-se do caso médio

Uma maneira de aproximar-se do caso é escolhendo um pivô aleatoriamente, ao invés de utilizar sempre o último elemento.

```
int particaoAleatoria (int[] A, int ini, int fim) {
  int i, temp;
  double f;
  // Escolhe um número aleatório entre ini e fim
  f = java.lang.Math.random();
  // retorna um real f tal que 0 <= f < 1
  i = (int) (ini + (fim - ini) * f);
  // i é tal que ini <= i < fim
  // Troca de posicao A[i] e A[fim]
  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Aproximando-se do caso médio

Como é que fica o *QuickSort* com a partição aleatória?

QuickSort Aleatório

```
void quickSortAleatorio(int[] A, int ini, int fim) {
  if (ini < fim) {
    int q = particaoAleatoria(A, ini, fim);
    quickSortAleatorio(A, ini, q - 1);
    quickSortAleatorio(A, q + 1, fim);
  }
}
```

Resumo

- *QuickSort*:
  - Ordenação local (*in-place*): utiliza quantidade de memória constante além do próprio vetor.
  - Pior caso: $\Theta(n^2)$.
  - Melhor caso: $O(n \log n)$.
  - Caso médio: $O(n \log n)$.

Referências

Referências utilizadas: [1] (páginas 117-124)

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. *Algoritmos - Tradução da 2a. Edição Americana*. Editora Campus, 2002.