AULA 06 – QUICKSORT

Prof. Daniel Kikuti

Universidade Estadual de Maringá

13 de agosto de 2014

Conteúdo

- Descrição do algoritmo Quicksort.
- Correção.
- Análise de desempenho.
- Versão aleatória.
- Exercícios.

Introdução

- Quicksort é um algoritmo de ordenação proposto por C.A.R. Hoare (1960).
- Características:
  - Tempo de execução no pior caso: $\Theta(n^2)$.
  - Tempo de execução esperado: $\Theta(n \lg n)$.
  - Os fatores constantes escondidos em $\Theta(n \lg n)$ são pequenos (bom desempenho na prática).
  - Ordenação local.

Descrição do `Quicksort`

- Baseado no paradigma **divisão e conquista**
- Para ordenar um vetor $A[p..r]$:
  - **Dividir**: divida o vetor $A[p..r]$ em dois subvetores $A[p..q-1]$ e $A[q+1..r]$, tal que $A[p..q-1] \leq A[q] \leq A[q+1..r]$.
  - **Conquistar**: Ordenar os dois subvetores *recursivamente* usando o `Quicksort`.
  - **Combinar**: Como os subvetores são ordenados localmente, não é necessário nenhum trabalho para combiná-los.

Descrição do `Quicksort`

- Baseado no paradigma **divisão e conquista**
- Para ordenar um vetor $A[p..r]$:
  - **Dividir**: divida o vetor $A[p..r]$ em dois subvetores $A[p..q-1]$ e $A[q+1..r]$, tal que $A[p..q-1] \leq A[q] \leq A[q+1..r]$.
  - **Conquistar**: Ordenar os dois subvetores *recursivamente* usando o `Quicksort`.
  - **Combinar**: Como os subvetores são ordenados localmente, não é necessário nenhum trabalho para combiná-los.

Importante

O passo de divisão é feito pelo procedimento `Partition`, que devolve o índice q que marca a posição de divisão dos subvetores.

Procedimento `Quicksort`

Rearranja um vetor $A[p..r]$ em ordem crescente.

```
Quicksort(A, p, r)
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

- Para ordenar todo um array A, a chamada inicial é `Quicksort(A, 1, A.comprimento)`.
- Antes de entender o `Quicksort`, temos que entender o `Partition`.

O que faz o procedimento `Partition`?

Problema

Rearranjar $A[p..r]$ e devolver um índice q, $p \leq q \leq r$, tal que:

$$A[p..q-1] \leq A[q] \leq A[q+1..r].$$

Entrada

$$A = [8, 1, 6, 4, 0, 3, 9, 5]$$

Saída

$$A = [1, 4, 0, 3, 5, 8, 9, 6]$$

$$\mathrm{q} = 5 \text{ (índice)}.$$

Procedimento Partition

```
Partition(A, p, r)
1 x = A[r]   //x é o pivo
2 i = p - 1
3 for j = p to r - 1
4  if A[j] <= x
5    i = i + 1
6    troca(A[i], A[j])
7 troca(A[i+1], A[r])
8 return i + 1
```

Invariantes

No começo de cada iteração da linha 3:

1. $A[p..i] \leq x$.
2. $A[i+1..j-1] > x$.
3. $A[r] = x$.

Correção do `Partition`

Demonstração (versão completa no livro-texto)

- **Inicialização**: Antes do início do laço todas as condições da invariante são satisfeitas, porque o r é o pivô e os subvetores $A[p..i]$ e $A[i+1..j-1]$ são vazios.
- **Manutenção**: Durante a execução do laço, se $A[j] \leq x$, então $A[j]$ e $A[i+1]$ são trocados e i e j são incrementados. Se $A[j] > x$, apenas o j é incrementado.
- **Término**: Quando o laço termina, $j = r$, todos os elementos de A estão particionados em uma das três casos: $A[p..i] \leq x$, $A[i+1..j-1] > x$ e $A[r] = x$.

Desempenho do Partition

Complexidade de tempo ($n = r - p + 1$)

```
Partition(A, p, r)                    Tempo
1 x = A[r] //x é o pivo
2 i = p - 1
3 for j = p to r - 1
4    if A[j] <= x
5       i = i + 1
6       troca(A[i], A[j])
7 troca(A[i+1], A[r])
8 return i + 1
```

Desempenho do Partition

Complexidade de tempo ($n = r - p + 1$)

Partition(A, p, r)	Tempo
1 x = A[r] //x é o pivo	$\Theta(1)$
2 i = p - 1	$\Theta(1)$
3 for j = p to r - 1	$\Theta(n)$
4 if A[j] <= x	$\Theta(n)$
5 i = i + 1	$O(n)$
6 troca(A[i], A[j])	$O(n)$
7 troca(A[i+1], A[r])	$\Theta(1)$
8 return i + 1	$\Theta(1)$

Conclusão

$$T(n) = 2\Theta(n) + 4\Theta(1) + 2O(n) = \Theta(n).$$

A complexidade de Partition é $\Theta(n)$.

Procedimento Quicksort

Rearranja um vetor $A[p..r]$ em ordem crescente.

```
Quicksort(A, p, r)
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

Considere a entrada:

$$A = [8, 1, 6, 4, 0, 3, 9, 5]$$

Procedimento Quicksort

Rearranja um vetor $A[p..r]$ em ordem crescente.

```
Quicksort(A, p, r)
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

Considere a entrada:

$$A = [8, 1, 6, 4, 0, 3, 9, 5]$$

Após a execução da linha 2 temos:

$$A = [1, 4, 0, 3, \mathbf{5}, 8, 9, 6]$$

Procedimento Quicksort

Rearranja um vetor $A[p..r]$ em ordem crescente.

```
Quicksort(A, p, r)
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

Considere a entrada:

$$A = [8, 1, 6, 4, 0, 3, 9, 5]$$

Chamada recursiva da linha 3 para a parte destacada:

$$A = [\mathbf{1}, \mathbf{4}, \mathbf{0}, \mathbf{3}, 5, 8, 9, 6]$$

Procedimento Quicksort

Rearranja um vetor $A[p..r]$ em ordem crescente.

```
Quicksort(A, p, r)
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

Considere a entrada:

$$A = [8, 1, 6, 4, 0, 3, 9, 5]$$

Chamada recursiva da linha 4 para a parte destacada:

$$A = [1, 4, 0, 3, 5, \mathbf{8}, \mathbf{9}, \mathbf{6}]$$

Complexidade do Quicksort

Complexidade de tempo ($n = r - p + 1$)

```
Quicksort(A, p, r)                    Tempo
1 if p < r
2   q = Partition(A, p, r)
3   Quicksort(A, p, q - 1)
4   Quicksort(A, q + 1, r)
```

Complexidade do Quicksort

Complexidade de tempo ($n = r - p + 1$)

`Quicksort(A, p, r)`	Tempo
`1 if p < r`	$\Theta(1)$
`2  q = Partition(A, p, r)`	$\Theta(n)$
`3  Quicksort(A, p, q - 1)`	$T(k)$
`4  Quicksort(A, q + 1, r)`	$T(n-k-1)$

Portanto

$$T(n) = T(k) + T(n-k-1) + \Theta(n+1),$$

$$0 \leq k = q - p \leq n - 1.$$

Recorrência

$$T(n) = \begin{cases} \Theta(0) & n = 0 \\ \Theta(1) & n = 1 \\ T(k) + T(n-k-1) + \Theta(n) & n \geq 2 \end{cases}$$

- O tempo de execução do `Quicksort` depende do particionamento dos subvetores.
- **Caso ruim**: quando os subvetores estão completamente desbalanceados: um com **0** elementos e outro com **n-1**.

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Análise do pior caso

Podemos demonstrar que o caso ruim apontado anteriormente é o pior caso se considerarmos a recorrência:

$$T(n) = \begin{cases} \Theta(0) & n = 0 \\ \Theta(1) & n = 1 \\ \max\limits_{0 \leq k \leq n-1} \{T(k) + T(n-k-1)\} + \Theta(n) & n \geq 2 \end{cases}$$

Usando o método da substituição ($T(n) \leq cn^2$):

$$\begin{aligned} T(n) &= \max_{0 \leq k \leq n-1} \{T(k) + T(n-k-1)\} + dn \\ &\leq \max_{0 \leq k \leq n-1} \{ck^2 + c(n-k-1)^2\} + dn \\ &= c. \max_{0 \leq k \leq n-1} \{k^2 + (n-k-1)^2\} + dn \end{aligned}$$

A expressão $k^2 + (n-k-1)^2$ atinge valor maximo quando $k = 0$ ou $k = n-1$ [Exercício CLRS], provando que a divisão em subvetores de tamanhos 0 e $n-1$ é de fato o pior caso.

Análise do pior caso – continuação

Então fazendo $k = 0$ (ou $k = n - 1$) na expressão $k^2 + (n - k - 1)^2$, temos:

$$\begin{aligned} T(n) &\leq c(n-1)^2 + dn \\ &= cn^2 - 2cn + c + dn \\ &\leq cn^2, \end{aligned}$$

para $c > d/2$ e $n_0 \geq c/(2c - d)$ (d é uma constante positiva).

Exercício

Prove que $T(n) = \Omega(n^2)$.

Análise no melhor caso

- Ocorre quando os subarrays estão balanceados.
- Um subarray tem tamanho $\lfloor n/2 \rfloor$ e o outro tem tamanho $\lceil n/2 \rceil - 1$.
- Obtemos a recorrência:

$$\begin{aligned} T(n) &\leq 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \text{caso 2 do teorema mestre.} \end{aligned}$$

Particionamento constante

- O tempo médio de execução do `Quicksort` é muito mais próximo do melhor caso do que do pior caso.
- Suponha que o algoritmo de particionamento sempre produza uma divisão na proporção 9 para 1.
- Obtemos a recorrência

$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n). \end{aligned}$$

- Por que? Vejamos a árvore de recursão.

Árvore de recursão

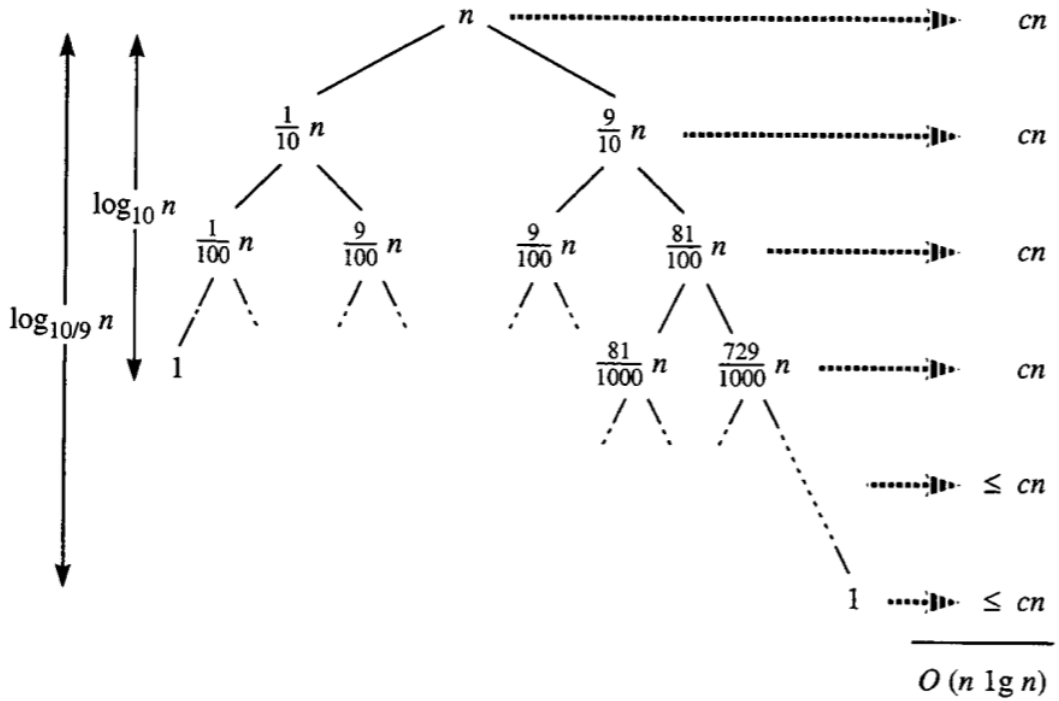


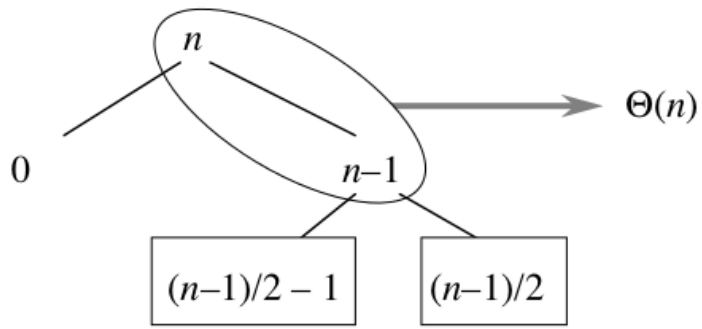
FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo $\Theta(n)$

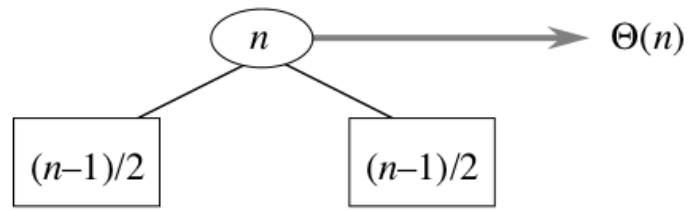
Particionamento constante

- O custo de cada nível é $O(n)$.
- Temos $\log_{10} n$ níveis completos e $\log_{10/9} n$ níveis não vazios.
- Desde que seja constante, a base do logaritmo não importa para a notação assintótica.
- Qualquer divisão de proporção constante gerará uma árvore de recursão de profundidade $\Theta(\lg n)$.

Intuição para o caso médio

- A proporção das divisões na árvore de recursão não será sempre constante.
- No caso médio, haverá uma mistura de divisões boas e ruins.
- Para facilitar o entendimento, suponha que que as divisões boas e ruins alternem seus níveis na árvore.





- O custo da divisão ruim adiciona uma constante oculta na notação Θ.
- Em ambas as figuras, o tempo de execução é $O(n \lg n)$.

Versão aleatória do `Quicksort`

- Para explorar o caso médio, assumimos que todas as permutações de entrada são igualmente possíveis.
- O que nem sempre é verdade.
- Para corrigir esta situação, adicionamos aleatoriedade ao `Quicksort`.
- A ideia é não usar sempre $A[r]$ como pivô. Ao invés, escolhemos um elemento do vetor aleatoriamente.
- Como o pivô é escolhido aleatoriamente, esperamos que a divisão do vetor de entrada seja equilibrada na média.

Versão aleatória do quicksort

```
randomized-partition(A, p, r)
1 i = random(p, r)
2 troca(A[r], A[i])
3 return partition(A, p, r)

randomized-quicksort(A, p, r)
1 if p < r
2   q = randomized-partition(A, p, r)
3   randomized-quicksort(A, p, q - 1)
3   randomized-quicksort(A, q + 1, r)
```

Exercícios

7.1-1 Usando a figura 7.1 (ver livro-texto) como modelo, ilustre a operação de `Partition` sobre o array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1-2 Que valor de q `Partition` retorna quando todos os elementos no arranjo $A[p..r]$ têm o mesmo valor? Modifique `Partition` de forma que $q = (p + r)/2$ quando todos os elementos no array $A[p..r]$ têm o mesmo valor.

7.1-3 Forneça um breve argumento mostrando que o tempo de execução de `Partition` sobre um subarray de tamanho n é $\Theta(n)$.

7.1-4 De que maneira você modificaria `Quicksort` para fazer a ordenação em ordem não crescente?

Exercícios

7.2-1 Use o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$.

7.2-2 Qual o tempo de execução de `quicksort` quando todos os elementos do array A têm o mesmo valor.

7.2-3 Mostre que o tempo de execução de `Quicksort` é $\Theta(n^2)$ quando o array A contém elementos distintos e está ordenado em ordem decrescente.

Exercícios

7.3-1 Por que analisamos o desempenho do caso médio de um algoritmo aleatório e não o seu desempenho no pior caso?

7.3-2 Durante a execução do procedimento `randomized-quicksort`, quantas chamadas são feitas ao gerador de números aleatórios `random` no pior caso? E no melhor caso? Dê a resposta em termos da notação Θ.

Exercícios

- Exercícios 7.1-1 a 7.1-4
- Exercícios 7.2-1 a 7.2-3
- Exercícios 7.3-1 a 7.3-2

Referências

- Thomas H. Cormen et al. Introdução a Algoritmos. 2ª edição em português. Capítulo 7.