MAC 5711 - Análise de Algoritmos

Rodrigo Augusto Dias Faria
Departamento de Ciência da Computação - IME/USP

21 de setembro de 2015

Lista 1

1. Lembre-se que lg n denota o logaritmo na base 2 de n. Usando a definição de notação O, prove que

(a) 3^n não é $O(2^n)$

Vamos assumir por contradição que 3^n é $O(2^n)$, então podemos assumir que existem as variáveis $c > 0$ e $n_0 > 0$ tal que:

$$3^n \leq c2^n, \forall n \geq n_0$$

Vamos dividir os dois lados por 2^n:

$$\frac{3^n}{2^n} \leq \frac{c2^n}{2^n}, \forall n \geq n_0$$
$$\frac{3^n}{2^n} \leq c, \forall n \geq n_0$$

Note que $3^n > 2^n$ e quando $n \to \infty$, então $\frac{3^n}{2^n} \to \infty$, logo podemos concluir que não importa quão grande seja a constante c sempre vai existir algum n suficientemente grande tal que:

$$3^n > c2^n$$

Portanto podemos concluir que $3^n \notin O(2^n)$. $\square$

(b) $\log_{10} n$ é $O(\lg n)$

Se existem as constantes $c > 0$ e $n_0 > 0$ tal que:

$$\log_{10} n \leq c \lg n, \forall n \geq n_0$$

então $\log_{10} n$ é $O(\lg n)$. Veja que uma das propriedades dos logaritmos nos diz que:

$$\log_c a = \frac{\log_b a}{\log_b c}$$

Disso concluimos que:

$$\log_{10} n = \frac{\lg n}{\lg 10}$$

Portanto se fizermos $c = \frac{1}{\lg 10}$ e $n_0 = 1$ temos que

$$\log_{10} n \leq \frac{\lg n}{\lg 10}, \forall n \geq 1$$

Portanto podemos concluir que $\log_{10} n = O(\lg n)$. $\square$

(c) lg n é $O(\log_{10} n)$

Se existem as constantes $c > 0$ e $n_0 > 0$ tal que:

$$\lg n \leq c \log_{10} n, \forall n \geq n_0$$

Então $\lg n = O(\log_{10} n)$. Note que pela propriedade dos logaritmos mostrada no exercício anterior podemos concluir que:

$$\lg n = \frac{\log_{10} n}{\log_{10} 2}$$

Logo se fizermos $c = \frac{1}{\log_{10} 2}$ e $n_0 = 1$ então teremos:

$$\lg n \leq \frac{\log_{10} n}{\log_{10} 2}, \forall n \geq 1$$

Portanto podemos concluir que $\lg n = O(\log_{10} n)$. $\square$

Lista 2

2. Escreva um algoritmo que ordena uma lista de n itens dividindo-a em três sublistas de aproximadamente $n/3$ itens, ordenando cada sublista recursivamente e intercalando as três sublistas ordenadas. Analise seu algoritmo concluindo qual é o seu consumo de tempo.

Para este exercício, devemos efetuar uma alteração no MERGESORT para a divisão do vetor A em três partições utilizando o MERGE duas vezes ao final para intercalar as três partes ordenadas em um único vetor.

```
MERGESORT3(A, p, r)
1  if p < r
2      k = ⌊(p + r)/3⌋
3      m = k + 1 + ⌊(p + r)/3⌋
4      MERGESORT3(A, p, k)
5      MERGESORT3(A, k + 1, m)
6      MERGESORT3(A, m + 1, r)
7      MERGE(A, p, k, m)
8      MERGE(A, p, m, r)
```

Consumo de tempo

As linhas 1-3 consomem $\Theta(1)$. As linhas 4-5 têm consumo $T(\lceil n/3 \rceil)$ e a linha 6 tem consumo $T(n - \lceil 2n/3 \rceil)$, já que a terceira partição não tem tamanho exatamente de $\lceil n/3 \rceil$. Sabemos que o consumo do MERGE é $\Theta(n)$, logo:

$$\begin{aligned} T(n) &= T(\lceil n/3 \rceil) + T(\lceil n/3 \rceil) + T(n - 2\lceil n/3 \rceil) + \Theta(n) + \Theta(n) \\ &= 2T(\lceil n/3 \rceil) + T(\lceil n/3 \rceil) + \Theta(2n) \\ &= 3T(\lceil n/3 \rceil) + \Theta(2n) \end{aligned}$$

Como $\Theta(2n)$ é $\Theta(n)$:

$$T(n) = 3T(\lceil n/3 \rceil) + \Theta(n)$$

Simplificando a recorrência, temos:

$$T(n) = \begin{cases} 1, & n = 1 \\ 3T\left(\dfrac{n}{3}\right) + n, & n >= 2 \text{ potência de 2} \end{cases}$$

Por expansão:

$$
\begin{aligned}
T(n) &= 3T\left(\frac{n}{3}\right) + n \\
&= 3\left(3T\left(\frac{n}{3^2}\right) + \left(\frac{n}{3}\right)\right) + n & &= 3^2T\left(\frac{n}{3^2}\right) + n + n \\
&= 3^2\left(3T\left(\frac{n}{3^3}\right) + \left(\frac{n}{3^2}\right)\right) + n + n & &= 3^3T\left(\frac{n}{3^3}\right) + n + n + n \\
&= ... \\
&= 3^kT\left(\frac{n}{3^k}\right) + kn
\end{aligned}
$$

Assumindo $k = \log_3 n$ e $3^k = n$:

$$
\begin{aligned}
T(n) &= nT\left(\frac{n}{n}\right) + \log_3 n \\
&= T(1)n + \log_3 n(n) \\
&= n + n(\log_3 n)
\end{aligned}
$$

Portanto, $T(n) = n + n(\log_3 n)$ é $\Theta(n \log n)$.

Demonstração. Prova por indução em k.

Base: para $n = 1$

$$T(1) = 1 = 1 + 1(\log_3 1) = 1 + 0 = 1$$

Hipótese de Indução: Assuma que $T(x) = x + x(\log_3 x)$ vale para $1 >= x < n$

Passo: para $n >= 2$

$$
\begin{aligned}
T(n) = 3T\left(\frac{n}{3}\right) + n &= 3\left(\left(\frac{n}{3}\right) + \left(\frac{n(\log_3 \frac{n}{3})}{3}\right)\right) + n & &\text{(por HI)} \\
&= 3\left(\frac{n}{3}\right) + 3\left(\left(\frac{n}{3}\right)\log_3 \frac{n}{3}\right) + n \\
&= n + n + n \log_3 \frac{n}{3} \\
&= 2n + n \log_3 n - n \\
&= n + n \log_3 n
\end{aligned}
$$

Como queríamos demonstrar!

$\square$

Lista 3

2. Qual é o consumo de espaço do QUICKSORT no pior caso?

A avaliação de um algoritmo quanto ao consumo de espaço está relacionada com a necessidade de alocação de espaço adicional na pilha de recursão.

No pior caso, o QUICKSORT será executado uma vez para cada elemento da lista dada de tamanho n, ou seja, teremos n chamadas recursivas.

Isso significa que, com uma lista de n elementos, n novas chamadas serão adicionadas à pilha no pior caso, o que nos leva a uma complexidade de espaço O(n).

3. Quando um algoritmo recursivo tem como último comando executado, em algum de seus casos, uma chamada recursiva, tal chamada é denominada recursão de calda (*tail recursion*). Um exemplo de recursão de calda acontece no QUICKSORT.

```
Quicksort(A, p, r)
  q = Partition(A, p, r)
  Quicksort(A, p, q - 1)
  Quicksort(A, q + 1, r)
```

Toda recursão de calda pode ser substituída por uma repetição. No caso do Quicksort, obtemos o seguinte:

```
Quicksort(A, p, r)
  while p < r
      q = Partition(A, p, r)
      Quicksort(A, p, q - 1)
      p = q + 1
```

Mostre como essa ideia pode ser usada (de uma maneira mais esperta) para melhorar significativamente o consumo de espaço no pior caso do Quicksort.

O benefício da utilização do loop ao invés da recursão de cauda é que, geralmente, precisamos de menos memória na pilha para ordenar todos os elementos do vetor A, já que a implementação sem a recursão de cauda reusa o ambiente da pilha (variáveis locais, parâmetros, etc) a cada iteração.

A profundidade das chamadas recursivas está relacionada com a ordem em que os elementos se encontram. Se, por exemplo, o vetor A está em ordem decrescente, teremos a execução no pior caso. Isso implica na forma em que cada partição é gerada.

Para ter um resultado ainda mais eficiente, os intervalos podem ser comparados para se certificar de que a maior partição é sempre processada de forma iterativa e a menor de forma recursiva, o que garante a menor profundidade de recursão possível para um determinado vetor de entrada e pivô.

```
HALF-TAIL-QUICKSORT(A, p, r)
while p < r
    q = PARTITION(A, p, r)
    if (q - p) < (r - q)
        HALF-TAIL-QUICKSORT(A, p, q - 1)
        p = q + 1
    else
        HALF-TAIL-QUICKSORT(A, q + 1, r)
        r = q - 1
```

4. Considere o seguinte algoritmo que determina o segundo maior elemento de um vetor $v[1..n]$ com $n >= 2$ números positivos distintos.

```
Algoritmo Máximo (v, n)
1. maior ← 0
2. segundo_maior ← 0
3. para i ← 1 até n faça
4.     se v[i] > maior
5.         então segundo_maior ← maior
6.               maior ← v[i]
7.         senão se v[i] > segundo_maior
8.               então segundo_maior ← v[i]
9. devolva segundo_maior
```

Suponha que a entrada do algoritmo é uma permutação de 1 a n escolhida uniformemente dentre todas as permutaçõees de 1 a n. Qual é o número esperado de comparações executadas na linha 6 do algoritmo? Qual é o número esperado de atribuições efetuadas na linha 7 do algoritmo?

Vamos calcular $E[X]$ para o algoritmo dado. Seja:
A = número de vezes que a linha 5 do algoritmo foi executada
B = número de vezes que a linha 8 do algoritmo foi executada

X = A + B

$$X_i = \begin{cases} 1, & \text{se A ou B} \\ 0, & \text{c.c} \end{cases}$$

Logo:
$E[X_i] = 0 * Pr\{X_i = 0\} + 1 * Pr\{X_i = 1\} = Pr\{X_i = 1\}$

Sabemos que a execução da linha 8 depende da avaliação da linha 5, logo:
$E[X_i] = Pr\{A\} + (1 - Pr\{A\}) * Pr\{B|\bar{A}\}$

Como já vimos para a versão original do algoritmo MÁXIMO é $Pr\{A\} = \frac{1}{i}$.
Para a probabilidade de execução da linha 8, temos que:
$Pr\{B|\bar{A}\} = \frac{1}{1-i}$

Portanto:

$$\begin{aligned} E[X_i] &= \left(\frac{1}{i}\right) + \left(1 - \frac{1}{i}\right)\left(\frac{1}{i-1}\right) \\ &= \left(\frac{1}{i}\right) + \left(\frac{i-1}{i}\right)\left(\frac{1}{i-1}\right) \\ &= \frac{2}{i} \end{aligned}$$

É fato que a linha 5 sempre será executada na primeira iteração, assumindo que $v[1..n]$ contenha apenas inteiros positivos e $n >= 2$. Logo:

$$E[X_i] = 1 + \sum_{i=2}^{n} \frac{2}{i} = 1 + 2\sum_{i=2}^{n} \frac{1}{i}$$

6. (**CLRS 8.4-3**) Seja X uma variável aleatória que é igual ao número de caras em duas jogadas de uma moeda justa. Quanto vale $E[X^2]$? Quanto vale $E[X]^2$?

Como temos dois lançamentos, o espaço amostral é dado por:

$$S = \{CC, CK, KC, KK\}$$

Sabendo que a $Pr\{X = cara\} = 1/4$, temos que $E[X]$:

$$\begin{aligned} E[X] &= 2 * \frac{1}{4} + 1 * \frac{1}{4} + 1 * \frac{1}{4} + 0 * \frac{1}{4} \\ &= \frac{2+1+1}{4} + 0 \\ &= 1 \end{aligned}$$

Logo, para $E[X^2]$, temos:

$$\begin{aligned} E[X^2] &= 2^2 * \frac{1}{4} + 1^2 * \frac{1}{4} + 1^2 * \frac{1}{4} + 0^2 * \frac{1}{4} \\ &= \frac{4+1+1}{4} + 0 \\ &= \frac{3}{2} \end{aligned}$$

e para $E^2[X]$, temos o produto das esperanças de X:

$$\begin{aligned} E^2[X] &= E[X] * E[X] \\ &= 1 * 1 \\ &= 1 \end{aligned}$$

Lista 4

1. Escreva uma função que recebe um vetor com n letras A's e B's e, por meio de trocas, move todos os A's para o início do vetor. Sua função deve consumir tempo O(n).

Resposta

3. Sejam $X[1..n]$ e $Y[1..n]$ dois vetores, cada um contendo n números ordenados. Escreva um algoritmo O($\lg n$) para encontrar uma das medianas de todos os $2n$ elementos nos vetores X e Y.

Sabemos que a mediana de X e Y está em $i = \lfloor q/2 \rfloor$ e $j = \lfloor s/2 \rfloor$, respectivamente. Note que $n = q + s$ é par, e é por isso que nós estamos usando a função **piso**.

Se $X[i]$ é maior do que $Y[j]$, significa que a mediana global está à esquerda de $X[i]$ e à direita de $Y[j]$. Se $X[i]$ é menor ou igual a $Y[j]$, nós procuramos a mediana à esquerda de $Y[j]$ e à direita de $X[i]$.

A condição de parada dá-se quando $p == q$, o que significa que a mediana global está dentro do vetor X. Caso contrário, se $r == s$, a mediana está em Y.

O pseudocódigo FIND-MEDIAN mostra a operação descrita acima que, também, é o resultado do exercício 9.3-8 CLRS 3ed.

```
FIND-MEDIAN(X, Y, p, q, r, s)
if p == q
// We have found the median between p, q and r
    return X[p]
elseif r == s
// We have found the median between q, r and s
    return Y[r]
i = p + (q - p)/2
j = r + (s - r)/2
if X[i] > Y[j]
    q = i
    r = j
else
    p = i
    s = j
return FIND-MEDIAN(X, Y, p, q, r, s)
```

4. (**CLRS 9.3-5**) Para esta questão, vamos dizer que a mediana de um vetor $A[p..r]$ com números inteiros é o valor que ficaria na posição $A[\lfloor (p + r)/2 \rfloor]$ depois que o vetor $A[p..r]$ fosse ordenado.

Dado um algoritmo linear "caixa-preta" que devolve a mediana de um vetor, descreva um algoritmo simples, linear, que, dado um vetor $A[p..r]$ de inteiros distintos e um inteiro k, devolve o *k-ésimo* mínimo do vetor. (O *k-ésimo* mínimo de um vetor de inteiros distintos é o elemento que estaria na *k-ésima* posição do vetor se ele fosse ordenado.)

Assumindo que o procedimento MEDIAN retorna a mediana do vetor $A[p..r]$ em tempo linear, a versão modificada do SELECT abaixo retorna, então, o *k-ésimo* menor elemento de $A[p..r]$.

O algoritmo usa o PARTITION determinístico para pegar um elemento da partição e utilizá-lo como parâmetro de entrada.

```
SELECTION(A, p, r, k)
if p == r
    return A[p]

x = MEDIAN(A, p, r)
q = PARTITION(x)
k = q − p + 1
if i == k
    return A[q]
elseif k < i
    return SELECTION(A, p, q − 1, k)
else
    return SELECTION(A, p, q + 1, r, k − i)
```

8. (**CLRS 8.3-2**) Quais dos seguintes algoritmos de ordenação são estáveis: insertionsort, mergesort, heapsort, e quicksort. Descreva uma maneira simples de deixar qualquer algoritmo de ordenação estável. Quanto tempo e/ou espaço adicional a sua estratégia usa?

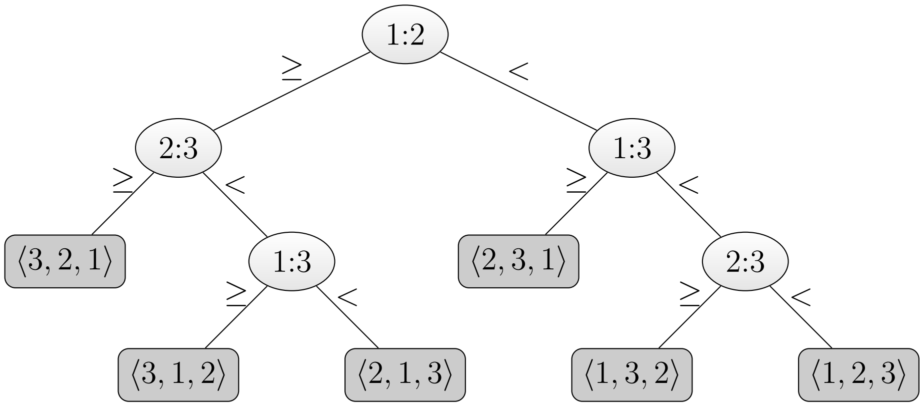
Os algoritmos estáveis são o insertionsort e o mergesort (versão do Cormen). Os demais não são estáveis.

Uma forma simples de deixar qualquer algoritmo de ordenação estável é criar um mecanismo de indexação que mantenha a ordem em que os elementos aparecem originalmente, ou seja, basta termos um índice para cada elemento de um vetor de n elementos.

Esse mecanismo necessita de $\Theta(n)$ espaço extra para armazenar os n índices do vetor de n elementos.

Lista 5

1. Desenhe a árvore de decisão para o SELECTIONSORT aplicado a $A[1..3]$ com todos os elementos distintos.



2. **(CLRS 8.1-1)** Qual a menor profundidade (= menor nível) que uma folha pode ter em uma árvore de decisão que descreve um algoritmo de ordenação baseado em comparações?

A menor profundidade da árvore de decisão é, coincidentemente, a cota inferior das alturas de todas as árvores de decisão nas quais aparecem uma folha (uma das $n!$ permutações da entrada).

Também podemos dizer que é o melhor caso em tempo de execução de qualquer algoritmo de ordenação baseado em comparações.

Logo, é o caso em que apenas $n-1$ comparações são realizadas para ordenar o vetor que ocorre, por exemplo, quando o vetor já está ordenado.

3. Mostre que $lg(n!) \geq \frac{n}{4} lgn$ para $n \geq 4$ sem usar a fórmula de Stirling.

$n!$ pode ser escrito como um produtório:

$$\prod_{k=1}^{n} k$$

Também podemos escrever o produtório como um somatório, pela seguinte identidade (CLRS pp 1061 - 2ed):

$$lg\Big(\prod_{k=1}^{n} k\Big) = \sum_{k=1}^{n} lg(k)$$

Logo:

$$
\begin{aligned}
lg(n!) = \sum_{k=1}^{n} lg(k) &= \sum_{k=1}^{\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=\left\lfloor \frac{n}{4} \right\rfloor+1}^{2\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=2\left\lfloor \frac{n}{4} \right\rfloor+1}^{3\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=3\left\lfloor \frac{n}{4} \right\rfloor+1}^{n} lg(k) \\
&\geq \sum_{k=\left\lfloor \frac{n}{4} \right\rfloor+1}^{n} lg(k) \qquad \text{(descartando 1/4 da som} \\
&\geq \sum_{k=\frac{n}{4}}^{n} lg(\frac{n}{4}) \\
&\geq \frac{n}{4} lg \frac{n}{4} = \frac{n}{4}(lgn - lg2^2) = \frac{n}{4}(lgn - 2) = \frac{n}{4} lgn - \frac{1}{2}n
\end{aligned}
$$

$$\therefore lg(n!) \geq \frac{n}{4} lgn$$

CLRS (Outros)

A.1-7 Avalie o produtório $\prod_{k=1}^{n} 2(4^k)$.

$$\prod_{k=1}^{n} 2(4^k) = \prod_{k=1}^{n} 2({(2^2)}^k) = \prod_{k=1}^{n} 2(2^{2k}) = \prod_{k=1}^{n} 2^{2k+1}$$

Se avaliarmos o produtório para $n = 3$, por exemplo:

$$\prod_{k=1}^{n} 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

Percebemos que o expoente de 2 cresce em uma série aritmética:

$$\sum_{k=1}^{n} 2k + 1 = \sum_{k=1}^{n} 2k + \sum_{k=1}^{n} 1 = 2 \sum_{k=1}^{n} k + n = 2(\frac{n(n+1)}{2}) + n = n(n+2)$$

Portanto:

$$\prod_{k=1}^{n} 2(4^k) = 2^{n(n+2)}$$