

# MAC 5711 - Análise de Algoritmos

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

29 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  $2^n$ :

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

Note que  $3^n > 2^n$  e quando  $n \rightarrow \infty$ , então  $\frac{3^n}{2^n} \rightarrow \infty$ , 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  $\lg n = O(\log_{10} n)$ .  $\square$

2. Usando a definição de notação  $O$ , prove que

(a)  $n^2 + 10n + 20 = O(n^2)$

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

$$n^2 + 10n + 20 \leq cn^2, \forall n \geq n_0$$

Então  $n^2 + 10n + 20 = O(n^2)$ . Note também a seguinte relação:

$$n^2 + 10n + 20 \leq n^2 + 10n^2 + 20n^2 = 31n^2$$

Logo se fizermos  $c = 31$  e  $n_0 = 1$  teremos que:

$$n^2 + 10n + 20 \leq 31n^2, \forall n \geq 1$$

Portanto podemos concluir que  $n^2 + 10n + 20 = O(n^2)$ .  $\square$

(b)  $\lceil n/3 \rceil = O(n)$

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

$$\lceil n/3 \rceil \leq cn, \forall n \geq n_0$$

Então  $\lceil n/3 \rceil = O(n)$ . Note também a seguinte relação.

$$\lceil \frac{n}{3} \rceil \leq \frac{n}{3} + 1 \leq \frac{n}{3} + n = \frac{4n}{3}, \forall n \geq 1$$

Logo se fizermos  $c = 4/3$  e  $n_0 = 1$ , teremos:

$$\lceil n/3 \rceil \leq \frac{4n}{3}, \forall n \geq 1$$

Portanto podemos concluir que  $\lceil n/3 \rceil = O(n)$ .  $\square$

(c)  $\lg n = O(\log_{10} n)$

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 1-(b) 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  $\lg n = O(\log_{10} n)$ .  $\square$

(d)  $n = O(2^n)$

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

$$n \leq 2^n, \forall n \geq n_0$$

Então  $n = O(2^n)$ . Note trivialmente que se fizermos  $c = 1$  e  $n_0 = 1$ , teremos:

$$n \leq 2^n, \forall n \geq 1$$

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

(e)  $n/1000$  não é  $O(1)$

Vamos assumir por contradição que  $n/1000 = O(1)$ , e portanto que existem as constantes  $c > 0$  e  $n_0 > 0$  tal que:

$$n/1000 \leq c, \forall n \geq n_0$$

Note que quando  $n \rightarrow \infty$ , então  $n/1000 \rightarrow \infty$ , portanto não importa quão grande seja  $c$ , com certeza existe um  $n$  suficientemente grande tal que:

$$n/1000 > c$$

Logo podemos concluir que  $n/1000 \notin O(1)$ .  $\square$ .

(f)  $n^2/2$  não é  $O(n)$

Vamos assumir por contradição que  $n^2/2 = O(n)$ , e portanto que existem as constantes  $c > 0$  e  $n_0 > 0$  tal que:

$$n^2/2 \leq cn, \forall n \geq n_0$$

Dividindo os dois lados por  $n$  teremos:

$$\begin{aligned} \frac{n^2}{2n} &\leq \frac{cn}{n}, \forall n \geq n_0 \\ \frac{n}{2} &\leq c, \forall n \geq n_0 \end{aligned}$$

Note que quando  $n \rightarrow \infty$ , então  $\frac{n}{2} \rightarrow \infty$ , portanto não importa quão grande seja  $c$  com certeza existe um  $n$  suficientemente grande tal que:

$$n^2/2 > cn$$

Portanto podemos concluir que  $n^2/2$  não é  $O(n)$ .  $\square$

## Lista 2

1. Resolva as recorrências abaixo:

(a)  $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n^2)$

(b)  $T(n) = 8T(\lfloor n/2 \rfloor) + \Theta(n^2)$

(c)  $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n^3)$

Vamos simplificar a recorrência acima escolhendo a função  $n^3$  para representar  $\Theta(n^3)$  e vamos também assumir que  $n$  é uma potência de 2 e assim podemos remover o operador chão. Logo podemos simplificar  $T(n)$  para:

$$T(n) = 2T(n/2) + n^3$$

Vamos encontrar uma formula fechada para recorrência acima pelo método da expansão, para isso vamos assumir que  $n = 2^k$  e  $k = \lg n$ .

$$T(n) = 2T\left(\frac{n}{2}\right) + n^3$$

$$T\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^3\right] + n^3 = 2^2T\left(\frac{n}{2^2}\right) + 2\left(\frac{n}{2}\right)^3 + n^3$$

$$T\left(\frac{n}{2^2}\right) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^3\right] + 2\left(\frac{n}{2^2}\right)^3 + n^3 = 2^3T\left(\frac{n}{2^3}\right) + 2^2\left(\frac{n}{2^2}\right)^3 + 2\left(\frac{n}{2^2}\right)^3 + n^3$$

$$\begin{aligned} T\left(\frac{n}{2^3}\right) &= 2^3\left[2T\left(\frac{n}{2^4}\right) + \left(\frac{n}{2^3}\right)^3\right] + 2^2\left(\frac{n}{2^3}\right)^3 + 2\left(\frac{n}{2^3}\right)^3 + n^3 = \\ &= 2^4T\left(\frac{n}{2^4}\right) + 2^3\left(\frac{n}{2^3}\right)^3 + 2^2\left(\frac{n}{2^3}\right)^3 + 2\left(\frac{n}{2^3}\right)^3 + n^3 \end{aligned}$$

Note que podemos desenvolver os termos gerados pela função  $n^3$  da seguinte maneira:

$$\begin{aligned} 2^3\left(\frac{n}{2^3}\right)^3 + 2^2\left(\frac{n}{2^2}\right)^3 + 2\left(\frac{n}{2}\right)^3 + n^3 &= \\ 2^3\frac{n^3}{2^9} + 2^2\frac{n^3}{2^6} + 2\frac{n^3}{2^3} + n^3 &= \\ \frac{n^3}{2^6} + \frac{n^3}{2^4} + \frac{n^3}{2^2} + n^3 & \end{aligned}$$

Por essas expansão podemos supor que esse sumatório até  $k$  pode ser da seguinte forma:

$$\sum_{i=0}^{k-1} \frac{n^3}{2^i} = n^3 \sum_{i=0}^{k-1} \frac{1}{2^i}$$

Logo voltando para a recorrência podemos e considerando que  $k = \lg n$  teremos

$$\begin{aligned}
T(n) &= 2^k T(n/2^k) + n^3 \sum_{i=0}^{k-1} \frac{1}{2^{2i}} \\
T(n) &= 2^{\lg n} T(n/2^{\lg n}) + n^3 \sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} \\
T(n) &= n T(n/n) + n^3 \sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} \\
T(n) &= n T(1) + n^3 \sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} \\
T(n) &= n + n^3 \sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}}
\end{aligned}$$

Agora vamos simplificar essa formula fechada resolvendo o somatório. Note que esse somatório é uma serie geometrica de razão  $r = 1/4$ , logo:

$$\begin{aligned}
\sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} &= \frac{(1/4)^{\lg n} - 1}{1/4 - 1} \\
\sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} &= \frac{1^{\lg n} / 4^{\lg n} - 1}{3/4} \\
\sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} &= -\frac{4}{3} \left[ \frac{1}{(2^{\lg n})^2} - 1 \right] \\
\sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} &= -\frac{4}{3} \left[ \frac{1}{n^2} - 1 \right] \\
\sum_{i=0}^{\lg n - 1} \frac{1}{2^{2i}} &= \frac{4}{3} - \frac{4}{3n^2}
\end{aligned}$$

voltando a recorrência teremos:

$$\begin{aligned}
T(n) &= n + n^3 \left[ \frac{4}{3} - \frac{4}{3n^2} \right] \\
T(n) &= n + \frac{4n^3}{3} - \frac{4n}{3} \\
T(n) &= n \left[ 1 - \frac{4}{3} + \frac{4n^2}{3} \right] \\
T(n) &= n \left[ \frac{4n^2}{3} - \frac{1}{3} \right] \\
T(n) &= n \left[ \frac{4n^2 - 1}{3} \right]
\end{aligned}$$

Agora vamos provar que a formula fechada acima vale para a recorrência  $T(n)$  para isso vamos assumir que  $n = 2^k$ , portanto a formula fechada em função de  $2^k$  será:

$$T(2^k) = 2^k \left[ \frac{2^{2k+2} - 1}{3} \right]$$

$$T(2^k) = 2^k \left[ \frac{2^{2k+2} - 1}{3} \right]$$

Vamos provar por indução em  $k$

- **caso base:** Para  $k = 0$  teremos:

$$T(2^0) = T(1) = 2^0 \left[ \frac{2^{2 \cdot 0 + 2} - 1}{3} \right] = \frac{2^2 - 1}{3} = \frac{3}{3} = 1$$

- **Hipótese de indução:** Vamos assumir que a formula abaixo vale para  $k > 1$ :

$$T(2^{k-1}) = 2^{k-1} \left[ \frac{2^{2(k-1)+2} - 1}{3} \right] = 2^{k-1} \left[ \frac{2^{2k} - 1}{3} \right]$$

- **Prova:** Vamos provar que a formula fechada vale para qualquer  $k$ :

$$\begin{aligned} T(2^k) &= 2 \left[ 2^{k-1} \left[ \frac{2^{2k} - 1}{3} \right] \right] + 2^{3k} \\ &= 2^k \left[ \frac{2^{2k} - 1}{3} \right] + 2^{3k} \\ &= \frac{2^{3k}}{3} - \frac{2^k}{3} + 2^{3k} \\ &= 2^{3k} \left[ \frac{1}{3} + 1 \right] - \frac{2^k}{3} \\ &= 2^{3k} \left[ \frac{4}{3} \right] - \frac{2^k}{3} \\ &= \frac{2^{3k} 2^2}{3} - \frac{2^k}{3} \\ &= \frac{2^{3k+2} - 2^k}{3} \\ &= 2^k \left[ \frac{2^{2k+2} - 1}{3} \right] \end{aligned}$$

Logo concluímos que a formula é válida.  $\square$

Conforme demonstrado no exercício a recorrência  $T(n) = n \left[ \frac{4n^2 - 1}{3} \right]$  para  $n$  sendo uma potência de 2. Portanto podemos concluir que  $T(n) = \Theta(n^3)$ .

(d)  $T(n) = 7T(\lfloor n/3 \rfloor) + \Theta(n^2)$

(e)  $T(n) = 2T(\lfloor 9n/10 \rfloor) + \Theta(n)$

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       $k = \lfloor (p+r)/3 \rfloor$ 
3       $m = k+1 + \lfloor (p+r)/3 \rfloor$ 
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:

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

Por expansão:



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

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  $1 \leq x < n$

**Passo:** para  $n \geq 2$

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

Como queríamos demonstrar!

□

### 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( $A, p, r$ )

```
1  q = PARTITION(A, p, r)
2  QUICKSORT(A, p, q - 1)
3  QUICKSORT(A, q + 1, r)
```

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

QUICKSORT( $A, p, r$ )

```
1  while p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q - 1)
4      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( $A, p, r$ )

```

1  while  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      if  $(q - p) < (r - q)$ 
4          HALF-TAIL-QUICKSORT( $A, p, q - 1$ )
5           $p = q + 1$ 
6      else
7          HALF-TAIL-QUICKSORT( $A, q + 1, r$ )
8           $r = q - 1$ 

```

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

**Algoritmo Máximo** ( $v, n$ )

```

1.  $maior \leftarrow 0$ 
2.  $segundo\_maior \leftarrow 0$ 
3. para  $i \leftarrow 1$  até  $n$  faça
4.     se  $v[i] > maior$ 
5.         então  $segundo\_maior \leftarrow maior$ 
6.          $maior \leftarrow v[i]$ 
7.     senão se  $v[i] > segundo\_maior$ 
8.         então  $segundo\_maior \leftarrow v[i]$ 
9. devolva  $segundo\_maior$ 

```

Suponha que a entrada do algoritmo é uma permutação de 1 a  $n$  escolhida uniformemente dentre todas as permutações 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  $n \geq 2$ . 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( $X, Y, p, q, r, s$ )

```
1  if  $p == q$ 
2    // We have found the median between p, q and r
3    return  $X[p]$ 
4  elseif  $r == s$ 
5    // We have found the median between q, r and s
6    return  $Y[r]$ 
7   $i = p + (q - p)/2$ 
8   $j = r + (s - r)/2$ 
9  if  $X[i] > Y[j]$ 
10      $q = i$ 
11      $r = j$ 
12 else
13      $p = i$ 
14      $s = j$ 
15 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( $A, p, r, k$ )
1  if  $p == r$ 
2      return  $A[p]$ 
3
4   $x = \text{MEDIAN}(A, p, r)$ 
5   $q = \text{PARTITION}(x)$ 
6   $k = q - p + 1$ 
7  if  $i == k$ 
8      return  $A[q]$ 
9  elseif  $k < i$ 
10     return SELECTION( $A, p, q - 1, k$ )
11 else
12     return SELECTION( $A, p, q + 1, r, k - i$ )
13

```

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?

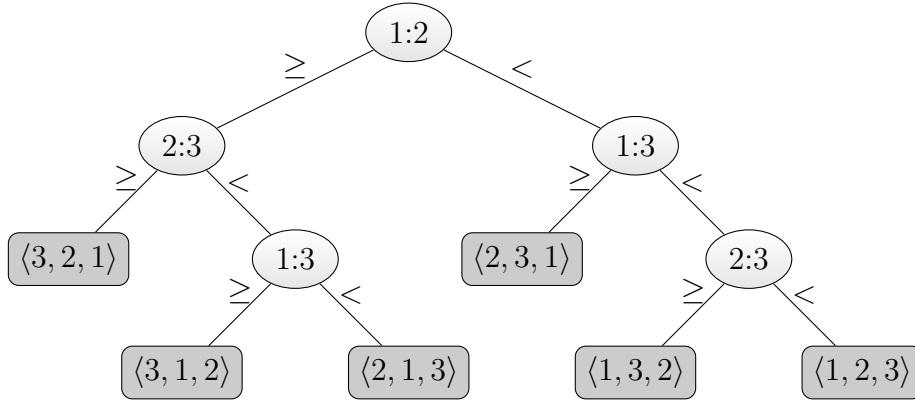
Os algoritmos estáveis são o insertionsort e o mergesort (versão do Cormen). Os demais não são estáveis.

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?

**Resposta:** 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  $\lg(n!) \geq \frac{n}{4} \lg n$  para  $n \geq 4$  sem usar a fórmula de Stirling.

**Resposta:**  $n!$  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):

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

Logo:

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

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

4. (**CLRS 8.1-3**) Mostre que não há algoritmo de ordenação baseado em comparações cujo consumo de tempo é linear para pelo menos metade de  $n!$  permutações de 1 a  $n$ . O que acontece se trocarmos "metade" por uma fração de  $1/n$ ? O que acontece se trocarmos "metade" por uma fração de  $1/2^n$ ?

**Resposta:** Assim como visto em aula uma árvore de decisão pode ser utilizada para representar o número de comparações executadas por um algoritmo. A árvore de decisão é uma árvore binária onde cada nó não folha é uma comparação e cada folha é uma permutação de entrada, o caminho da raiz da árvore até uma de suas folhas mostra a quantidade de comparações executadas para aquela permutação, portanto a distância da raiz para a folha mais distante (altura da árvore) reflete o tempo gasto pelo pior caso do algoritmo.

Vamos verificar qual a altura da árvore de decisão para saber qual o tempo gasto no pior caso para pelo menos metade das possíveis permutações  $n!/2$ . Seja  $l$  o número de folhas do algoritmo e  $h$  a altura da árvore, como a árvore de decisão é uma árvore binária sabemos que ela terá no máximo  $2^h$  folhas. Portanto temos a seguinte relação:

$$\frac{n!}{2} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonicamente crescente temos que:

$$\begin{aligned}
lg 2^h &\geq lg \frac{n!}{2} \\
h &\geq lg n! - lg 2 \\
h &\geq lg n! - 1
\end{aligned}$$

Como visto em aula  $lg n! \geq \frac{1}{2}n lg n$ , dessa inequação chegamos que  $h \geq lg n! \geq \frac{1}{2}n lg n$ . Disso podemos concluir que  $h \geq \frac{1}{2}n lg n - 1$ , ou seja, mesmo utilizando apenas a metade das



permutações assintoticamente o algoritmo ótimo gasta  $\Omega(n \lg n)$  no pior caso.

Trocando a "metade" por uma fração  $1/n$ , podemos utilizar o mesmo raciocínio, definindo  $l$  como a quantidade de folhas da árvore de decisão e  $h$  a altura da árvore, teremos a seguinte inequação:

$$\frac{n!}{n} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonicamente crescente temos que:

$$\begin{aligned} \lg 2^h &\geq \lg \frac{n!}{n} \\ h &\geq \lg n! - \lg n \\ h &\geq n \lg n - \lg n \end{aligned}$$

Note  $n \lg n - \lg n = \Theta(n \lg n)$ , portanto podemos concluir que para uma fração de  $1/n$  das permutações também mantem-se gasto de tempo de  $\Omega(n \lg n)$  no pior caso.

Trocando a "metade" pela fração de  $1/2^n$ , podemos utilizar o mesmo raciocínio para calcular o pior caso de qualquer algoritmo de ordenação baseada em comparações. Seja  $l$  o número de folhas e  $h$  a altura da árvore teremos a seguinte relação:

$$\frac{n!}{2^n} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonicamente crescente temos que:

$$\begin{aligned} \lg 2^h &\geq \lg \left( \frac{n!}{2^n} \right) \\ h &\geq \lg(n!) - \lg 2^n \\ h &\geq n \lg n - n \end{aligned}$$

Note que  $n \lg n - n = \Theta(n \lg n)$ , portanto da relação acima temos que mesmo para a fração de  $1/2^n$  de permutações os algoritmos de ordenação que utilização comparações consomem tempo  $\Omega(n \lg n)$  no pior caso.

5. (CLRS 8.2-1) Simule a execução do COUNTINGSORT usando como entrada o vetor:

$$A[1 \cdots 11] = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$$

**Resposta:** A configuração inicial do algoritmo após o laço inicial será:

A =            k = 6											C =						
6	0	2	0	1	3	4	6	1	3	2	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6

O vetor  $A$  nunca é alterado, o vetor  $C$  é representado abaixo após o final do segundo laço (esquerda) e após o final do terceiro laço (direita):

C =							C =						
2	2	2	2	1	0	2	2	4	6	8	9	9	11
0	1	2	3	4	5	6	0	1	2	3	4	5	6

O último laço começa pelo com o índice  $j = n$  e vai até 1, o vetor  $B$  começa vazio e vai sendo preenchido até o final do algoritmo. Abaixo segue os estados dos vetores  $C$  e  $B$  no fim das iterações  $j = 8, j = 4$  e  $j = 1$ :

B =            para j = 11 até 8											C =						
			1		2		3			6	2	3	5	7	9	9	10
1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6
B =            para j = 7 até 4											C =						
	0	1	1		2	3	3	4		6	1	2	5	6	8	9	10
1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6
B =            para j = 3 até 1											C =						
0	0	1	1	2	2	3	3	4	6	6	0	2	4	6	8	9	9
1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6

Então o algoritmo se encerra mantendo o vetor  $B$  ordenado.

6. (CLRS 8.2-2) Mostre que o COUNTING-SORT é estável.

**Resposta:** Vimos em sala que um algoritmo de ordenação é estável se sempre que, inicialmente,  $A[i] = A[j]$  para  $i < j$ , a cópia  $A[i]$  termina em uma posição menor do vetor que a cópia  $A[j]$ .

O COUNTING-SORT conta quantas vezes um inteiro de  $1..k$  aparece em  $A[1..n]$  e armazena essa informação no vetor  $C$ . A próxima iteração faz com que o vetor  $C$  tenha a contagem acumulada dos elementos em  $A$ . Logo, após essa iteração nós sabemos que  $C[i] < C[i + 1]$ .

A última iteração ordena efetivamente  $A$ , copiando seus elementos de  $n..1$  para um vetor auxiliar  $B$ . Como a contagem foi feita na ordem  $1..n$  e os valores de  $C$  são usados como índices em  $B$  e eles são decrementados a cada vez que uma cópia é feita, a ordem relativa em que os elementos aparecem em  $A$  é preservada, mesmo no caso onde há repetições, garantindo, assim, a estabilidade do algoritmo.

8. (**CLRS 8.2-4**) Descreva um algoritmo que, dados  $n$  inteiros no intervalo de 1 a  $k$ , preprocesse sua entrada e então responda em  $O(1)$  qualquer consulta sobre quantos dos  $n$  inteiros dados caem em um intervalo  $[a \cdots b]$ . O preprocessamento efetuado pelo seu algoritmo deve consumir tempo  $O(n + k)$ .

**Resposta:** O algoritmo desenvolvido é baseado no algoritmo de *Counting Sort* descrito no livro do Cormen. O algoritmo possui duas rotinas, uma de preprocessamento que devolve um vetor de contagem  $C$  e um de consulta que retorna a quantidade de elementos conforme o enunciado. O algoritmo de preprocessamento abaixo recebe um vetor  $A$  de  $n$  inteiros qualquer tal que seus elementos estão dentro do intervalo  $[1 \cdots k]$  e realiza o preprocessamento devolvendo o vetor " $C$ " do *counting sort* antes de ordena-lo no vetor  $B$ . Esse vetor possui o índice do último valor no vetor ordenado de cada índice, por exemplo,  $C[x] = y$ , indica que o último elemento  $x$  está no índice  $y$  do vetor ordenado.

PREPROCESSAMENTO( $A, k, n$ )

```

1  for  $i = 1$  to  $k$ 
2       $C[i] = 0$ 
3  for  $i = 1$  to  $n$ 
4       $C[A[i]] = C[A[i]] + 1$ 
5  for  $i = 2$  to  $k$ 
6       $C[i] = C[i] + C[i - 1]$ 
7  return  $C$ 
```

O procedimento CONTANOINTERVALO recebe o vetor gerado pelo PREPROCESSAMENTO e os extremos do intervalo  $[a \cdots b]$  e devolve a quantidade de elementos do vetor original  $A$  que estão dentro desse vetor.

CONTANOINTERVALO( $C, a, b$ )

```

1  if  $a \leq 1$ 
2       $a' = 1$ 
3  else  $a' = C[a - 1] + 1$ 
4  return  $C[b] - a' + 1$ 
```

**Corretude**

O procedimento de preprocessamento utiliza o *counting sort* disso sabemos que o vetor  $C[i]$

contém a quantidade de elementos menores ou iguais a  $i$  do vetor original, isso implica que o valor de  $C[i]$  é a posição do último elemento  $i$  (não o elemento da posição  $i$ ) do vetor original no vetor ordenado, isso implica também que o valor de  $C[i - 1] + 1$  é a posição do primeiro elemento  $i$  do vetor original  $A$  no vetor ordenado. Disso temos que  $C[b] - C[a - 1] + 1$  é quantidade de elementos com os valores entre  $[a \cdots b]$  do vetor original. Note que o *if* do procedimento CONTANOINTERVALO, apenas garante que  $a - 1 \geq 1$  para que não seja acessado uma posição fora do vetor  $C$ , caso isso ocorra consideramos que a posição inicial é 1, note que isso é válido para quando existem valores 1 em  $A$  e quando não existem.

### Desempenho

Veja a tabela de gastos em cada linha do procedimento PREPROCESSAMENTO:

Linha	Tempo
1	$\Theta(k)$
2	$\Theta(k)$
3	$\Theta(n)$
4	$\Theta(n)$
5	$\Theta(k)$
6	$\Theta(k)$
7	$\Theta(1)$
Total	$\Theta(n + k)$

O procedimento gasta o tempo total  $\Theta(n+k)$ . O tempo gasto no procedimento CONTANOINTERVALO está expresso na tabela abaixo:

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$\Theta(1)$
4	$\Theta(1)$
Total	$\Theta(1)$

Portanto o algoritmo gasta tempo  $\Theta(n + k)$  para o préprocessamento e tempo  $\Theta(1)$  para a chamada a contagem dos elementos conforme enunciando do exercício.

## Lista 6

1. (CLRS 15.2-1) Determine a parentização ótima de um produto de cadeia de matrizes cuja sequência de dimensões é  $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$ .

**Resposta:** Como a sequência tem 7 elementos, o valor de  $n = 6$ . A figura 1 mostra cada iteração da entrada para geração da matriz  $m$  que contém o número mínimo de multiplicações escalares necessário para calcular o produto  $A_1..A_n$ .

l	i	n-l+1	j	k	q
2	1	5	2	1	150
	2	5	3	2	360
	3	5	4	3	180
	4	5	5	4	3000
	5	5	6	5	1500
3	1	4	3	1	960
	1	4	3	2	330
	2	4	4	2	330
	2	4	4	3	960
	3	4	5	3	4800
	3	4	5	4	930
	4	4	6	4	1860
4	4	4	6	5	6600
	1	3	4	1	580
	1	3	4	2	405
	1	3	4	3	630
	2	3	5	2	2430
	2	3	5	3	9360
	2	3	5	4	2830
	3	3	6	3	2076
5	3	3	6	4	1770
	3	3	6	5	1830
	1	2	5	1	4930
	1	2	5	2	1830
	1	2	5	3	6330
	1	2	5	4	1655
	2	2	6	2	1950
	2	2	6	3	2940
6	2	2	6	4	2130
	2	2	6	5	5430
	1	1	6	1	2250
	1	1	6	2	2010
	1	1	6	3	2550
	1	1	6	4	2055
	1	1	6	4	3155

Figura 1: Cálculo do número mínimo de multiplicações escalares

Os valores destacados em azul são aqueles menores para uma dada iteração  $i, j$  e, conseqüentemente, são armazenados na matriz  $m[i, j]$ , como pode ser visto na figura 2.

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

Figura 2: Matriz  $m$  preenchida após todas as iterações

A figura 3 mostra uma segunda matriz  $s$  utilizada para armazenar o valor de  $k$ , de forma que possamos rastrear a sequência ótima para multiplicação posteriormente.

	1	2	3	4	5	6
1	0	1	2	2	4	2
2		0	2	2	2	2
3			0	3	4	4
4				0	4	4
5					0	5
6						0

Figura 3: Cálculo do número mínimo de multiplicações escalares

Sendo assim, a sequência ótima é  $((A_1 A_2)((A_3 A_4)(A_5 A_6)))$ .

2. **(15.2-2)** Adapte o algoritmo dado em aula para que calcule uma matriz  $s$  que pode ser usada para determinar uma ordem ótima de multiplicação das matrizes. Dê um algoritmo recursivo que recebe as matrizes  $A_1, \dots, A_n$  em uma matriz tridimensional  $A$ , recebe a matriz  $s$  e índices  $i$  e  $j$ , com  $i \leq j$ , e faz o produto das matrizes  $A_i \dots A_j$  usando a ordem ótima dada em  $s$ .

**Resposta:** Uma modificação do algoritmo dado no CLRS p. 375 já realiza o procedimento com a matriz  $s$ , sendo o parâmetro  $p$  a sequência com as dimensões das matrizes e  $n$  o tamanho de  $p - 1$ .

MATRIX-CHAIN-ORDER( $p, n$ )

```

1   $m[1..n, 1..n]$  e  $s[1..n, 1..n]$  são novas tabelas
2  for  $i = 1$  to  $n$ 
3       $m[i, i] = 0$ 
4  for  $l = 2$  to  $n$  //  $l$  é o tamanho da cadeia
5      for  $i = 1$  to  $n - l + 1$ 
6           $j = i + l - 1$ 
7           $m[i, j] = \infty$ 
8          for  $k = i$  to  $j - 1$ 
9               $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
10             if  $q < m[i, j]$ 
11                  $m[i, j] = q$ 
12                  $s[i, j] = k$ 
13 return  $m, s$ 

```

Agora, basta calcular o resultado das multiplicações das matrizes na sequência ótima dada por  $s$ . Quando  $i = j$ , estamos na base da recursão, ou seja, no ponto em que uma das matrizes de  $A$  deve ser retornada para realização da multiplicação. Caso contrário, efetuamos o produto dos pares de matrizes e do resultado do produto dos pares de matrizes que, como é realizado *bottom-up*, termina com uma única matriz quando todas as chamadas recursivas forem concluídas.

MATRIX-CHAIN-MULTIPLY( $A, s, i, j$ )

```

1  if  $i == j$ 
2      return  $A[i]$ 
3  else  $a = \text{MATRIX-CHAIN-MULTIPLY}(A, s, i, s[i, j])$ 
4       $b = \text{MATRIX-CHAIN-MULTIPLY}(A, s, s[i, j] + 1, j)$ 
5      return  $a * b$ 

```

4. (15.4-5) Escreva um algoritmo que, dado  $n$  e um vetor  $v[1..n]$  de inteiros, determina uma subsequência crescente mais longa de  $v$ . Seu algoritmo deve consumir tempo  $O(n^2)$ . Se puder, dê um algoritmo para este problema que consome tempo  $O(n \lg n)$ .

**Resposta:** Para este exercício, basta utilizarmos os algoritmos LCS-LENGTH e PRINT-LCS disponíveis no CLRS 15.4, sendo que o parâmetro  $u$  é uma cópia do vetor  $v$  ordenado por algum algoritmo de ordenação que, neste caso, usamos o QUICKSORT.

BUILD-LONGEST-INCREASING-SUBSEQUENCE( $v, n$ )

```

1   $u[1..n]$  vetor auxiliar utilizado para copiar e ordenar  $v$ 
2  for  $i = 1$  to  $n$ 
3       $u[i] = v[i]$ 
4  QUICKSORT( $u, 1, u.length$ )
5   $c, b = \text{LCS-LENGTH}(v, u)$ 
6  PRINT-LCS( $b, v, v.length, u.length$ )

```

Como nós queremos encontrar a maior subsequência crescente de  $v$ , e  $u$  é a cópia ordenada de  $v$  em ordem crescente, o LCS-LENGTH deverá, então, encontrá-la dentre todas as subsequências crescentes dadas por  $u$  no vetor original  $v$ .

A tabela 1 mostra o tempo de execução do algoritmo BUILD-LONGEST-INCREASING-SUBSEQUENCE.

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(n)$
3	$\Theta(n)$
4	$\Theta(n \log n)$
5	$O(n * n)$
6	$O(n + n)$
Total	$O(n^2 + 2n) + \Theta(n \log n) + \Theta(2n) = O(n^2)$

Tabela 1: Tempo de execução

Originalmente, o LCS-LENGTH tem tempo de execução em função de  $m$  e  $n$  ( $O(mn)$ ), porém, como sabemos que  $v$  e  $u$  têm o mesmo tamanho ( $n$ ), teremos um tempo de execução  $O(n^2)$ . A mesma analogia vale para o algoritmo PRINT-LCS que é executado na linha 6.



## 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:

$$\prod_{k=1}^3 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

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

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

Portanto:

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