MAC 5711 - Análise de Algoritmos

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

13 de outubro 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 $\lg n = O(\log_{10} n)$. $\square$

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 $n \to \infty$, então $n/1000 \to \infty$, 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:

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

Note que quando $n \to \infty$, então $\frac{n}{2} \to \infty$, 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 pontência 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 metódo da expansão, para isso vamos assumir que $n = 2^k$ e $k = \lg n$.

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

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:

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

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

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

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

$$T(n) = nT(n/n) + n^3 \sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}}$$

$$T(n) = nT(1) + n^3 \sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}}$$

$$T(n) = n + n^3 \sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}}$$

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

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}} = \frac{(1/4)^{\lg n} - 1}{1/4 - 1}$$

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}} = \frac{1^{\lg n}/4^{\lg n} - 1}{3/4}$$

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

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}} = -\frac{4}{3}\left[\frac{1}{n^2} - 1\right]$$

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^{2i}} = \frac{4}{3} - \frac{4}{3n^2}$$

voltando a recorrência teremos:

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

$$T(n) = n + \frac{4n^3}{3} - \frac{4n}{3}$$

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

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

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

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:

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

Vamos provar por indução em k

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

$$T(2^0) = T(1) = 2^0 \left[\frac{2^{2.0+2} - 1}{3}\right] = \frac{2^2 - 1}{3} = \frac{3}{3} = 1$$

- **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. □

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 concluír 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 = ⌊(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!

□

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 = 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 $n >= 2$ 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)

```
 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 = MEDIAN(A, p, r)
 5  q = 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?

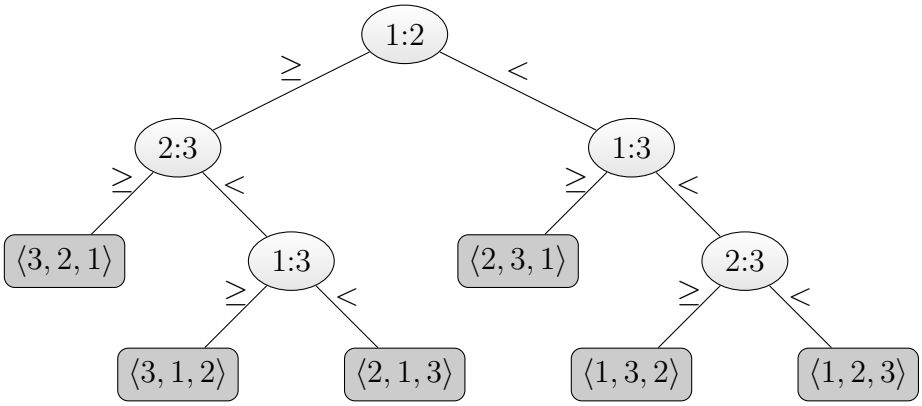
Os algoritmos estáveis são o insertionsort e o mergesort (versão do Cormen). Os demais não são estáveis.

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

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

Logo:

$$lg(n!) = \sum_{k=1}^{n} lg(k) = \sum_{k=1}^{\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=\left\lfloor \frac{n}{4} \right\rfloor+1}^{2\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=2\left\lfloor \frac{n}{4} \right\rfloor+1}^{3\left\lfloor \frac{n}{4} \right\rfloor} lg(k) + \sum_{k=3\left\lfloor \frac{n}{4} \right\rfloor+1}^{n} lg(k)$$

$$\geq \sum_{k=\left\lfloor \frac{n}{4} \right\rfloor+1}^{n} lg(k) \qquad \text{(descartando 1/4 da som}$$

$$\geq \sum_{k=\frac{n}{4}}^{n} lg(\frac{n}{4})$$

$$\geq \frac{n}{4} lg \frac{n}{4} = \frac{n}{4}(lgn - lg2^2) = \frac{n}{4}(lgn - 2) = \frac{n}{4}lgn - \frac{1}{2}n$$

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

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 distânte (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:

$$\lg 2^h \geq \lg \frac{n!}{2}$$
$$h \geq \lg n! - \lg 2$$
$$h \geq \lg n! - 1$$

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:

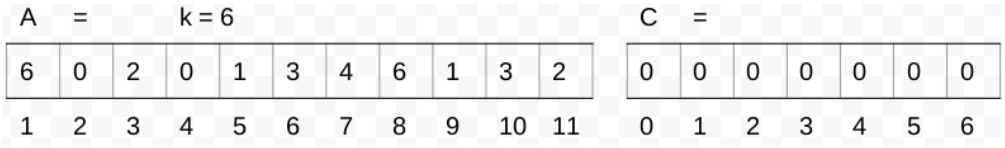
$$\begin{aligned} \lg 2^h &\geq \lg(\frac{n!}{2^n}) \\ h &\geq \lg(n!) - \lg 2^n \\ h &\geq n \lg n - n \end{aligned}$$

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.

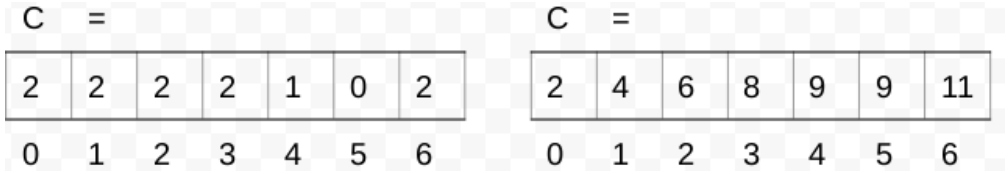
5. (**CLRS 8.2-1**) Simule a execução do CountingSort usando como entrada o vetor:

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

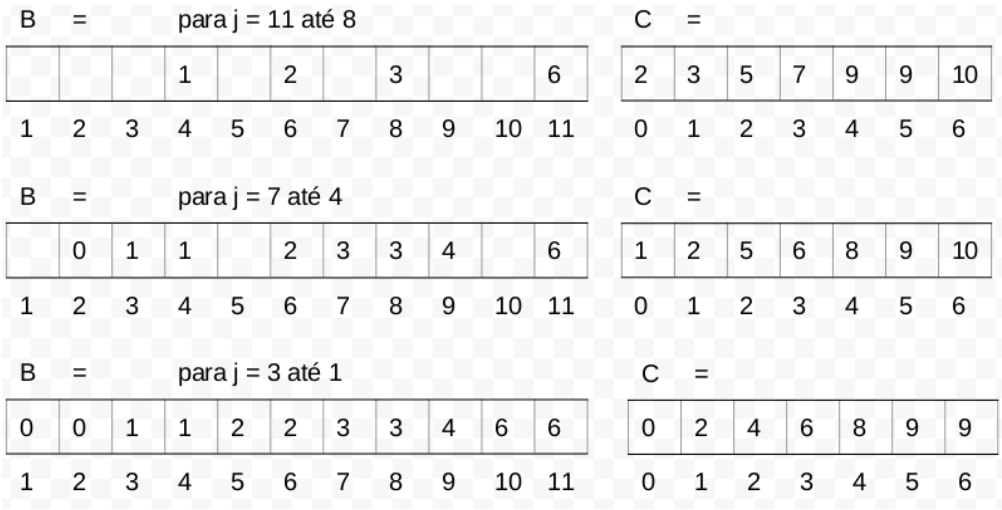
Resposta: A configuração inicial do algoritmo após o laço inicial será:



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):



O último laço começa pelo com o índice $j = n$ e vai até 1, o vetor B começa vázio 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 $j = 1$:



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.

7. **(CLRS 8.2-3)** Suponha que o **para** da linha 7 do COUNTINGSORT é substituído por

```
1  for i = 1 to n
```

Mostre que o COUNTINGSORT ainda funciona. O algoritmo resultante continua estável?

Resposta: O COUNTINGSORT continua ordenando o vetor de entrada, o laço substituído ficará:

```
1  for j = 1 to n
2        B[C[A[j]]] = A[i]
3        C[A[i]] = C[A[i]] − 1
```

Sabemos que o algoritmo não foi modificado antes da linha 7 (do algoritmo original), então o vetor C continua informando, qual a última posição que os elementos de A devem ser inseridos, ou seja, $C[i]$ indica onde $A[i]$ deve ser inserido no vetor ordenado B, após inseri-lo ele $C[i]$ é decrementando e indicando que o proximo elemento de A que seja igual a $A[i]$ deve ser colocado em uma posição anterior assim como o algoritmo dado em aula faz, logo o algoritmo continua ordenado o vetor A. Mas com essa alteração o algoritmo perde informação satelite, ou seja, perde estabilidade, por que ao encontrar um elemento $A[i]$ o mesmo será inserido em $C[A[i]]$ que indica a última posição daquele elemento, ao encontrar $A[j] = A[i]$ para $j > i$, o algoritmo ira inseri-lo em $C[A[i]] - 1$, perdendo assim a estabilidade já que $A[i]$ estará em uma posição maior que $A[i]$ estará em uma posição maior que $A[j]$ para $A[i] = A[j]$ e $i < j$.

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 consomir tempo $O(n + k)$.

Resposta: O algoritmo desenvolvido é baseado no algoritmo de *Counting Sort* descrito no livro do Cormen. O algoritmo possuí 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 possuí 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 ≤ 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:

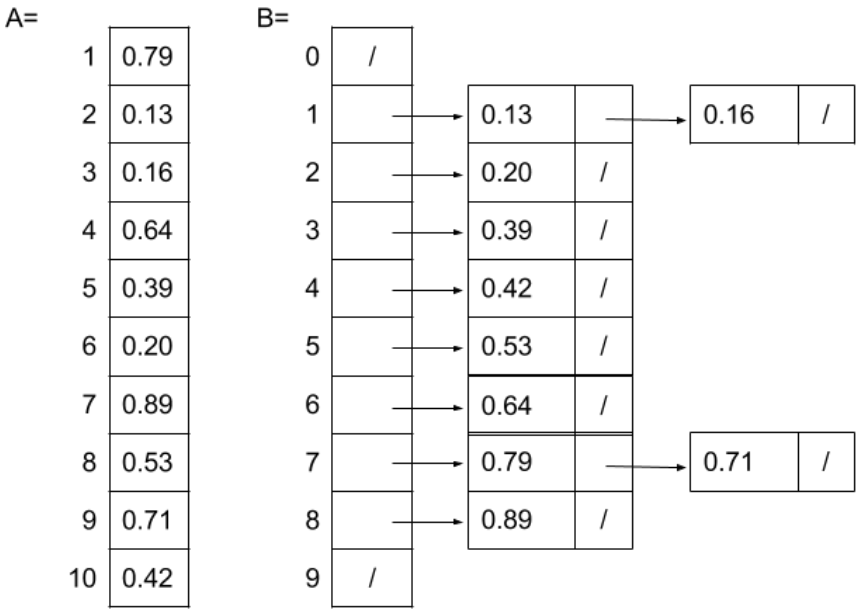
Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$\Theta(1)$
4	$\Theta(1)$
Total	$\Theta(1)$

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

10. Simule a execução do BucketSort com o vetor.

$$A[1..10] = \langle 0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42 \rangle$$

Resposta:



Então cada lista de B é ordenada e concatenada em ordem para gerar o vetor ordenado.

11. **(CLRS 8.4-2**: Qual é o consumo de tempo do pior caso para o BucketSort? Que simples ajuste do algoritmo melhora o seu pior caso para $O(n \lg n)$ e mantem o seu consumo esperado de tempo linear.

Resposta: A resposta do exercício é baseada no algoritmo do livro do CLRS, conforme descrito abaixo:

```
BucketSort(A)
1  Let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], · · · , B[n − 1] together in order
```

O pior caso do BucketSort acontence quando a entrada não segue uma distribuição uniforme e todos os itens caem em um único *Bucket*. Como o algoritmo apresentado no CLRS utiliza *insertion sort* para ordenar cada *Bucket* ele gastará tempo $O(n^2)$ no pior caso. Podemos trocar o *insertion sort* pelo Mergesort ou Heapsort que gastam $O(n \lg n)$ no pior caso. Na prática o *insertion sort* é a melhor opção por que embora gaste tempo $O(n^2)$ *insertion sort* é mais rápido que o Mergesort ou o Heapsort para vetores pequenos, note que para que o BucketSort seja uma boa opção cada *Bucket* tem que ter um número pequeno de elementos.

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	6	5	6600
4	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
	3	3	6	4	1770
	3	3	6	5	1830
5	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
	2	2	6	4	2130
	2	2	6	5	5430
6	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, consequentemente, 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 $((A_1A_2)((A_3A_4)(A_5A_6)))$.

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 $A_1, \ldots, A_n$ em uma matriz tridimensional A, recebe a matriz s e índices i e j, com $i \leq j$, e faz o produto das matrizes $A_i \ldots A_j$ 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] = ∞
 8            for k = i to j − 1
 9                 q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_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 = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
4       b = 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 $O(nlgn)$.

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 = 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	$\Theta(nlogn)$
5	$O(n * n)$
6	$O(n + n)$
Total	$O(n^2 + 2n) + \Theta(nlogn) + \Theta(2n) = O(n^2)$

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.

5. Quantas árvores de busca binária existem que guardam 6 diferentes chaves?

Resposta: Vimos em sala que podemos representar 3 nós em 5 árvores de busca binária distintas.

Vamos definir por $t(n)$ o número de diferentes árvores de busca binária para n nós. Sendo assim, o número de diferentes árvores de busca binária que podemos obter deve ter uma raiz, uma subárvore à esquerda com i nós e outra à direita com $n - 1 - i$ nós para cada i, o que nos dá:

$$t(n) = t(0)t(n-1) + t(1)t(n-2) + ... + t(n-1)t(0)$$

Logo, podemos estabelecer $t(n)$ como uma recorrência:

$$t(n) = \begin{cases} 1, & n = 0 \text{ ou } n = 1 \\ \sum_{i=1}^{n} t(i-1)t(n-i), & n > 1 \end{cases}$$

A base da recorrência nos diz que há apenas uma árvore de busca binária com nenhum ou um nó.

Sendo assim, para 6 nós, temos 132 árvores de busca binária:

$$
\begin{aligned}
&t(0) = 1\\
&t(1) = t(0)t(0) = 1\\
&t(2) = t(0)t(1) + t(1)t(0) = 2\\
&t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 5\\
&t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 14\\
&t(5) = t(0)t(4) + t(1)t(3) + t(2)t(2) + t(3)t(1) + t(4)t(0) = 42\\
&t(6) = t(0)t(5) + t(1)t(4) + t(2)t(3) + t(3)t(2) + t(4)t(1) + t(5)t(0) = 132
\end{aligned}
$$

O número de árvores de busca binária também pode ser calculado como o n-*ésimo* número de catalão:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!} = \frac{(2\times 6)!}{6!(6+1)!} = \frac{12\times 11\times 10\times 9\times 8\times 7!}{6!7!} = 132$$

8. Pode-se generalizar o problema da árvore de busca binária ótima para que inclua na sua descrição uma estimativa do número de buscas mal-sucedidas. Mais concretamente, nessa variante do problema, seriam dados não apenas os números de acessos a_i de cada chave i, mas também números b_i, para $i = 0, ..., n$, representando uma estimativa do número de buscas mal-sucedidas a elementos entre i e $i + 1$ (considere 0 como $-\infty$ e $n - 1$ como $+\infty$ para que b_0 e b_n sejam o que se espera). Encontre a recorrência para o custo de uma árvore de busca binária ótima para esta variante do problema. Escreva o algoritmo de programação dinâmica gerado a partir dessa recorrência. Quanto tempo consome o seu algoritmo?

Resposta: Para generalizarmos o problema da ABB ótima, tal que possamos incluir estimativas de acesso mal sucedidas, vamos pensar em uma ABB que armazena as chaves $k = \langle k_1, k_2, ..., k_n\rangle$ ordenadas e, para cada chave, tenhamos a estimativa do número de acessos a cada elemento de k dado por a_i, para todo $i = 1, 2, ..., n$. Essa representação é a que nós já vimos em sala de aula e pode ser vista na figura 4.

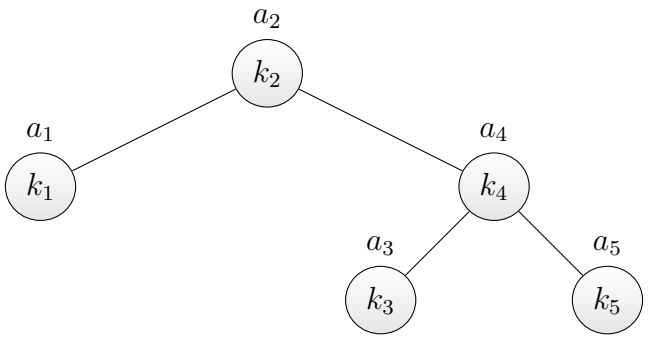


Figura 4: ABB com número de acessos a cada chave

Como temos buscas que são mal sucedidas, ou seja, valores que não estão em k, nós devemos estender a ABB de forma a incluir $n+1$ chaves que representam todas essas buscas que, neste caso, denominamos por $q_0, q_1, ..., q_n$. A figura 5 mostra essa nova representação. Note que, cada nó terminal k_i na estrutura anterior agora recebe novos nós (folhas) que representam as buscas mal sucedidas entre k_i e k_{i+1}. Vale ressaltar que, q_0 representa todas os valores menores que k_1 e q_n representa todos os valores maiores que k_n.

Da mesma forma que nas buscas bem sucedidas, temos a estimativa do número de acessos associado a cada chave q_i não encontrada em k que é dado por $b_0, b_1, ..., b_n$. Por exemplo, se tivéssemos $k = \langle 3, 5, 7, 11, 12 \rangle$, se em uma dada ABB $k_3 = 7$ e $k_4 = 11$, o nó terminal q_3 representaria os valores não encontrados $\{8, 9, 10\}$.

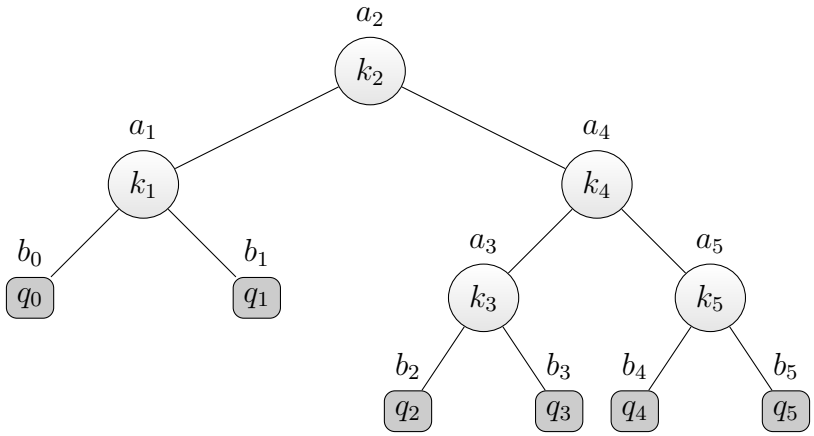


Figura 5: ABB estendida adicionando número de buscas mal-sucedidas

Visto isso, podemos definir a **subestrutura ótima** do problema: se uma ABB é ótima, então as subárvores esquerda e direita também são ótimas. Vale ressaltar que, quando escolhemos uma das chaves, digamos k_r $(i \leq r \leq j)$, como a raiz de uma subárvore $k_i, ..., k_j$, a subárvore à esquerda de k_r deve conter as chaves $k_i, ..., k_{r-1}$, bem como as representantes de buscas mal sucedidas $q_{i-1}, ..., q_{r-1}$ e a subárvore à direita de k_r deve conter as chaves $k_{r+1}, ..., k_j$ e $q_r, ..., q_j$.

Agora, devemos definir uma solução recursiva para encontrar uma ABB ótima para as chaves $k_1, k_2, ..., k_n$ cujo número esperado de comparações para uma busca bem sucedida $a_1, a_2, ..., a_n$ e o número esperado de comparações para uma busca mal sucedida $b_0, b_1, b_2, ..., b_n$ seja mínimo.

Seja $e[i, j]$ o custo esperado para uma busca em uma ABB ótima com as chaves $i, i+1, ..., j$. Vale a seguinte recorrência para $e[i, j]$:

$$e[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ \min\limits_{i \leq r \leq j} e[i, r-1] + e[r+1, j] + w[i,j], & \text{se } i \leq j \end{cases}$$

Onde:

$$w[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ w[i, j-1] + a_j + b_j, & \text{se } i \leq j \end{cases}$$

Ou seja, $w[i, j]$ é a soma das estimativas do número de buscas bem e mal sucedidas para um dado i, j:

$$w[i, j] = \sum_{l=i}^{j} a_l + \sum_{l=i-1}^{j} b_l$$

Agora, basta efetuarmos as alterações necessárias no algoritmo dado em sala de aula para calcularmos as matrizes w e e. Também vamos armazenar o valor de r em uma matriz *root*, ou seja, qual a raiz foi escolhida para uma determinada subárvore $k_i, ..., k_j$. O algoritmo EXTENDED-OPTIMAL-BST mostra o resultado.

EXTENDED-OPTIMAL-BST(a, b, n)

```
 1  for i = 1 to n + 1
 2      e[i, i − 1] = b_{i−1}
 3      w[i, i − 1] = b_{i−1}
 4  for l = 1 to n
 5      for i = 1 to n − l + 1
 6          j = i + l − 1
 7          w[i, j] = w[i, j − 1] + a_j + b_j
 8          e[i, j] = ∞
 9          for r = i to j
10              aux = e[i, r − 1] + e[r + 1, j] + w[i, j]
11              if aux < e[i, j]
12                  e[i, j] = aux
13                  root[i, j] = r
14  return e, root
```

O custo mínimo da ABB ótima é retornado na posição $e[1, n]$. A tabela 2 mostra o tempo de execução do algoritmo EXTENDED-OPTIMAL-BST. As linhas 4-13 têm 3 *loops* encadeados, que tomam no máximo n iterações cada um, o que nos dá um consumo total $\Theta(n^3)$.

Linha	Tempo
1-3	$\Theta(n)$
4-13	$\Theta(n^3)$
14	$\Theta(1)$
Total	$\Theta(n) + \Theta(n^3) = \Theta(n^3)$

Tabela 2: Tempo de execução

12. Escreva uma função que recebe como parâmetros um inteiro $n > 0$ e um vetor que armazena uma sequência de n inteiros e devolve o comprimento de uma subsequência crescente da sequência dada de soma máxima. Sua função deve consumir tempo $O(n^2)$. Modifique a sua função (sem piorar a sua complexidade assintótica) para que ela devolva também uma subsequência crescente da sequência dada de soma máxima.

Resposta: Assim como no exercício 4, podemos usar uma variação do LCS-Length para resolver este problema.

Vamos, então, definir a **subestrutura ótima** do problema. Seja $X[1..n]$ um vetor com n inteiros e $Y[1..n]$ uma cópia do vetor X ordenado, temos que $Z[1..k]$ é uma subsequência crescente de X de soma máxima, se:

- $X[n] = Y[n]$, então $Z[k] = X[n] = Y[n]$ e $Z[1..k-1]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n-1]$
- $X[n] \neq Y[n]$, então $Z[k] \neq X[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n]$
- $X[n] \neq Y[n]$, então $Z[k] \neq Y[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n]$ e $Y[1..n-1]$

Logo, percebemos que a subestrutura ótima do problema original não muda. Vamos definir por $c[i, j]$ a soma dos elementos de uma subsequência crescente de soma máxima de $X[1..i]$ e $Y[1..j]$. Vejamos, então, a **recorrência** para calcular o comprimento da subsequência crescente de soma máxima:

$$c[i,j] = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + X[i], & \text{se } i, j > 0 \text{ e } X[i] = Y[j] \\ \max(c[i, j-1], c[i-1, j]), & \text{se } i, j > 0 \text{ e } X[i] \neq Y[j] \end{cases}$$

A diferença para a recorrência original se dá no caso 2, onde o tamanho da LCS era contabilizada. Neste caso, somamos cada elemento de X[i] em uma dada subsequência $X[i..j]$.

O algoritmo Build-Max-Sum-LCS efetua, então, o cálculo desejado a partir de um vetor $X[1..n]$. Inicialmente, uma cópia de X é criada e ordenada com um algoritmo de ordenação que, neste caso, utilizamos o Quicksort. Posteriormente, chamamos a função LCS-Max-Sum-Length que calcula e retorna a matriz c com a soma e a matriz b para que possamos rastrear a subsequência crescente, respectivamente.

Assim que a matriz b é calculada, ela é submetida à rotina Get-LCS-Max-Sum que retorna, então, o vetor Z com a subsequência crescente de soma máxima, bem como o tamanho da subsequência.

```
Build-Max-Sum-LCS(X, n)
1  Y[1..n] vetor auxiliar utilizado para copiar e ordenar X
2  for i = 1 to n
3        Y[i] = X[i]
4  Quicksort(Y, 1, Y.length)
5  c, b = LCS-Max-Sum-Length(X, Y)
6  Z, l = Get-LCS-Max-Sum(X, X.length, b)
7  return Z, l
```

LCS-MAX-SUM-LENGTH(X, Y)

```
 1  n = X.length
 2  for i = 0 to n
 3       c[i, 0] = 0
 4  for j = 1 to n
 5       c[0, j] = 0
 6  for i = 1 to n
 7       for j = 1 to n
 8            if X[i] == Y[j]
 9                 c[i, j] = c[i − 1, j − 1] + X[i]
10                 b[i, j] = " ↖ "
11            elseif c[i − 1, j] ≥ c[i, j − 1]
12                 c[i, j] = c[i − 1, j]
13                 b[i, j] = " ↑ "
14            else c[i, j] = c[i, j − 1]
15                 b[i, j] = " ← "
16  return c, b
```

A última linha do algoritmo GET-LCS-MAX-SUM nos devolve a partição de Z que foi preenchida e o tamanho do vetor resultante, ou seja, $n - k$.

GET-LCS-MAX-SUM(X, n, b)

```
 1  k = n
 2  i = n
 3  j = n
 4  while i > 0 and j > 0
 5       if b[i, j] == " ↖ "
 6            Z[k] = X[i]
 7            k = k − 1
 8            i = i − 1
 9            j = j − 1
10       elseif b[i, j] == " ↑ "
11            i = i − 1
12       else j = j − 1
13  return Z[k..n], n-k
```

Sabemos que o tempo de execução do QUICKSORT, LCS-LENGTH e GET-LCS é $\Theta(nlogn)$, $\Theta(mn)$ e $O(m+n)$, respectivamente. As alterações que fizemos em LCS-MAX-SUM-LENGTH e GET-LCS-MAX-SUM das versões originais dos respectivos algoritmos não alteram o comportamento assintótico.

Vale lembrar que, o comportamento assintótico do LCS-LENGTH e GET-LCS são em função de m e n originalmente, como ambos vetores X e Y são de tamanho n, o tempo de execução passa a ser em função de n, exclusivamente.

Portanto, o consumo de tempo total do BUILD-MAX-SUM-LCS é dado na tabela 3:

Linha	Tempo
1	$\Theta(0)$
2-3	$\Theta(n)$
4	$\Theta(nlogn)$
5	$\Theta(n^2)$
6	$O(n+n)$
7	$\Theta(1)$
Total	$\Theta(n) + \Theta(nlogn) + \Theta(n^2) + O(2n) = \Theta(n^2)$

Tabela 3: Tempo de execução

21. (Bandeiras) No dia da Bandeira na Rússia o proprietário de uma loja decidiu decorar a vitrine de sua loja com faixas de tecido das cores branca, azul e vermelha. Ele deseja satisfazer as seguintes condições: faixas da mesma cor não podem ser colocadas uma ao lado da outra. Uma faixa azul sempre está entre uma branca e uma vermelha, ou uma vermelha e uma branca. Escreva um programa que, dado o número n de faixas a serem colocadas na vitrine, calcule o número de maneiras de satisfazer as condições do proprietário.

Exemplo: Para $n = 3$, o resultado são as seguintes combinações: BVB, VBV, BAV, VAB.

Resposta: Podemos pensar no processo de formação da combinação das cores da seguinte maneira: a primeira posição, obrigatoriamente, deve conter um B ou um V. Vamos fazer uma simulação iniciando com B que, ao final, basta multiplicarmos por 2 para termos o número total de combinações. No segundo nível teremos então duas possibilidades: V e A.

Percebemos então que, a cada vez que temos um A, temos apenas uma possibilidade no nível seguinte. Quando temos um B ou um V, temos duas novas possibilidades. Devemos ter o cuidado ao finalizar as combinações para que não haja nós adjacentes com o mesmo valor e, também, nenhuma terminação em A.

Logo, percebemos que o número de combinações de $1..n-1$ cresce conforme a série de Fibonacci e, portanto, podemos calcular o número de maneiras de satisfazer as condições do proprietário com a própria recorrência que calcula a série de Fibonacci:

$$t(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ t(n-1) + t(n-2) + 1, & \text{se } n \geq 2 \end{cases}$$

Como temos duas possibilidades (iniciando a série com B ou V), basta multiplicarmos o resultado da série de Fibonacci(n) por 2 e teremos, então, o número de diferentes maneiras que podemos organizar as faixas de tecido na vitrine. O algoritmo ARRANGE-FLAGS mostra a solução que, na verdade, é uma pequena alteração do algoritmo original de Fibonacci para programação dinâmica.

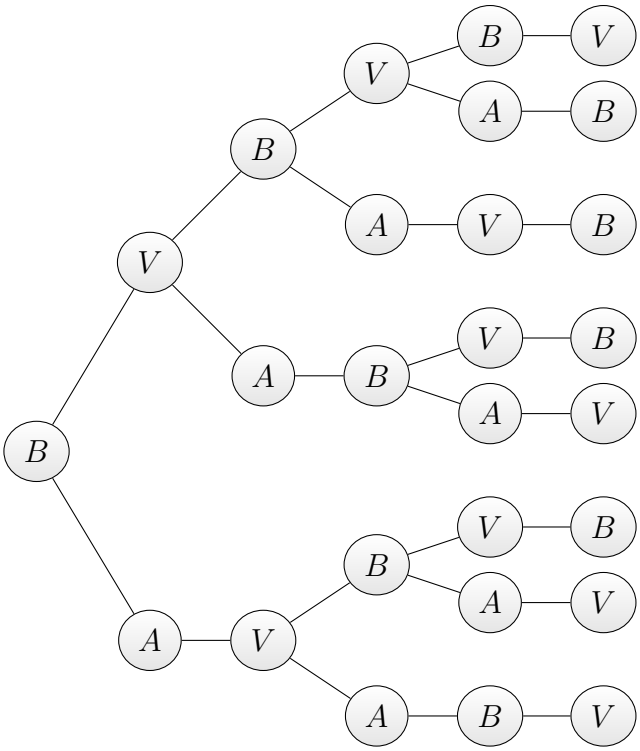


Figura 6: Árvore de combinações possíveis para $n = 6$

ARRANGE-FLAGS(n)

```
1  f[0] = 0
2  f[1] = 1
3  for i = 2 to n
4        f[i] = f[i − 1] + f[i − 2]
5  return 2 * f[n]
```

Sendo assim, no exemplo supra citado, para $n = 6$, temos que o número de combinações possíveis é 16.

Obviamente o consumo de espaço e tempo do algoritmo é $\Theta(n)$, assim como é o original.

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:

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