# 1ª Avaliação de Introdução à Análise de Algoritmos

## Prof. Glauber Cintra

1) **(0,5 pontos)** Numere as funções abaixo a partir do 1 em ordem **estritamente** crescente de dominação assintótica. Se f e g são tais que f ∈ o(g) então f deve ter número menor do que g. Se f ∈ θ(g) então f e g devem ter o mesmo número.

( 7 ) $n! \to n^n$    ( 1 ) $\log n$    ( 5 ) $3^n$    ( 1 ) $2\log n^3 \to \log(n)$    ( 5 ) $3^{n+1} \to 3^n$

( 6 ) fib(n)    ( 4 ) $n^3$    ( 2 ) $n\log n$    ( 3 ) $6n^2 \to n^2$    ( 3 ) $2n + n^2 \to n^2$

2) **(0,5 pontos)** Explique o significado dos termos *algoritmo*, *algoritmo computacional, algoritmo correto, algoritmo eficiente* e *tamanho da entrada de um algoritmo*.

**Algoritmo:** sequência de instruções que visam resolver instâncias um problema;
**Algoritmo Computacional:** algoritmo constituído apenas de instruções bem definidas, não ambíguas;
**Algoritmo Correto:** algoritmo que resolve corretamente todas as instâncias de um problema para o qual foi desenvolvido;
**Algoritmo Eficiente:** algoritmo que requer que seja executada uma quantidade de instruções elementares limitadas por um polinômio do tamanho da entrada;
**Tamanho da Entrada de um Algoritmo:** quantidade de bits para representar os dados de uma entrada.

3) **(1,5 pontos)** Indique quais são o melhor caso e o pior caso do algoritmo abaixo e sua região crítica. Qual a complexidade temporal desse algoritmo no melhor e no pior caso? Qual a complexidade espacial desse algoritmo? O algoritmo é eficiente? É de cota inferior? Justifique suas respostas. Finalmente, prove que esse algoritmo é correto.

```
Algoritmo Contido
Entrada: os vetores A e B com m e n posições, respectivamente
Saída: Sim, se todos os elementos de A estão contidos em B; Não, caso contrário
para i = 0 até m – 1
   j = 0
 enquanto j < n e A[i] ≠ B[j]
    j++
 se j >= n
    devolva Não e pare
devolva Sim
```

- <u>Melhor Caso:</u> para m ≥ n, o melhor caso ocorre quando o primeiro elemento de A não estiver contido em B. Se m < n, o melhor caso ocorre quando A[i] = B[0] (para i = 0, 1, ..., m). A complexidade temporal é de θ(min(m, n)).
- <u>Pior Caso:</u> é quando todos os elementos de A são iguais somente ao B[n-1]. A complexidade temporal é de θ(mn).
- <u>Região Critica:</u> é a condição do Enquanto (`enquanto j < n e A[i] ≠ B[j]`), porque será verificado se tal elemento de A ocorre em B.
- <u>Complexidade Espacial:</u> o espaço requerido pelo algoritmo é O(1), pois há somente declarações de variáveis para os laços.
- O algoritmo <u>é eficiente</u>, pois, no pior caso, requer um tempo quadrático no tamanho da entrada.
- O algoritmo <u>não é de cota inferior</u>. Se ordenarmos um dos vetores, podemos resolver o problema em tempo linearítmico no tamanho da entrada.
- <u>Prova de Corretude:</u>

<u>Teorema:</u> o algoritmo contido é correto

<u>Prova:</u> Suponha que um elemento *x* do vetor A não ocorra no vetor B. Logo, a 2ª condição do Enquanto (`A[i] ≠ B[j]`) sempre será satisfeita e, após percorrer o vetor B, j = n, e assim, a condição do Se será satisfeita e o algoritmo devolverá *Não*, o que é correto. Suponha agora que todos os elementos de A ocorram em B. Seja *k* a posição de um elemento *x* em A e *l* a posição desse elemento em B. Nota-se que $0 \le k \le m - 1$ e $0 \le l \le n - 1$. Quando $i = k$ e $j = l$, a 2ª condição do Enquanto não será satisfeita, consequentemente a condição do Se também é falsa. Como todos elementos de A ocorrem em B, isso sempre ocorrerá. No final do laço externo, o algoritmo devolverá *Sim*, o que é correto.

4) **(2 pontos)** Escreva um algoritmo que receba um vetor de números (e, naturalmente, seu tamanho) e devolva *Sim* se existir um número par no vetor; *Não*, caso contrário. Caracterize o pior e o melhor caso do seu algoritmo e determine a complexidade temporal no pior e no melhor caso. Determine também a complexidade espacial do algoritmo. O seu algoritmo é eficiente? É de cota inferior? Prove que seu algoritmo é correto.

```
Algoritmo temPar
   Entrada: um vetor V com n números
   Saída: Sim, se existir um número par no vetor; Não, caso contrário.
   Para i = 0 até n-1
         Se V[i] é par
               Devolva Sim e pare
   Devolva Não
```

- <u>Melhor Caso:</u> quando o primeiro elemento do vetor for par, logo, a condição será satisfeita.
- <u>Pior Caso:</u> quando não há números pares no vetor, assim o algoritmo percorrerá todo o vetor
- <u>Região Critica:</u> é a condição do Se, pois ali verifica se tal número é par.
- <u>Complexidade Temporal:</u> no <u>melhor caso</u>, logo na primeira iteração do laço, a condição do Se será verdadeira, logo o tempo requerido para este caso é O(1). Já no <u>pior caso</u>, o vetor V será todo percorrido e a condição do Se nunca será satisfeita. Por isso, o tempo requerido no pior caso é θ(n).
- <u>Complexidade Espacial:</u> o espaço requerido é O(1), pois há somente declarações de variáveis para os laços.

- O algoritmo é eficiente, pois o tempo requerido pelo algoritmo é (O(n)) e a entrada tem tamanho n, logo, o tempo é delimitado por um polinômio do tamanho da entrada.
- O algoritmo é de cota inferior, pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(n) e o algoritmo em questão requer tempo O(n).
- Prova de Corretude:

Teorema: o algoritmo temPar é correto,

Prova: Suponha que não haja nenhum elemento par no vetor A. Logo percebemos que a condição do Se nunca será satisfeita e, no final da iteração, o algoritmo devolverá *Não*, o que é correto. Suponha agora que haja um elemento par no vetor A. Seja *k* a posição desse elemento no vetor. Nota-se que $0 \leq k \leq n - 1$. Quando $i = k$, a condição do Se será satisfeita, e o algoritmo devolverá *Sim*, o que é correto.

5) **(1 ponto)** Resolva as seguintes fórmulas de recorrência:

a) T(n) = 2T(⌊n/2⌋) + n, T(1) = 1

Fazendo $n = 2^k$:

$$
\begin{array}{lrl}
\text{Eq. 0} & T(n) & = 2T\left(\frac{n}{2}\right) + n \\
\text{Eq. 1} & 2T\left(\frac{n}{2}\right) & = 4T\left(\frac{n}{4}\right) + n \\
\text{Eq. 2} & 4T\left(\frac{n}{4}\right) & = 8T\left(\frac{n}{8}\right) + n \\
\dots & \dots & \dots \\
\text{Eq. k-1} & 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) & = 2^k T(1) + n \\
\text{Eq. k} & 2^k T(1) & = 1
\end{array}
$$

-----------------------------------------------------

$T(n) = k * n + 2^k$.

Como $n = 2^k \rightarrow \log_2 n = k$, então:

$T(n) = n\log_2 n + n$. Usando notação assintótica, verifica-se que $T(n) \in \theta(n \log n)$.

b) T(n) = T(n – 1) + n/2, T(1) = ½

$$
\begin{array}{rl}
T(n) & = T(n-1) + \frac{n}{2} \\
T(n-1) & = T(n-2) + \frac{n-1}{2} \\
T(n-2) & = T(n-3) + \frac{n-2}{2} \\
\dots & \dots \\
T(2) & = T(1) + 1 \\
T(1) & = \frac{1}{2}
\end{array}
$$

---------------------------------------------------

$T(n) = \frac{1}{2} + 1 + \cdots + \frac{n-2}{2} + \frac{n-1}{2} + \frac{n}{2} = \frac{1}{2}\left(\frac{(n+1)*n}{2}\right) = \frac{n^2+n}{4}$.

Usando notação assintótica, verifica-se que $T(n) \in \theta(n^2)$.

6) **(1 ponto)** Considere o algoritmo *enigma* descrito a seguir:

```
Algoritmo enigma Entrada: um vetor V
     e a posição n se (n = 0)
     devolva V[n] se não
       devolva V[n] * enigma(V, n - 1)
```

Seja L = [8, 3, 5, 2, 9]. Simule o cálculo de *enigma*(L, 4). Determine a complexidade temporal e espacial do algoritmo *enigma* (exiba os cálculos realizados para determinar tais complexidades). Para que serve esse algoritmo? Ele é eficiente? Prove que o algoritmo é correto.

enigma(L, 4) = 9 * enigma(L, 3) = 9 * 2 * enigma(L, 2) = 9 * 2 * 5 * enigma(L, 1) = 9 * 2 * 5 * 3 * enigma(L, 0) =
= 9 * 2 * 5 * 3 * 8 = 2160

- Complexidade Temporal: Denotamos por T(n) o tempo requerido pelo algoritmo enigma(V, n), logo temos:

T(n) = T(n – 1) + c
T(n - 1) = T(n – 2) + c
... ...
T(2) = T(1) + c
T(1) = c

--------------------------------------

T(n) = n * c

Expressando em notação assintótica, concluimos que T(n) ∈ θ(n).

- Complexidade Espacial: Denotamos por E(n) o espaço requerido pelo algoritmo enigma(V, n), logo temos:

E(n) = E(n – 1) + c
E(n - 1) = E(n – 2) + c
... ...
E(2) = E(1) + c
E(1) = c
---------------------------------------
E(n) = n * c
Expressando em notação assintótica, concluímos que E(n) ∊ θ(n).

- Esse algoritmo devolve o resultado da multiplicação dos números contidos entre as posições 0 e n de V.
- O algoritmo é eficiente, pois o tempo requerido é θ(n) e a entrada tem tamanho n, ou seja, o tempo é delimitado por um polinômio do tamanho da entrada.
- Prova de corretude:

Teorema: o algoritmo enigma é correto.
Prova: por indução em n.
Base: n = 0. De fato, a multiplicação dos elementos de V[0, ..., 0] é o próprio V[0], que é retornado pelo algoritmo.
H.I.: Suponha agora que enigma(V, n – 1) resulta na multiplicação dos elementos de V[0, ..., n – 1].
Ao receber n > 0, o algoritmo devolve V[n] * enigma(V, n – 1), e de fato, resultará na multipicação dos elementos de V[0, ..., n].

7) **(1 ponto)** Escreva um algoritmo **recursivo** que receba um número *a* e um número natural *b* e devolva $a^b$. Prove que seu algoritmo é correto e determine a complexidade temporal e espacial do algoritmo (bônus de **0,5 pontos** se o algoritmo for eficiente).

```
Algoritmo: potencia
   Entrada: a ∈ N e b ∈ N
   Saída: a^b
   aux = ⌊b/2⌋
   Se b é par
         Devolva potencia(a, aux) * potencia(a, aux)
   Senão
         Devolva potencia(a, aux) * potencia(a, aux) * a
```

- Prova de Corretude:
Teorema: o algoritmo *potencia* é correto.
Prova: por indução em b
Base: b = 0. Trivial, pois $a^0 = 1$.
H.I.: Suponha, agora que o algoritmo *potencia* funciona para todo número natural menor que b. Isso quer dizer que potencia(a, $\lfloor \frac{b}{2} \rfloor$) devolve $a^{\lfloor \frac{b}{2} \rfloor}$.
Se b for par e maior que 0, *potencia* devolve:
aux * aux = $\text{potencia}\left(a, \lfloor \frac{b}{2} \rfloor\right) * \text{potencia}\left(a, \lfloor \frac{b}{2} \rfloor\right) = a^{\lfloor \frac{b}{2} \rfloor} * a^{\lfloor \frac{b}{2} \rfloor} = a^{\frac{b}{2}} * a^{\frac{b}{2}} = a^b$.
Se b for ímpar, o algoritmo devolve:
aux * aux * a = $\text{potencia}\left(a, \lfloor \frac{b}{2} \rfloor\right) * \text{potencia}\left(a, \lfloor \frac{b}{2} \rfloor\right) * a = a^{\lfloor \frac{b}{2} \rfloor} * a^{\lfloor \frac{b}{2} \rfloor} * a = a^{\frac{b-1}{2}} * a^{\frac{b-1}{2}} * a = a^b$.

- Complexidade Temporal: Denotamos por T(b) o tempo requerido pelo algoritmo. Logo temos:
$T(b) = T\left(\lfloor \frac{b}{2} \rfloor\right) + c$. Fazendo $b = 2^k$, temos $T(b) = T\left(\frac{b}{2}\right) + c$.

| | | |
|---|---|---|
| Eq. 0 | $T(b)$ | $= T\left(\frac{b}{2}\right) + c$ |
| Eq. 1 | $T\left(\frac{b}{2}\right)$ | $= T\left(\frac{b}{4}\right) + c$ |
| Eq. 2 | $T\left(\frac{b}{4}\right)$ | $= T\left(\frac{b}{8}\right) + c$ |
| ... | ... | ... |
| Eq. k-1 | $T\left(\frac{b}{2^{k-1}}\right)$ | $= T(1) + c$ |
| Eq. k | $T(1)$ | $= T(0) + c$ |
| Eq. k+1 | $T(0)$ | $= c$ |

---------------------------------------------
$T(b) = k * c + 2c$
Como $b = 2^k \rightarrow \log_2 b = k$, então:
$T(n) = c * \log_2 b + 2c$. Expresando em notação assintótica, temos que $T(b) \in \theta(log b)$.

- <u>Complexidade Espacial</u>: Denotamos por E(b) o espaço requerido pelo algoritmo. Logo temos:
$\mathrm{E}(b) = \mathrm{E}\left(\left\lfloor\frac{b}{2}\right\rfloor\right) + c$. Fazendo $b = 2^k$, temos $\mathrm{E}(b) = \mathrm{E}\left(\frac{b}{2}\right) + c$.

| | | |
|---|---|---|
| Eq. 0 | $\mathrm{E}(b)$ | $= \mathrm{E}\left(\frac{b}{2}\right) + c$ |
| Eq. 1 | $\mathrm{E}\left(\frac{b}{2}\right)$ | $= \mathrm{E}\left(\frac{b}{4}\right) + c$ |
| Eq. 2 | $\mathrm{E}\left(\frac{b}{4}\right)$ | $= \mathrm{E}\left(\frac{b}{8}\right) + c$ |
| ... | ... | ... |
| Eq. k-1 | $\mathrm{E}\left(\frac{b}{2^{k-1}}\right)$ | $= \mathrm{E}(1) + c$ |
| Eq. k | $\mathrm{E}(1)$ | $= \mathrm{E}(0) + c$ |
| Eq. k+1 | $\mathrm{E}(0)$ | $= c$ |

---------------------------------------------
$\mathrm{E}(b) = k * c + 2c$
Como $b = 2^k \rightarrow \log_2 b = k$, então:
$\mathrm{E}(n) = c * \log_2 b + 2c$. Expresando em notação assintótica, temos que $\mathrm{E}(b) \in \theta(logb)$.

8) **(0,5 pontos)** Cite o nome de um algoritmo de *cota superior* para o *problema de encontrar uma Trilha de Euler* (*Eulerian Trail*). Este algoritmo também é de cota inferior? Justifique.

O algoritmo de Hierholzer requer tempo θ(m), onde m é a quantidade de arestas. Como o tamanho da saída é θ(m), concluímos que esse algoritmo é de cota superior, logo também é de cota inferior.

9) **(1 ponto)** Escreva um algoritmo de *cota inferior* que receba uma matriz de número com *n* linhas e *m* colunas e devolva a soma dos números contidos na matriz. Determine a complexidade temporal e espacial do algoritmo e justifique porque ele é um algoritmo de cota inferior.

```
Algoritmo: somaMatriz
 Entrada: uma matriz A com n linhas e m colunas
 Saída: a soma dos elementos da matriz
 soma = 0
 Para i = 0 até n - 1
        Para j = 0 até m - 1
              soma = soma + A[i, j]
 Devolva soma
```

- <u>Complexidade Temporal:</u> o algoritmo percorre as n linhas e, para cada linha, percorre as m colunas da matriz, logo a complexidade temporal é O(nm).
- <u>Complexidade Espacial:</u> o espaço requerido é O(1), pois há somente declarações de variáveis para os laços e uma variável usada para guardar a soma dos elementos da matriz.
- O algoritmo <u>é de cota inferior</u> pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(nm) e o algoritmo em questão requer tempo O(nm).

10) **(1 ponto)** Podemos implementar uma fila *F* usando duas pilhas *P1* e *P2*. A operação *insereNaFila(x, F)* é feita empilhando se *x* em *P1*. A operação *removeDaFila(F)* é feita como indicado abaixo:

```
Algoritmo removeDaFila
   Entrada: uma fila F implementada através das pilhas P1 e P2
   Saída: remove um elemento de F
   Se P2 for vazia
         Enquanto P1 não for vazia
               empilha(desempilha(P1), P2)
   devolva desempilha(P2)  // pode haver um erro (underflow) se P2 for vazia
```

Utilizando dois dos métodos de análise amortizada que estudamos, mostre que o custo de uma sequência de *n* operações *insereNaFila* e *removeDaFila* é θ*(n)* e que o *custo amortizado* da operação *removeDaFila* é *O(1)*, considerando que inicialmente a fila está vazia.

- <u>Método contábil:</u>
Vamos atribuir custo amortizado 3 para a operação de empilhamento em P1 (gera crédito), custo amortizado 0 para as operações de desempilhamento em P1 e empilhamento em P2 (geram débito) e custo amortizado 1 para o desempilhamento em P2. Com tais custos amortizados, podemos garantir que o saldo nunca fica negativo, logo, a soma dos custos amortizados é maior ou igual à soma dos custos reais. O custo amortizado da *removeDaFila* é O(1), logo a soma dos custos reais é θ(n).
- <u>Método da agregação</u>:
Observamos que os custos dos empilhamentos e desempilhamentos é O(n). Logo, em uma sequência de n operações, o custo das operações *insereNaFila* e *removeDaFila* é θ*(n)*, e o custo amortizado da *removeDaFila* é O(n/n) = O(1).