# 1ª Lista de Exercícios de Introdução à Análise de Algoritmos
## Prof. Glauber Cintra

**Equipe:** Francisco Lucas Lima da Silva
José Mateus Azevedo de Sousa
Nicolas Holanda de Menezes
Raul Assis Cá
Victor de Sousa Firmino

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! → $n^n$ | ( 1 ) log n | ( 6 ) $3^n$ | ( 1 ) 2log $n^3$ → log(n) | ( 6 ) $3^{n+1}$ → $3^n$ |
| ( 5 ) fib(n) | ( 4 ) $n^3$ | ( 2 ) nlog n | ( 3 ) $6n^2$ → $n^2$ | ( 3 ) 2n + $n^2$ → $n^2$ |

n! = n*(n – 1)*(n – 2)*...*2*1 → (prop. Aditiva) → $n^n$
2log($n^3$) = 6log(n) → (prop. Multiplicativa) → log(n)
$3^{n+1}$ = 3*$3^n$ → (prop. Multiplicativa) → $3^n$
$6n^2$ → (prop. Multiplicativa) → $n^2$
2n + $n^2$ → (prop. Aditiva) → $n^2$
fib(n) = $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$ → (prop. Aditiva e Multiplicativa) → $\left(\frac{1+\sqrt{5}}{2}\right)^n$

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 instancias 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
```

- Melhor Caso: é quando o primeiro elemento de A não estiver em B, ou seja, para todo elemento contido no vetor B, A[0] ≠ B[j], sendo j = 0, 1, ..., n-1.
- Pior Caso: é quando o vetor A estiver contido em B, ou seja, todos os elementos de A ocorrem também em B.
- Complexidade Temporal: no melhor caso, será verificado se o primeiro elemento de A não ocorre em B. Para isso, o algoritmo percorrerá o vetor B para saber se há o elemento no vetor. Por isso, o tempo requerido para este caso é θ(n). Já no pior caso, deve-se verificar se todos os elementos de A ocorrem em B, logo, para cada elemento de A, será percorrido o vetor B para saber se o elemento ocorre no vetor. Logo, o tempo requerido para o pior caso é θ(nm).
- Região Critica: é a condição do Enquanto (`enquanto j < n e A[i] ≠ B[j]`), porque será verificado se tal elemento de A ocorre em B.
- Complexidade Espacial: o espaço requerido pelo algoritmo é O(1), pois há somente declarações de variáveis para os laços.
- O algoritmo não é eficiente, pois o tempo requerido não 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(nm) e o algoritmo em questão requer tempo O(nm).
- Prova de Corretude:

Teorema: o algoritmo contido é correto

Prova: 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
```

- Melhor Caso: quando o primeiro elemento do vetor for par, logo, a condição será satisfeita.
- Pior Caso: quando não há números pares no vetor, assim o algoritmo percorrerá todo o vetor
- Região Critica: é a condição do Se, pois ali verifica se tal número é par.
- Complexidade Temporal: no melhor caso, logo na primeira iteração do laço, a condição do Se será verdadeira, logo o tempo requerido para este caso é O(1). Já no pior caso, 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).
- Complexidade Espacial: 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 $n = 2^k$:

Eq. 0 $\quad T(n) = 2T\left(\frac{n}{2}\right) + n$

Eq. 1 $\quad 2T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + n$

Eq. 2 $\quad 4T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + n$

... $\quad$ ... $\quad$ ...

Eq. k-1 $\quad 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) = 2^kT(1) + n$

Eq. k $\quad 2^kT(1) = 2^k$

---

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

$T(n) = T(n-1) + \frac{n}{2}$

$T(n-1) = T(n-2) + \frac{n-1}{2}$

$T(n-2) = T(n-3) + \frac{n-2}{2}$

... $\quad$ ...

$T(2) = T(1) + 1$

$T(1) = \frac{1}{2}$

---

$T(n) = \frac{1}{2} + 1 + \cdots + \frac{n-2}{2} + \frac{n-1}{2} + \frac{n}{2} = \frac{1}{2}\left(\frac{(n+1)*n}{2}\right) = \frac{n^2+n}{4}$.
Usando notação assintótica, verifica-se que $T(n) \in \theta(n^2)$.

6) **(1,5 pontos)** 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.

- Simulação do Algoritmo:

L = [8, 3, 5, 2, 9]
enigma(L, 4) = V[4] * enigma(L, 3)
enigma(L, 3) = V[3] * enigma(L, 2)
enigma(L, 2) = V[2] * enigma(L, 1)
enigma(L, 1) = V[1] * enigma(V, 0)
enigma(V, 0) = V[0] = 8
enigma(L, 1) = 3 * 8 = 24
enigma(L, 2) = 5 * 24 = 120
enigma(L, 3) = 2 * 120 = 240
enigma(L, 4) = 9 * 240 = 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 $T(n) \in \theta(n)$.

- 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, concluimos que $E(n) \in \theta(n)$.

- Esse algoritmo serve para, dado um valor *n*, multiplicar os elementos do vetor V, da posição 0 até a posição *n*, ou seja, multiplicar os valores de *V[0, ..., n]*.
- 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 e b ∈ N
  Saída: a^b
  Se b = 0 Devolva 1
  aux = potencia(a, ⌊b/2⌋)
  Se b é par
        Devolva aux * aux
  Senão
        Devolva aux * 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 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 = potencia $\left(a, \lfloor \frac{b}{2} \rfloor\right)$ * 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 impar, o algoritmo devolve:

aux * aux * a = potencia $\left(a, \lfloor \frac{b}{2} \rfloor\right)$ * 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:

$T(b) = T\left(\left\lfloor \frac{b}{2} \right\rfloor\right) + c$. 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:

$T(n) = c * \log_2 b + 2c$. Expressando em notação assintótica, temos que $T(b) \in \theta(\log b)$.

- Complexidade Espacial:

Denotamos por E(b) o tempo requerido pelo algoritmo. Logo temos:

$E(b) = E\left(\left\lfloor \frac{b}{2} \right\rfloor\right) + c$. Fazendo $b = 2^k$, temos $E(b) = E\left(\frac{b}{2}\right) + c$.

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

---

$E(b) = k * c + 2c$

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

$E(n) = c * \log_2 b + 2c$. Expressando em notação assintótica, temos que $E(b) \in \theta(\log b)$.

- O algoritmo é eficiente, pois o tempo requerido pelo algoritmo (θ(log b)) é delimitado por um polinômio do tamanho da entrada(log b + log a).

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*, proposto em 1873, foi um dos primeiros a tratar trilhas eulerianas. A ideia é, a partir de um vértice qualquer, percorrer arestas até retornar ao vértice inicial. Porém, desta forma, pode ser obtido um ciclo que não inclua todas as arestas do grafo. Enquanto houver um vértice que possui arestas ainda não exploradas, comece um caminho neste vértice e tente voltar a ele, usando somente arestas ainda não percorridas. O tempo requerido para este algoritmo é O(m), ou seja, requer tempo de, no máximo, proporcional ao numero de arestas. Este é um algoritmo de cota superior, pois todo algoritmo conhecido que resolva o problema gasta tempo O(m), e o algoritmo de Hierholzer gasta tempo O(m). No entanto, o algoritmo não é de cota inferior, pois o tempo requerido para resolver o problema é Ω($m^2$).

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
```

- Complexidade Temporal: o algoritmo percorre as n linhas e, para cada linha, percorre as m colunas da matriz, logo a complexidade temporal é O(nm).
- Complexidade Espacial: 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 é de cota inferior 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) **(2 pontos)** 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.

- Método da Agregação:

Sabemos que os métodos *empilha* e *desemplilha* gasta tempo constante (O(1)), e o método *insereNaFila* é uma chamada ao método *empliha*, logo também gasta tempo O(1). A função *removeDaFila* consiste nas chamadas aos métodos *empilha* e *desempilha*. Se a fila possui k elementos, o *removeDaFila* requer tempo O(k). Para *n* chamadas, o custo total das n chamadas à *insereNaFila* é θ(n) é o custo total das chamadas à *removeDaFila* é também θ(n). Logo o custo total de n operações é θ(2n) = θ(n) e o custo amortizado de cada operação é θ(n/n) = O(1).

- Método Contábil:

Atribuindo custo 2 para o método *empilha* (crédito) e custo 0 para o *desempliha* (débito), verificamos então que o método *insereNaFila* terá custo 2 (crédito) e o *removeDaFila* terá custo 2 (débito). Ao analisarmos o custo, verificamos que o saldo nunca fica negativo. Dessa forma, os custos dos métodos *insereNaFila* e *removeDaFila* é 2, ou seja, constante. Concluimos então que, para n chamadas aos métodos, a soma dos custos reais é θ(n) e o custo amortizado de cada operação é O(1).