Complexidade de Algoritmos

Disciplina: Estruturas de Dados

Professor: Marcelo Andrade Teixeira

Introdução

- Um **algoritmo** é um processo sistemático para a **resolução de um problema**
- O desenvolvimento de algoritmos é particularmente importante para problemas a serem solucionados em um computador
- Um algoritmo computa uma **saída**, o resultado do problema, a partir de uma **entrada**, as informações inicialmente conhecidas e que permitem encontrar a solução do problema

Introdução (cont.)

- Durante o processo de computação, o algoritmo manipula **dados**, gerados a partir de sua entrada
- Quando os dados são dispostos e manipulados de uma forma homogênea, constituem um **tipo abstrato de dados**
- Este é composto por um modelo matemático acompanhado por um conjunto de operações definido sobre esse modelo

Introdução (cont.)

- Um algoritmo é projetado em termos de tipos abstratos de dados
- Para implementá-los numa linguagem de programação, é necessário encontrar uma forma de representá-los nessa linguagem, utilizando tipos e operações suportadas pelo computador
- Na representação do modelo matemático emprega-se uma **estrutura de dados**

Introdução (cont.)

- As estruturas diferem umas das outras pela disposição ou manipulação de seus dados
- A escolha correta da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente
- O conhecimento de princípios de <u>**complexidade computacional**</u> é, portanto, requisito básico para se avaliar corretamente a adequação de uma estrutura de dados

Complexidade de Algoritmos

- A teoria de complexidade é utilizada para avaliar o desempenho de algoritmos
- Na avaliação desse desempenho podemos considerar o tempo de execução ou a memória utilizada
- Conceitos básicos: problema e instância
- Problema: Uma descrição formalizada de todos os seus parâmetros (entrada de dados, variáveis)

Complexidade (cont.)

- Instância: Consiste de um conjunto de valores associados aos parâmetros que caracterizam a entrada do problema
- Tamanho de uma instância: número inteiro que representa a "quantidade" dos dados de entrada (custo logarítmico – número de bits necessários para codificar essa entrada)
- Exemplos: Calcular 6! →

  tamanho da instância = 6

Complexidade (cont.)

- Calcular n! → tamanho da instância = n
- Ordenar um vetor com 6 elementos →

  tamanho da instância = 6
- Ordenar um vetor com n elementos →

  tamanho da instância = n
- Multiplicação de matrizes:

  $$(A)_{m \times n} * (B)_{n \times p} = (C)_{m \times p} \quad \rightarrow$$

  tamanho da instância = m.n + n.p

Complexidade (cont.)

- Definimos <u>**complexidade de tempo**</u> de um algoritmo A a uma função T: tam(I) $\rightarrow R^+$ onde tam(I) representa o tamanho de uma instância I qualquer e R^+ representa o número total de unidades de tempo para processar essa instância.
- Seja um algoritmo A e $C = \{E_1, E_2, ..., E_n\}$ um conjunto de entradas de A.
- Seja $T(E_k)$ o tempo de A para computação da entrada E_k e $p(E_k)$ a probabilidade da entrada E_k ocorrer.

Complexidade (cont.)

- <u>**Complexidade de Pior Caso**</u>: estaremos interessados em $\max_{1 \leq k \leq n}\{T(E_k)\}$

- <u>**Complexidade de Melhor Caso**</u>: estaremos interessados em $\min_{1 \leq k \leq n}\{T(E_k)\}$

- <u>**Complexidade de Caso Médio**</u>: estaremos interessados em $\sum^{n}_{k=1} p(E_k).T(E_k)$

Critério de Custo Uniforme

- No critério de custo uniforme a complexidade local de um algoritmo A é o número total de passos necessários para solucionar o problema P para uma certa entrada E
- O tempo de execução de cada instrução do algoritmo é considerado igual à 1 unidade de tempo
- Para o cálculo da complexidade local desprezamos instruções de entrada de dados, saída de dados e atribuições

Complexidade Local

- Exemplo:

  Calcular a complexidade local T(n) do algoritmo que calcula o valor de f(n), onde

  $f(n) = n^n$ se $n > 0$, e

  $f(n) = 1$ se $n \leq 0$

```
Algoritmo
    ler(n)
    se n ≤ 0 então
        escrever("f(n) = 1")
    senão
        X ← n
        Y ← n – 1
        enquanto Y > 0 faça
            X ← X * n
            Y ← Y – 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

```
Algoritmo
    ler(n)
    se [n ≤ 0] então
        escrever("f(n) = 1")
    senão
        X ← n
        Y ← [n – 1]
        enquanto [Y > 0] faça
            X ← [X * n]
            Y ← [Y – 1]
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n – 1 vezes

```
Algoritmo
    ler(n)
    se n ≤ 0 então
        escrever("f(n) = 1")
    senão
        X ← n
        Y ← n – 1
        enquanto Y > 0 faça
            X ← X * n
            Y ← Y – 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n – 1 vezes

$$T(n) = 1 + 1 + (n - 1).3 + 1$$

$$T(n) = 3 + 3n - 3$$

$$T(n) = 3n$$

```
Algoritmo
   ler(n)
   se n ≤ 0 então
         escrever("f(n) = 1")
   senão
         X ← n
         Y ← 1
         enquanto Y < n faça
                  X ← X * n
                  Y ← Y + 1
         fim_enquanto
         escrever("f(n) = ", X)
   fim_se
fim_algoritmo
```

Algoritmo

ler(n)

se $\boxed{n \leq 0}$ então

escrever("f(n) = 1")

senão

$X \leftarrow n$

$Y \leftarrow 1$

enquanto $\boxed{Y < n}$ faça

$X \leftarrow \boxed{X * n}$

$Y \leftarrow \boxed{Y + 1}$

fim_enquanto

n – 1 vezes

escrever("f(n) = ", X)

fim_se

fim_algoritmo

```
Algoritmo
    ler(n)
    se n ≤ 0 então
        escrever("f(n) = 1")
    senão
        X ← n
        Y ← 1
        enquanto Y < n faça
            X ← X * n
            Y ← Y + 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n – 1 vezes

$$\mathbf{T(n) = 1 + (n - 1).3 + 1}$$

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

$$\mathbf{T(n) = 3n - 1}$$

```
Algoritmo
    ler(n)
    se n < 0 então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← 1
        enquanto Y ≤ n faça
            X ← X * n
            Y ← Y + 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

```
Algoritmo
    ler(n)
    se n < 0 então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← 1
        enquanto Y ≤ n faça
            X ← X * n
n vezes     Y ← Y + 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

```
Algoritmo
    ler(n)
    se [n < 0] então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← 1
        enquanto [Y ≤ n] faça
            X ← [X * n]
            Y ← [Y + 1]
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n vezes

T(n) = 1 + n.3 + 1

T(n) = 3n + 2

```
Algoritmo
    ler(n)
    se n < 0 então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← n
        enquanto Y > 0 faça
            X ← X * n
            Y ← Y – 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

```
Algoritmo
    ler(n)
    se [n < 0] então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← n
        enquanto [Y > 0] faça
            X ← [X * n]
            Y ← [Y – 1]
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n vezes

```
Algoritmo
    ler(n)
    se n < 0 então
        escrever("f(n) = 1")
    senão
        X ← 1
        Y ← n
        enquanto Y > 0 faça
              X ← X * n
              Y ← Y – 1
        fim_enquanto
        escrever("f(n) = ", X)
    fim_se
fim_algoritmo
```

n vezes

$$\mathbf{T(n) = 1 + n.3 + 1}$$

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

Complexidade Local (cont.)

- Exemplo: Sejam 5 algoritmos A_1, A_2, ..., A_5 com as seguintes complexidades de tempo:

Algoritmo	Complexidade
A_1	$T(n) = n$
A_2	$T(n) = n.\log n$
A_3	$T(n) = n^2$
A_4	$T(n) = n^3$
A_5	$T(n) = 2^n$

Complexidade Local (cont.)

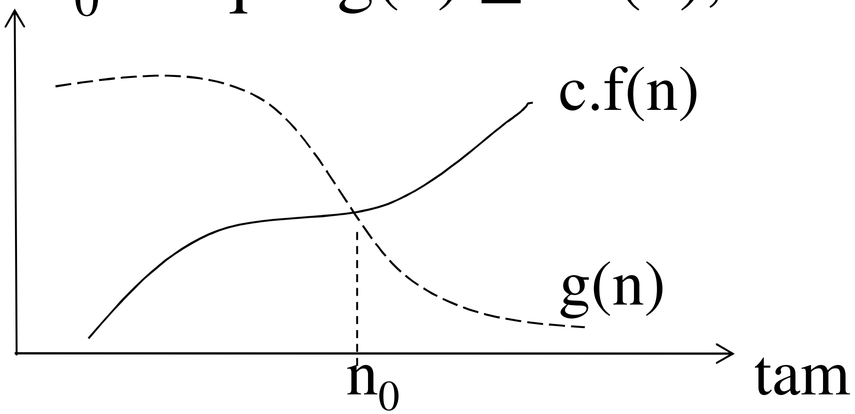
- Qual o maior tamanho possível de n para resolver o problema em 1 segundo?
- O tempo de execução de cada instrução do algoritmo é igual à 1 unidade de tempo
- Supondo 1 unidade de tempo = 1 milisegundo

Complexidade Local (cont.)

Algoritmo	Complexidade	Tamanho máximo
A_1	$T(n) = n$	1000
A_2	$T(n) = n.\log n$	140
A_3	$T(n) = n^2$	31
A_4	$T(n) = n^3$	10
A_5	$T(n) = 2^n$	9

Complexidade Assintótica

- Na maioria dos casos é difícil determinar a função de complexidade local T(.) considerando o pior caso
- Sejam f e g: $N \rightarrow R$ duas funções. Diremos que f domina g assintoticamente quando existem duas constantes c e n_0 tal que $g(n) \leq c.f(n), \forall\, n \geq n_0$



Complexidade Assintótica (cont.)

- Para exprimir a complexidade assintótica de um algoritmo utilizaremos a notação O(.)
- $T(n)$ é de ordem $O(f(n))$ se $\exists$ k e n_0 tal que
  $T(n) \leq k.f(n), \forall\ n \geq n_0$
- Exemplo: $T(n) = 3n^3 + 2n^2$ é de ordem $O(n^3)$?

Devemos encontrar k e n_0 tal que $T(n) \leq k.f(n), \forall\ n \geq n_0$

$T(n) \leq k.f(n) \Rightarrow 3n^3 + 2n^2 \leq k.n^3 \Rightarrow k.n^3 \geq 3n^3 + 2n^2 \Rightarrow$

$k.n^3 - 3n^3 \geq 2n^2 \Rightarrow n^3.(k-3) \geq 2n^2 \Rightarrow n.(k-3) \geq 2 \Rightarrow$

Complexidade Assintótica (cont.)

$n.(k - 3) \geq 2 \Rightarrow n \geq 2 / (k - 3)$

Podemos escolher $k = 4$ e assim

$n \geq 2 / (4 - 3) \Rightarrow n \geq 2$

Como isso deve valer para $\forall\ n \geq n_0$ escolhemos $n_0 = 2$

- Exemplo: $T(n) = 542$ é de ordem $O(1)$?

Devemos encontrar k e n_0 tal que $T(n) \leq k.f(n),\ \forall\ n \geq n_0$

$T(n) \leq k.f(n) \Rightarrow 542 \leq k.1 \Rightarrow k \geq 542$

Basta tomar um n_0 qualquer e um $k \geq 542$

Complexidade Assintótica (cont.)

- Exemplo: $T(n) = 3^n$ é de ordem $O(2^n)$?

Devemos encontrar k e n_0 tal que $T(n) \leq k.f(n), \forall\ n \geq n_0$

$T(n) \leq k.f(n) \Rightarrow 3^n \leq k.2^n \Rightarrow k.2^n \geq 3^n \Rightarrow k \geq (3/2)^n$

Quando $n \to \infty$ temos:

$\lim_{n \to \infty}(3/2)^n = +\infty$

Isso é um absurdo, pois o valor de k não pode ser maior ou igual à $+\infty$.

Logo, $T(n) = 3^n$ não é de ordem $O(2^n)$

Algoritmos com Vetores

- *Arrays* ou arranjos ou vetores ou matrizes são agrupamentos homogêneos de dados
- Uma variável desse tipo pode armazenar uma sequência de vários dados que possuem um mesmo tipo
- Por exemplo, VET $\leftarrow$ {2.1, 3.4, 4.7} é uma variável que armazena uma sequência de três valores reais
- O algoritmo a seguir coloca esses três valores no vetor VET

Algoritmos com Vetores (cont.)

Algoritmo

    VET[1] $\leftarrow$ 2.1

    VET[2] $\leftarrow$ 3.4

    VET[3] $\leftarrow$ 4.7

fim_algoritmo

- O algoritmo a seguir cria o seguinte vetor com n elementos: VET $\leftarrow$ {3.1, 5.1, 7.1, 9.1, 11.1, ...}

```
Algoritmo
   VET[1] ← 3.1
   I ← 2
   enquanto I ≤ n faça
      VET[I] ← VET[I – 1] + 2
      I ← I + 1
   fim_enquanto
fim_algoritmo
```

- Calcule a complexidade local **T(n)** desse algoritmo
- Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**)

```
Algoritmo
    VET[1] ← 3.1
    I ← 2
    enquanto I ≤ n faça
        VET[I] ← VET[I – 1] + 2
        I ← I + 1
    fim_enquanto
fim_algoritmo
```

n – 1 vezes

$$\boxed{\begin{aligned} T(n) &= (n-1).4 + 1 \\ T(n) &= 4n - 4 + 1 \\ T(n) &= 4n - 3 \end{aligned}}$$

T(n) é de ordem O(n)

Algoritmos com Vetores (cont.)

- $T(n) = 4n - 3$ é de ordem $O(n)$?

Devemos encontrar k e n_0 tal que $T(n) \leq k.f(n), \forall\ n \geq n_0$

$T(n) \leq k.f(n) \Rightarrow 4n - 3 \leq k.n \Rightarrow k.n \geq 4n - 3 \Rightarrow$

$k.n - 4n \geq - 3 \Rightarrow n.(k - 4) \geq - 3 \Rightarrow n \geq - 3 / (k - 4) \Rightarrow$

$n \geq 3 / (4 - k)$

Podemos escolher $k = 3$ e assim

$n \geq 3 / (4 - 3) \Rightarrow n \geq 3$

Como isso deve valer para $\forall\ n \geq n_0$ escolhemos $n_0 = 3$

{Algoritmo para calcular o somatório dos elementos de um vetor de n elementos}

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
    I ← 1
    enquanto I ≤ n faça
        ler(X[I])
        I ← I + 1
    fim_enquanto
    SOMA ← 0
    I ← 1
    enquanto I ≤ n faça
        SOMA ← SOMA + X[I]
        I ← I + 1
    fim_enquanto
    escrever("O somatório é ", SOMA)
fim_algoritmo
```

{Algoritmo para calcular o somatório dos elementos de um vetor de n elementos}

```
        Algoritmo
              I ← 1
            ┌ enquanto [I ≤ n] faça
   n        │         ler(X[I])
 vezes      ┤         I ← [I + 1]
            └ fim_enquanto
              SOMA ← 0
              I ← 1
            ┌ enquanto [I ≤ n] faça
   n        │         SOMA ← [SOMA + X[I]]
 vezes      ┤         I ← [I + 1]
            └ fim_enquanto
              escrever("O somatório é ", SOMA)
        fim_algoritmo
```

$$\boxed{\begin{array}{l}\mathbf{T(n) = n.2 + 1 + n.3 + 1} \\ \\ \mathbf{T(n) = 5n + 2}\end{array}}$$

$$\boxed{\mathbf{T(n)\ \acute{e}\ de\ ordem\ O(n)}}$$

{Algoritmo para calcular o somatório dos elementos de um vetor de n elementos}

```
Procedimento LERVETOR(var V)
 I ← 1
 enquanto I ≤ n faça
  ler(V[I])
  I ← I + 1
 fim_enquanto
fim_procedimento
```

```
Função SOMATORIO(V)
 SOMA ← 0
 I ← 1
 enquanto I ≤ n faça
  SOMA ← SOMA + V[I]
  I ← I + 1
 fim_enquanto
 SOMATORIO ← SOMA
fim_função
```

```
Algoritmo
 LERVETOR(X)
 RESULTADO ← SOMATORIO(X)
 escrever("O somatório é ", RESULTADO)
fim_algoritmo
```

- Qual a complexidade local e a complexidade assintótica da procedimento LERVETOR?

```
Procedimento LERVETOR(var V)
  I ← 1
  enquanto [I ≤ n] faça
    ler(V[I])
    I ← [I + 1]
  fim_enquanto
fim_procedimento
```

n vezes

$$\boxed{\begin{array}{l} T(n) = n.2 + 1 \\ \\ T(n) = 2n + 1 \end{array}}$$

$$\boxed{T(n) \text{ é de ordem } O(n)}$$

- Qual a complexidade local e a complexidade assintótica do função SOMATORIO?

```
Função SOMATORIO(V)
  SOMA ← 0
  I ← 1
  enquanto [I ≤ n] faça
    SOMA ← [SOMA + V[I]]
    I ← [I + 1]
  fim_enquanto
  SOMATORIO ← SOMA
fim_função
```

n vezes

$T(n) = n.3 + 1$

$T(n) = 3n + 1$

$T(n)$ é de ordem $O(n)$

{ Algoritmo para calcular o máximo de um vetor com n elementos }

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
 LERVETOR(X)
 MAX ← X[1]
 I ← 2
 enquanto I ≤ n faça
  se X[I] > MAX então
   MAX ← X[I]
  fim_se
  I ← I + 1
 fim_enquanto
 escrever("O máximo é ", MAX)
fim_algoritmo
```

{ Algoritmo para calcular o máximo de um vetor com n elementos }

```
Algoritmo
 LERVETOR(X)
 MAX ← X[1]
 I ← 2
 enquanto I ≤ n faça
   se X[I] > MAX então
     MAX ← X[I]
   fim_se
   I ← I + 1
 fim_enquanto
 escrever("O máximo é ", MAX)
fim_algoritmo
```

n – 1 vezes

$T(n) = (n - 1).3 + 1$

$T(n) = 3n - 3 + 1$

$T(n) = 3n - 2$

T(n) é de ordem O(n)

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
   I ← 1
   enquanto I ≤ n faça
      ler(VET[I])
      VET[I] ← VET[I] + 2
      I ← I + 1
   fim_enquanto
   VAL ← 5 * (1 + VET[1] + VET[n]) / 7
   I ← 2
   enquanto I < n faça
      se VET[I] > VAL ou VAL < 0 então
         VAL ← VET[I]
      fim_se
      VAL ← VAL + VET[I]
      I ← I + 1
   fim_enquanto
   VAL ← 2 * VAL / 3
   escrever("O valor é ", VAL)
fim_algoritmo
```

```
Algoritmo
   I ← 1
   enquanto I ≤ n faça
      ler(VET[I])
      VET[I] ← VET[I] + 2
      I ← I + 1
   fim_enquanto
   VAL ← 5 * (1 + VET[1] + VET[n]) / 7
   I ← 2
   enquanto I < n faça
      se VET[I] > VAL ou VAL < 0 então
         VAL ← VET[I]
      fim_se
      VAL ← VAL + VET[I]
      I ← I + 1
   fim_enquanto
   VAL ← 2 * VAL / 3
   escrever("O valor é ", VAL)
fim_algoritmo
```

n vezes (primeiro enquanto)

n – 2 vezes (segundo enquanto)

$T(n) = n * 3 + 1 + 4 + (n - 2) * 6 + 1 + 2$

$T(n) = 8 + 3n + 6n - 12$

$T(n) = 9n – 4$

T(n) é de ordem O(n)

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
   I ← 2
   enquanto I ≤ n faça
      ler(VET[I])
      I ← I + 1
   fim_enquanto
   VET[1] ← 2 * VET[2]
   I ← 1
   enquanto I < n faça
      escrever(VET[I])
      se VET[I] < 0 e VET[I] > VET[n] então
         VET[I] ← 0
      fim_se
      I ← I + 1
   fim_enquanto
   VET[n] ← (VET[1] + VET[2] + 1) * 3
   escrever(VET[n])
fim_algoritmo
```

$$T(n) = (n - 1) * 2 + 1 + 1 + (n - 1) * 5 + 1 + 3$$
$$T(n) = 6 + 2n - 2 + 5n - 5$$
$$T(n) = 7n - 1$$

T(n) é de ordem O(n)

```
       Algoritmo
          I ← 2
       ┌  enquanto [I ≤ n] faça
 n – 1 │     ler(VET[I])
 vezes │     I ← [I + 1]
       └  fim_enquanto
          VET[1] ← [2 * VET[2]]
          I ← 1
       ┌  enquanto [I < n] faça
       │     escrever(VET[I])
       │     se [VET[I] < 0] [e] [VET[I] > VET[n]] então
 n – 1 │        VET[I] ← 0
 vezes │     fim_se
       │     I ← [I + 1]
       └  fim_enquanto
          VET[n] ← ([VET[1] +] [VET[2] + 1)] [* 3]
          escrever(VET[n])
       fim_algoritmo
```

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
   I ← 1
   enquanto I < n faça
      ler(VET[I])
      VET[I] ← (VET[I] + 1) * 3
      I ← I + 1
   fim_enquanto
   VET[n] ← VET[1] / 2
   I ← 1
   enquanto I ≤ n faça
      J ← 2
      enquanto J ≤ n faça
         se VET[I] > VET[J] então
            VET[I] ← VET[J]
         fim_se
         escrever(VET[I])
         J ← J + 1
      fim_enquanto
      I ← I + 1
   fim_enquanto
fim_algoritmo
```

```
          Algoritmo
            I ← 1
            enquanto I < n faça
              ler(VET[I])
n – 1         VET[I] ← (VET[I] + 1) * 3
vezes         I ← I + 1
            fim_enquanto
            VET[n] ← VET[1] / 2
            I ← 1
            enquanto I ≤ n faça
              J ← 2
              enquanto J ≤ n faça
                se VET[I] > VET[J] então
n                 VET[I] ← VET[J]
vezes  n – 1    fim_se
       vezes    escrever(VET[I])
                J ← J + 1
              fim_enquanto
              I ← I + 1
            fim_enquanto
          fim_algoritmo
```

$$T(n) = (n - 1) * 4 + 1 + 1 + n * (1 + (n - 1) * 3 + 1 + 1) + 1$$

$$T(n) = 3 + 4n - 4 + n * (3 + 3n - 3)$$

$$T(n) = 4n - 1 + n * 3n$$

$$T(n) = 3n^2 + 4n – 1$$

$T(n)$ é de ordem $O(n^2)$

Calcule a complexidade local **T(n)** desse algoritmo.

Indique qual é a complexidade assintótica **O(.)** desse algoritmo (**T(n)** é de ordem **O(.)**).

```
Algoritmo
   I ← 1
   enquanto I ≤ n faça
      ler(VET[I])
      VET[I] ← VET[I] * 2
      I ← I + 1
   fim_enquanto
   VAL ← 0
   I ← 1
   enquanto I ≤ n faça
      J ← 1
      enquanto J < n faça
         VAL ← (VAL + VET[I] + VET[J]) / 3
         J ← J + 1
      fim_enquanto
      I ← I + 1
   fim_enquanto
   escrever("O valor é ", VAL)
fim_algoritmo
```

```
Algoritmo
   I ← 1
   enquanto I ≤ n faça
      ler(VET[I])
      VET[I] ← VET[I] * 2
      I ← I + 1
   fim_enquanto
   VAL ← 0
   I ← 1
   enquanto I ≤ n faça
      J ← 1
      enquanto J < n faça
         VAL ← (VAL + VET[I] + VET[J]) / 3
         J ← J + 1
      fim_enquanto
      I ← I + 1
   fim_enquanto
   escrever("O valor é ", VAL)
fim_algoritmo
```

n vezes (primeiro enquanto)

n vezes (segundo enquanto)

n – 1 vezes (enquanto interno)

$T(n) = n * 3 + 1 + n * (1 + (n - 1) * 5 + 1 + 1) + 1$

$T(n) = 2 + 3n + n * (3 + 5n - 5)$

$T(n) = 3n + 2 + n * (5n - 2)$

$T(n) = 3n + 2 + 5n^2 - 2n$

$T(n) = 5n^2 + n + 2$

T(n) é de ordem $O(n^2)$

Recursividade

- Consiste basicamente em utilizar, direta ou indiretamente, um procedimento (ou função) dentro do mesmo procedimento (ou função) que o define
- Exemplo: fatorial de N

$$n! = \begin{cases} 1 \text{ se } n = 0 & \text{(base da recursão)} \\ n.(n-1)! \text{ se } n > 0 & \text{(fórmula de recorrência)} \end{cases}$$

```
Função FATORIAL(n)
    se n = 0 então
        FATORIAL ← 1
    senão
        FATORIAL ← n * FATORIAL(n – 1)
    fim_se
fim_função

Algoritmo
    ler(NUM)
    RES ← FATORIAL(NUM)
    escrever(NUM, "! = ", RES)
fim_algoritmo
```

Recursividade (cont.)

- Se no algoritmo anterior o valor lido for 5, deverá ser calculado o FATORIAL(5):

  FATORIAL(5) = 5 * FATORIAL(4)

  FATORIAL(4) = 4 * FATORIAL(3)

  FATORIAL(3) = 3 * FATORIAL(2)

  FATORIAL(2) = 2 * FATORIAL(1)

  FATORIAL(1) = 1 * FATORIAL(0)

  FATORIAL(0) = 1

Recursividade (cont.)

- Logo, o resultado para o FATORIAL(5) será :

  FATORIAL(5) = 5 * 24 = 120

  FATORIAL(4) = 4 * 6 = 24

  FATORIAL(3) = 3 * 2 = 6

  FATORIAL(2) = 2 * 1 = 2

  FATORIAL(1) = 1 * 1 = 1

  FATORIAL(0) = 1

Recursividade (Complexidade)

- A função FATORIAL exige 3 instruções elementares (1 comparação, 1 multiplicação e 1 subtração) mais a função de complexidade para a chamada recursiva
- Logo $T(n) = 3 + T(n - 1)$ (I)
- $T(n - 1) = 3 + T(n - 2)$ (II)
- Substituindo (II) em (I):

  $T(n) = 3 + 3 + T(n - 2)$ (III)

Recursividade (Complexidade)

- $T(n) = 3 + 3 + T(n - 2)$ (III)
- $T(n - 2) = 3 + T(n - 3)$ (IV)
- Substituindo (IV) em (III):

  $T(n) = 3 + 3 + 3 + T(n - 3)$ (V)
- $T(n - 3) = 3 + T(n - 4)$ (VI)
- Substituindo (VI) em (V):

  $T(n) = 3 + 3 + 3 + 3 + T(n - 4) \Rightarrow$

  $T(n) = 3.4 + T(n - 4)$ (VII)

Recursividade (Complexidade)

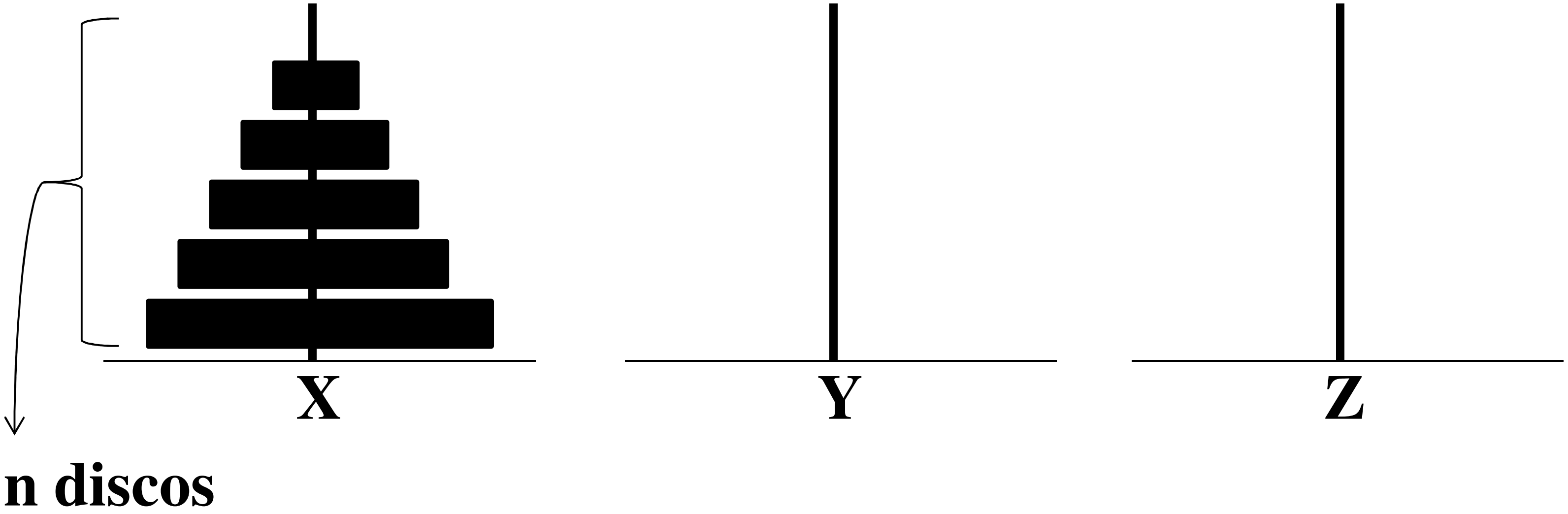
- $T(n) = 3.4 + T(n - 4)$ (VII)
- Generalizando: $T(n) = 3.k + T(n - k)$ (VIII)
- Esperamos chegar na base da recursão:

  $n - k = 0 \Rightarrow n = k$ (IX)
- Substituindo (IX) em (VIII):

  $T(n) = 3.n + T(0)$
- Como $T(0) = 1$ (base da recursão) temos:

  $T(n) = 3n + 1$ que é de ordem $O(n)$

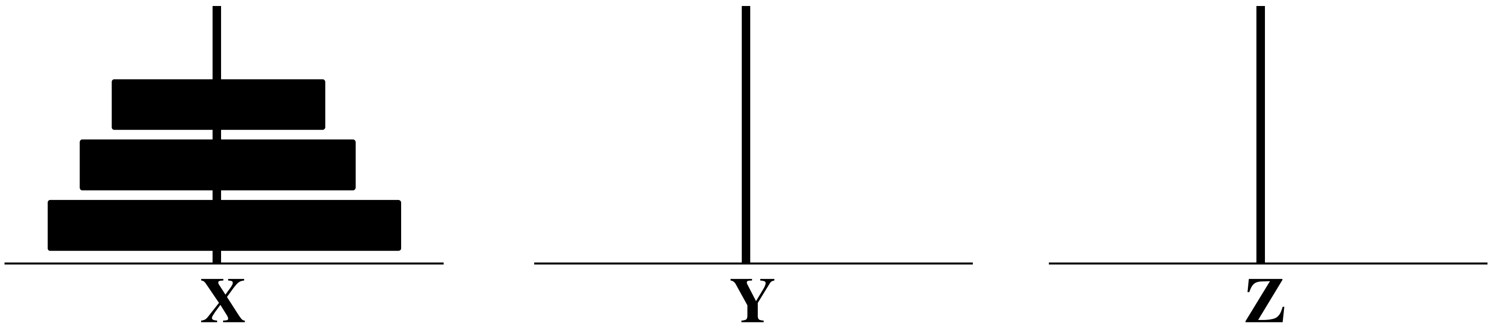
Torres de Hanói

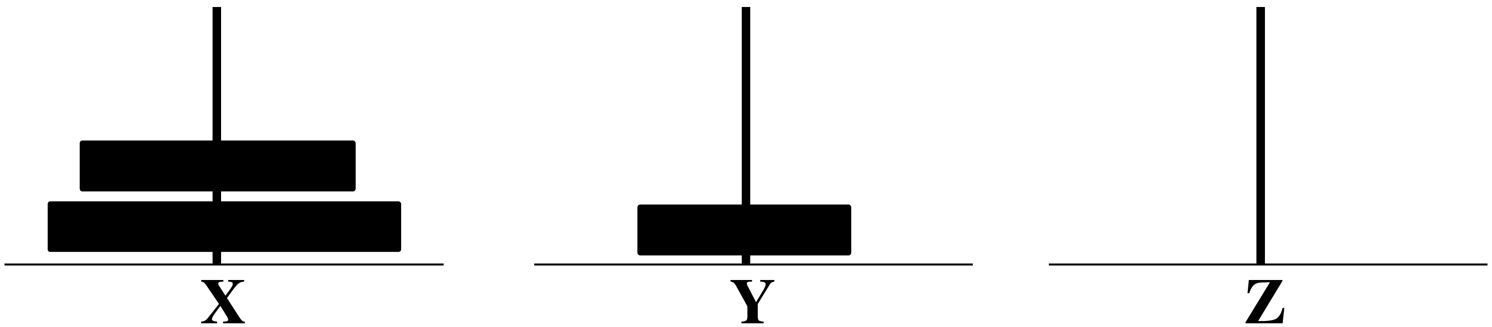


- Regras:
  - Só um disco pode ser movimentado de cada vez
  - Um disco maior não poder ser colocado sobre um menor

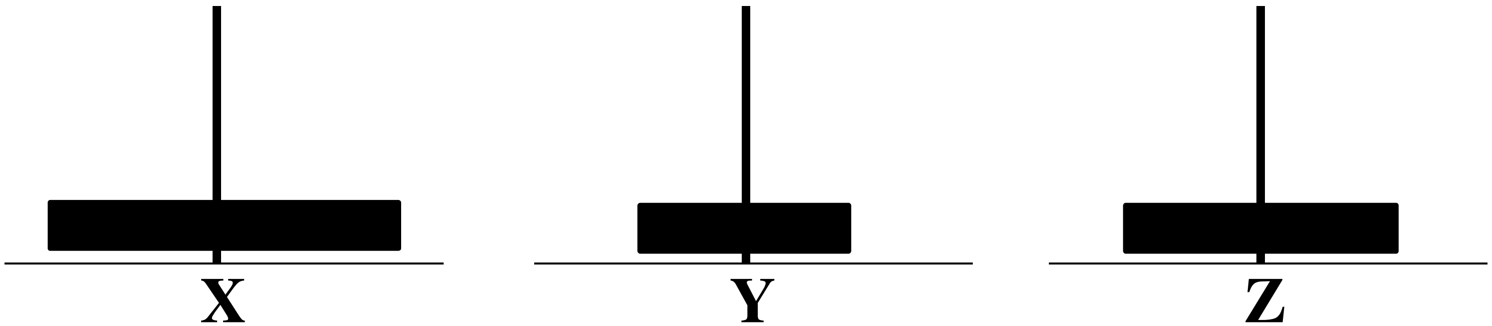
Torres de Hanói (cont.)

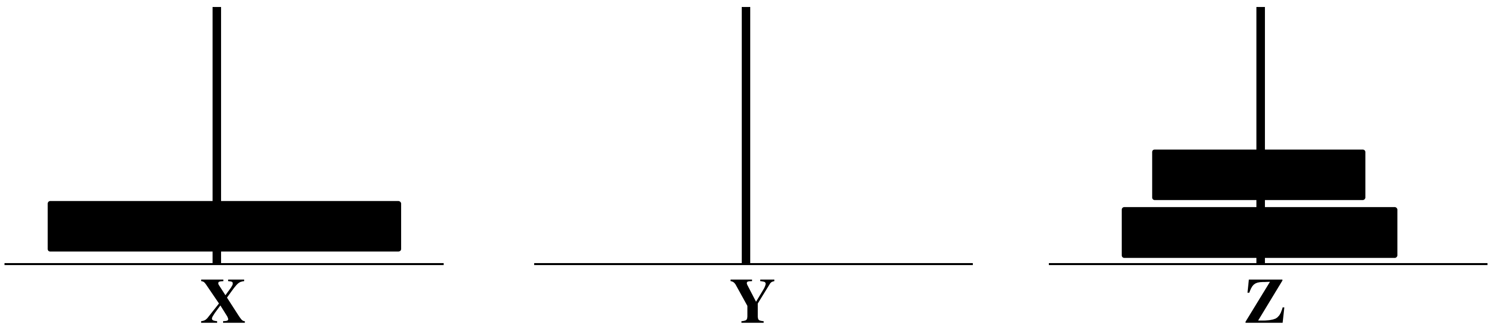
n = 3 discos



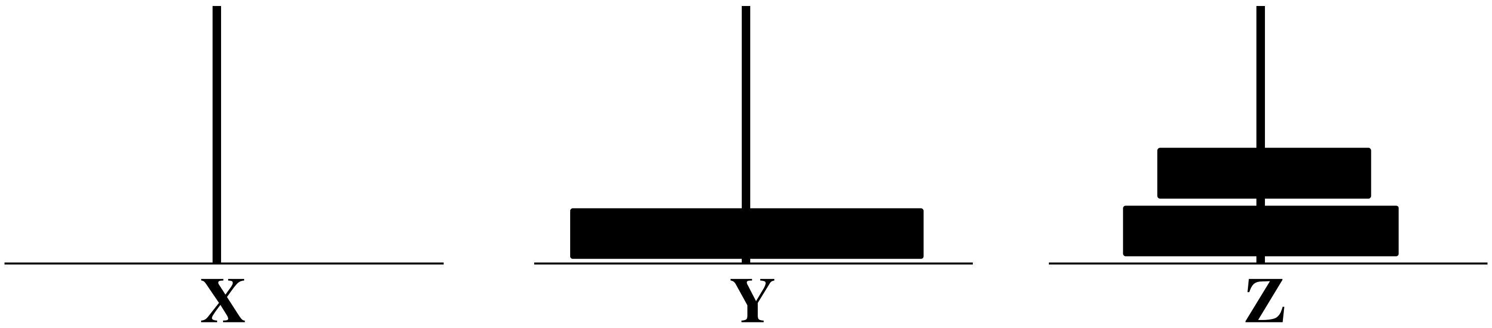


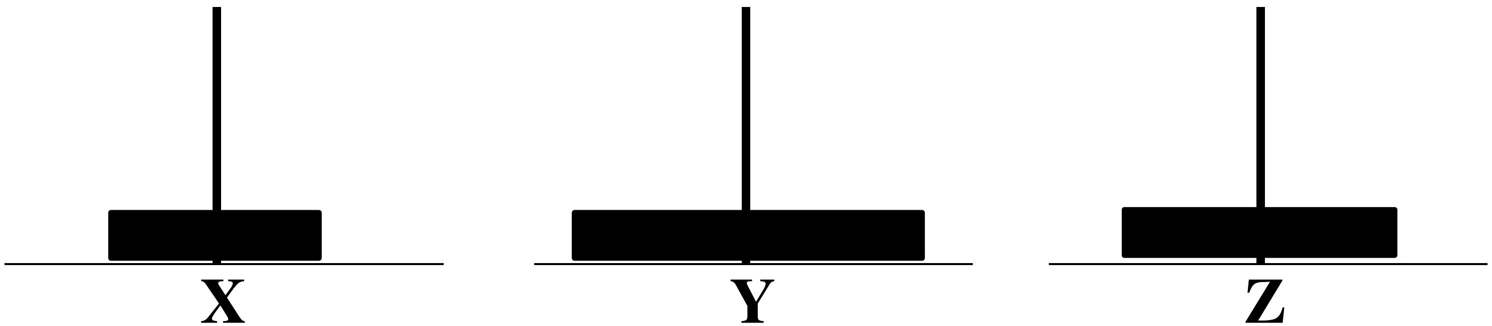
Torres de Hanói (cont.)



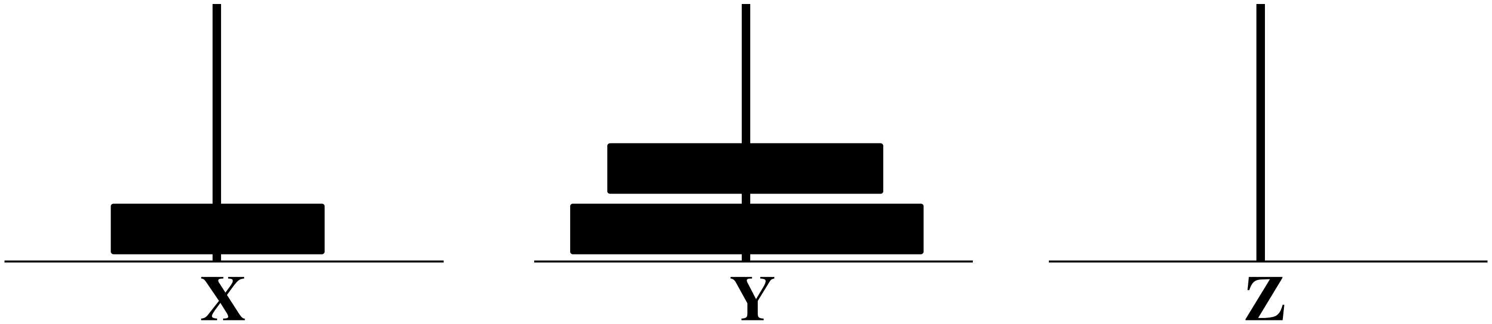


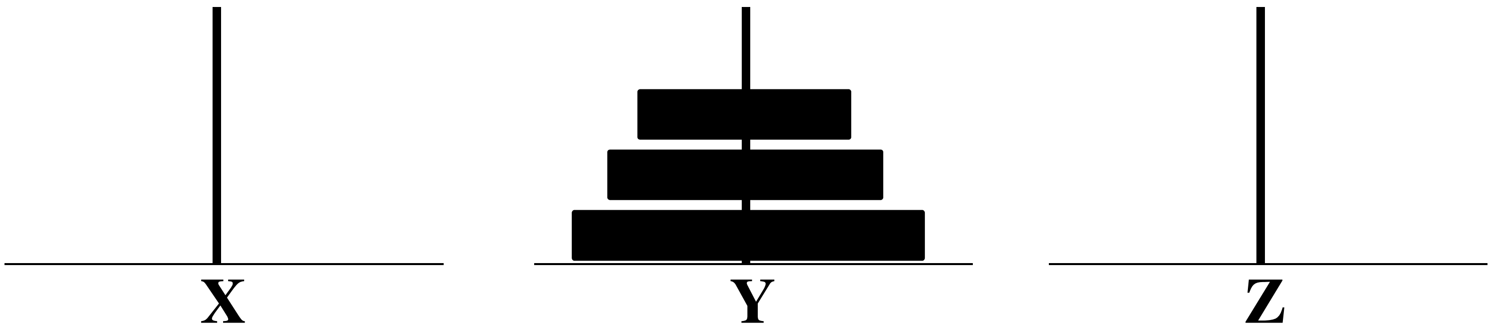
Torres de Hanói (cont.)





Torres de Hanói (cont.)





Torres de Hanói (cont.)

- Se n = 64 e 1 movimento leva 10^{-9} segundos:
  $T(64) \cong 585$ anos!!!
- A seguir é apresentado o procedimento
  HANOI(n, X, Y, Z)
- n discos são movidos do suporte X para o suporte Y usando o suporte Z

```
Procedimento HANOI(n, X, Y, Z)
    se n = 1 então
        escrever(“move um disco de ”, X, “ para ”, Y)
    senão
        HANOI(n – 1, X, Z, Y)
        escrever(“move um disco de ”, X, “ para ”, Y)
        HANOI(n – 1, Z, Y, X)
    fim_se
fim_procedimento

Algoritmo
    ler(NUM)
    HANOI(NUM, 1, 2, 3)
fim_algoritmo
```

Torres de Hanói (Complexidade)

- $T(n) = \begin{cases} 1 \text{ se } n = 1 & \text{(base da recursão)} \\ 2.T(n-1) + 1 \text{ se } n > 1 & \text{(fórmula recursiva)} \end{cases}$
- $T(n) = 2.T(n-1) + 1$ (I)
- $T(n-1) = 2.T(n-2) + 1$ (II)
- Substituindo (II) em (I):

  $T(n) = 2.(2.T(n-2) + 1) + 1 \quad \Rightarrow$

  $T(n) = 4.T(n-2) + 2 + 1$ (III)

Torres de Hanói (Complexidade)

- $T(n) = 4.T(n - 2) + 2 + 1$ (III)
- $T(n - 2) = 2.T(n - 3) + 1$ (IV)
- Substituindo (IV) em (III):

  $T(n) = 4.(2.T(n - 3) + 1) + 2 + 1 \quad \Rightarrow$

  $T(n) = 8.T(n - 3) + 4 + 2 + 1$ (V)
- $T(n - 3) = 2.T(n - 4) + 1$ (VI)
- Substituindo (VI) em (V):

  $T(n) = 8.(2.T(n - 4) + 1) + 4 + 2 + 1 \quad \Rightarrow$

Torres de Hanói (Complexidade)

$T(n) = 16.T(n - 4) + 8 + 4 + 2 + 1$ (VII)

- Generalizando (VII):

$T(n) = 2^k.T(n - k) + 2^{k-1} + 2^{k-2} + ... + 2^1 + 2^0$ (VIII)

- Esperamos chegar na base da recursão:

$n - k = 1 \Rightarrow k = n - 1$ (IX)

- Substituindo (IX) em (VIII):

$T(n) = 2^{n-1}.T(n - (n - 1)) + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0 \Rightarrow$

$T(n) = 2^{n-1}.T(1) + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0 \Rightarrow$

Torres de Hanói (Complexidade)

$T(n) = 2^{n-1}.1 + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0 \qquad \Rightarrow$

$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0$

- Soma dos termos de uma progressão geométrica:

  $S_n = a_1.(q^n - 1) / (q - 1)$

- Assim temos:

  $T(n) = 2^0.(2^n - 1) / (2 - 1) \qquad \Rightarrow \qquad T(n) = 2^n - 1$

- Logo T(n) é de ordem $O(2^n)$