# Estrutura de Dados I

Luciana Lee

# 1

## Apresentação da Disciplina

## Introdução à Complexidade de Algoritmos

## Revisão de Recursão

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;
  - implementações de listas utilizando vetor;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
    - listas simplesmente encadeadas e duplamente encadeadas;
    - listas circulares dinâmicas;
    - implementações de listas utilizando vetor;
    - operações de inserção, remoção e busca em listas.

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;
  - implementações de listas utilizando vetor;
  - operações de inserção, remoção e busca em listas.
- Estudo e implementação de Tipos Abstratos de Dados:

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;
  - implementações de listas utilizando vetor;
  - operações de inserção, remoção e busca em listas.
- Estudo e implementação de Tipos Abstratos de Dados:
  - Implementação de pilha e fila utilizando as estruturas de dados trabalhadas;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;
  - implementações de listas utilizando vetor;
  - operações de inserção, remoção e busca em listas.
- Estudo e implementação de Tipos Abstratos de Dados:
  - Implementação de pilha e fila utilizando as estruturas de dados trabalhadas;
- Algoritmos de busca em largura e busca em profundidade;

# Plano de Ensino

## Conteúdo Programático

- Introdução à análise de complexidade de algoritmos;
- Revisão de funções recursivas;
- Revisão de alocação dinâmica de memória em C;
- Estudo e implementação de estruturas de dados dinâmicas e estáticas:
  - listas simplesmente encadeadas e duplamente encadeadas;
  - listas circulares dinâmicas;
  - implementações de listas utilizando vetor;
  - operações de inserção, remoção e busca em listas.
- Estudo e implementação de Tipos Abstratos de Dados:
  - Implementação de pilha e fila utilizando as estruturas de dados trabalhadas;
- Algoritmos de busca em largura e busca em profundidade;
- Estudo e implementação de árvore binária de busca.

# Bibliografia I


- GUIMARÃES, Ângelo de Moura; LAGES, Newton Alberto de Castilho. Algoritmos e estruturas de dados. Rio de Janeiro: LTC, 1994. xii, 216 p. (Ciência da computação.). ISBN 9788521603788 (broch.).
- SILVA, Osmar Quirino da. Estrutura de dados e algoritmos usando C: fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007. xii, 460 p. ISBN 9788573936117 (broch.)
- TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 2008. xx, 884 p. ISBN 9788534603485 (broch.)
- ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Prentice Hall, 2008. 434 p. ISBN 9788576051480 (broch.)

# Bibliografia II


- JOYANES AGUILAR, Luis. Fundamentos de programação: algoritmos, estruturas de dados e objetos. São Paulo: McGraw-Hill, 2008. xxix, 690 p. ISBN 9788586804960 (broch.)
- ZIVIANI, Nivio. Projeto de algoritmos: com implementações em Java e C++. São Paulo: Thomson Learning, 2007. 621 p. ISBN 9788522105250 (broch.)
- LAFORE, Robert. Data structures & algorithms in Java. 2nd ed. Indianapolis, Ind.: Sams, 2003. 776 p. ISBN 9780672324536 (enc.)
- GOODRICH, Michael T; TAMASSIA, Roberto. Data structures and algorithms
- CORMEN, Thomas H. et al. *Introduction to algorithms.* 3rd ed. Cambridge, Mass.: The MIT Press; New York: McGraw-Hill, 2009. xix,1292 p. ISBN 9780262533058 (broch.)

# Avaliação

- As avaliações parciais serão compostas por duas provas teóricas (P1 e P2) e um trabalho prático (Trab).
- A Média Parcial será calculada da seguinte maneira:

$$MediaParcial = (P1 + P2) * 0.8 + Trab * 0.2,$$

- Caso o aluno não atinja $7,0$ pontos na média parcial, terá que fazer a prova final.
- A Média Final é dada por:

$$MediaFinal = (MediaParcial + ProvaFinal)/2.$$

# Avaliação

- Prova 1: 13/06/22
- Prova 2: 11/08/22
- Apresentação dos trabalhos: 01 e 04/08/22
- Prova Final: 22/08/22

# Introdução à complexidade de algoritmos

- Quando analisamos um algoritmo estamos verificando:

# Introdução à complexidade de algoritmos

- Quando analisamos um algoritmo estamos verificando:
  - O tempo de execução do algoritmo

# Introdução à complexidade de algoritmos

- Quando analisamos um algoritmo estamos verificando:
  - O tempo de execução do algoritmo
  - A quantidade de memória que o algoritmo demanda

# Introdução à complexidade de algoritmos

- Quando analisamos um algoritmo estamos verificando:
  - O tempo de execução do algoritmo
  - A quantidade de memória que o algoritmo demanda
- Tal análise é necessária quando precisamos escolher um algoritmo para resolver um determinado problema.

# Introdução à complexidade de algoritmos

Na área de análise de algoritmos, Knuth apontou dois tipos de problemas:

# Introdução à complexidade de algoritmos

Na área de análise de algoritmos, Knuth apontou dois tipos de problemas:

a) **Análise de um algoritmo em particular:** qual o custo de usar um dado algoritmo para um dado problema?

# Introdução à complexidade de algoritmos

Na área de análise de algoritmos, Knuth apontou dois tipos de problemas:

a) **Análise de um algoritmo em particular:** qual o custo de usar um dado algoritmo para um dado problema?

b) **Análise de uma classe de algoritmos:** qual é o melhor algoritmo para se resolver um determinado problema?

# Introdução à complexidade de algoritmos

Medidas para o custo de utilização de um algoritmo:

# Introdução à complexidade de algoritmos

Medidas para o custo de utilização de um algoritmo:

a) **Tempo de execução de um programa em um computador real:** os resultados dependem do compilador, do hardware e da memória disponível;

# Introdução à complexidade de algoritmos

Medidas para o custo de utilização de um algoritmo:

a) **Tempo de execução de um programa em um computador real:** os resultados dependem do compilador, do hardware e da memória disponível;

b) **Modelo matemático baseado em um computador idealizado:** geralmente são consideradas apenas as operações mais significativas.

# Introdução à complexidade de algoritmos

- Para medir o custo de execução de um algoritmo é comum definir uma **função de custo** ou **função de complexidade f**, em que **f**(**n**) é a medida do tempo necessário para executar um algoritmo em uma instância do problema de tamanho **n**.

# Introdução à complexidade de algoritmos

- Para medir o custo de execução de um algoritmo é comum definir uma **função de custo** ou **função de complexidade f**, em que **f**(**n**) é a medida do tempo necessário para executar um algoritmo em uma instância do problema de tamanho **n**.

- Neste caso, **f** é chamado de **função de complexidade de tempo** do algoritmo.

# Introdução à complexidade de algoritmos

- Para medir o custo de execução de um algoritmo é comum definir uma **função de custo** ou **função de complexidade f**, em que **f**(**n**) é a medida do tempo necessário para executar um algoritmo em uma instância do problema de tamanho **n**.
- Neste caso, **f** é chamado de **função de complexidade de tempo** do algoritmo.
- Se **f**(**n**) é uma medida da quantidade de memória, então chamamos **f** de **função de complexidade de espaço**.

# Exemplo: Resto de uma divisão

```
int resto (int a, int b){
  int q = a/b;
  int r = a - q*b;
  return r;
}
```

# Exemplo: Resto de uma divisão

```
int resto (int a, int b){
  int q = a/b; ................................1
  int r = a - q*b; ............................1
  return r;
}
```

# Exemplo: Resto de uma divisão

```
int resto (int a, int b){
  int q = a/b; ...................................1
  int r = a - q*b; ...............................1
  return r;
}
```

$$f(n) = 2$$

# Exemplo: Resto de uma divisão

```c
int retiraPrimeiro(int V[], int n){
  if (n > 0) {
    int valor = v[0];
    V[0] = V[n-1];
    printf("%d\n", valor);
    return n-1;
  }
  return n;
}
```

# Exemplo: Resto de uma divisão

```
int retiraPrimeiro(int V[], int n){
  if (n > 0) { ....................................1
    int valor = v[0]; ............................1
    V[0] = V[n-1]; ...............................1
    printf("%d\n", valor); ......................1
    return n-1;
  }
  return n;
}
```

# Exemplo: Resto de uma divisão

```
int retiraPrimeiro(int V[], int n){
  if (n > 0) { ...................................1
    int valor = v[0]; ............................1
    V[0] = V[n-1]; ...............................1
    printf("%d\n", valor); .......................1
    return n-1;
  }
  return n;
}
```

$$f(n) = 4$$

# Exemplo: Máximo de um conjunto

```c
int maximo (int lista[], int N) {
  int valormax, posmax;
  int i=0;
  if (N < 1) return -1;
  valormax = lista[0];
  posmax = 0;

  for (i=1 ; i<N ; i++) {
    if (valormax < lista[i]){
      valormax = lista[i];
      posmax = i;
    }
  }
  return posmax;
}
```

# Exemplo: Máximo de um conjunto

```
int maximo (int lista[], int N) {
  int valormax, posmax;
  int i=0; ....................................1
  if (N < 1) return -1; .......................1
  valormax = lista[0]; ........................1
  posmax = 0; .................................1

  for (i=1 ; i<N ; i++) { ....................?
    if (valormax < lista[i]){ ................?
      valormax = lista[i]; ...................?
      posmax = i; ............................?
    }
  }
  return posmax;
}
```

?

# Exemplo: Máximo de um conjunto

```
int maximo (int lista[], int N) {
  int valormax, posmax;
  int i=0; ..................................1
  if (N < 1) return -1; .....................1
  valormax = lista[0]; ......................1
  posmax = 0; ...............................1

  for (i = 1 ; i < N ; i++) { ...............?
    if (valormax < lista[i]){ ...............?
      valormax = lista[i]; ..................?
      posmax = i; ...........................?
    }
  }
  return posmax;
}
```

$$\text{for } (\overbrace{i=1}^{1} \; ; \; \overbrace{i<N}^{N} \; ; \; \overbrace{i++}^{N})$$

$N-1$

# Exemplo: Máximo de um conjunto

```
int maximo (int lista[], int N) {
  int valormax, posmax;
  int i=0; ....................................1
  if (N < 1) return -1; .........................1
  valormax = lista[0]; ..........................1
  posmax = 0; ...................................1

  for (i=1 ; i<N ; i++) { ................1 + 2N
    if (valormax < lista[i]){ ............N − 1
      valormax = lista[i]; ...............N − 1
      posmax = i; ........................N − 1
    }
  }
  return posmax;
}
```

$$\text{for } (\overbrace{i=1}^{1}\ ;\ \overbrace{i<N}^{N}\ ;\ \overbrace{i++}^{N})$$

$N-1$

$$f(N) = 4 + 1 + 2N + 3(N-1)$$

# Exemplo: Máximo de um conjunto

```
int maximo (int lista[], int N) {
  int valormax, posmax;
  int i=0;.......................................1
  if (N < 1) return -1; .........................1
  valormax = lista[0]; ..........................1
  posmax = 0;....................................1
```

$N-1$

```
  for (i = 1 ; i < N ; i++) { ..............1 + 2N
    if (valormax < lista[i]){ ...............N - 1
      valormax = lista[i]; ..................N - 1
      posmax = i; ...........................N - 1
    }
  }
  return posmax;
}
```

$$\overbrace{i=1}^{1} \; ; \; \overbrace{i<N}^{N} \; ; \; \overbrace{i++}^{N}$$

$$f(N) = 5N + 2$$

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]: inteiro):
VAR
    chave, i, j: inteiro;
Inicio:
    Para j ← 2 ate comprimento(A), faca:
        chave ← A[j];
        i ← j − 1;
        Enquanto i > 0 e A[i] > chave, faca:
            A[i + 1] ← A[i];
            i ← i − 1;
        Fim−Enquanto
        A[i + 1] ← chave;
    Fim−Para
Fim.
```

Qual é a complexidade de tempo do algoritmo?

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]: inteiro):
VAR
    chave, i, j: inteiro;
Inicio:
    Para j ← 2 ate comprimento(A), faca: ............. n − 1
        chave ← A[j]; ................................. n − 1
        i ← j − 1; .................................... n − 1
        Enquanto i > 0 e A[i] > chave, faca: ............. ?
            A[i + 1] ← A[i]; ............................. ?
            i ← i − 1; ................................... ?
        Fim−Enquanto
        A[i + 1] ← chave; ............................. n − 1
    Fim−Para
Fim.
```

Qual é a complexidade de tempo do algoritmo?

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]:inteiro):
VAR
    chave, i, j: inteiro;
Inicio:
    Para j ← 2 ate comprimento(A), faca: .............n − 1
        chave ← A[j]; ................................n − 1
        i ← j − 1; ...................................n − 1
        Enquanto i > 0 e A[i] > chave, faca: ............?
            A[i + 1] ← A[i]; ............................?
            i ← i − 1; ..................................?
        Fim−Enquanto
        A[i + 1] ← chave; ............................n − 1
    Fim−Para
Fim.
```

A cada iteração do laço **Para** temos *j* iterações do laço **Enquanto**.

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]: inteiro):
VAR
    chave, i, j: inteiro;
Inicio:
    Para j ← 2 ate comprimento(A), faca: ............. n − 1
        chave ← A[j]; ................................ n − 1
        i ← j − 1; ................................... n − 1
        Enquanto i > 0 e A[i] > chave, faca:
            A[i + 1] ← A[i];
            i ← i − 1;
        Fim−Enquanto
        A[i + 1] ← chave; ............................ n − 1
    Fim−Para
Fim.
```

$$\sum_{j=2}^{n} 3(j-1)$$

A cada iteração do laço **Para** temos *j* iterações do laço **Enquanto**.

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]:inteiro):
VAR
    chave, i, j: inteiro;
Inicio:
    Para j ← 2 ate comprimento(A), faca: ............ n − 1
        chave ← A[j]; ............................... n − 1
        i ← j − 1; .................................. n − 1
        Enquanto i > 0 e A[i] > chave, faca:  ⎫
            A[i + 1] ← A[i];                  ⎬  Σ_{j=2}^{n} 3(j − 1)
            i ← i − 1;                        ⎭
        Fim−Enquanto
        A[i + 1] ← chave; ........................... n − 1
    Fim−Para
Fim.
```

$$\text{Complexidade: } f(n) = 4n - 4 + \sum_{j=2}^{n} 3(j-1)$$

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]:inteiro):
VAR
      chave, i, j: inteiro;
Inicio:
      Para j ← 2 ate comprimento(A), faca: ............. n − 1
            chave ← A[j]; ................................ n − 1
            i ← j − 1; ................................... n − 1
            Enquanto i > 0 e A[i] > chave, faca:  ⎫
                  A[i + 1] ← A[i];                ⎬ Σ_{j=2}^{n} 3(j − 1)
                  i ← i − 1;                      ⎭
            Fim−Enquanto
            A[i + 1] ← chave; ............................ n − 1
      Fim−Para
Fim.
```

$$\sum_{j=2}^{n} 3(j-1)$$

Complexidade: $f(n) = 4n - 4 + 3n^2 - 1$

# Exemplo: Algoritmo de ordenação

```
Algoritmo InsertionSort(A[]: inteiro):
VAR
      chave, i, j: inteiro;
Inicio:
      Para j ← 2 ate comprimento(A), faca: .............. n − 1
            chave ← A[j]; ................................ n − 1
            i ← j − 1; ................................... n − 1
            Enquanto i > 0 e A[i] > chave, faca:    ⎫
                  A[i + 1] ← A[i];                  ⎬ Σ_{j=2}^{n} 3(j − 1)
                  i ← i − 1;                        ⎭
            Fim−Enquanto
            A[i + 1] ← chave; ............................ n − 1
      Fim−Para
Fim.
```

$$\sum_{j=2}^{n} 3(j-1)$$

Complexidade: $f(n) = 3n^2 + 4n - 5$

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

| $n$ | $f(n)$ | $g(n)$ |
|---|---|---|
| 5 | 28 | 105 |
| 10 | 103 | 110 |
| 11 | 124 | 111 |
| 20 | 403 | 120 |

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

$\Rightarrow$

| $n$ | $f(n)$ | $g(n)$ |
|---|---|---|
| 5 | 28 | 105 |
| 10 | 103 | 110 |
| 11 | 124 | 111 |
| 20 | 403 | 120 |

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

$\Rightarrow$

| $n$ | $f(n)$ | $g(n)$ |
|---|---|---|
| 5 | 28 | 105 |
| 10 | 103 | 110 |
| 11 | 124 | 111 |
| 20 | 403 | 120 |

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

| | $n$ | $f(n)$ | $g(n)$ |
|---|---|---|---|
| | 5 | 28 | 105 |
| | 10 | 103 | 110 |
| $\Rightarrow$ | 11 | 124 | 111 |
| | 20 | 403 | 120 |

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$

| | $n$ | $f(n)$ | $g(n)$ |
|---|---|---|---|
| | 5 | 28 | 105 |
| | 10 | 103 | 110 |
| | 11 | 124 | 111 |
| $\Rightarrow$ | 20 | 403 | 120 |

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$
- Quando estamos analisando algoritmos, consideramos para $n$ apenas valores *muito grandes*!

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$
- Quando estamos analisando algoritmos, consideramos para $n$ apenas valores *muito grandes*!
- Para valores <u>enormes</u> de $n$, temos que as funções: $f(n) = n^2$, $g(n) = n^2 - 10.000n$, $h(n) = n^2/5000$, etc. são todos "equivalentes". Ou seja, possuem a mesma "taxa de crescimento".

# Notação Assintótica

- Considere as funções $f(n) = n^2 + 3$ e $g(n) = n + 100$
- Quando estamos analisando algoritmos, consideramos para $n$ apenas valores *muito grandes*!
- Para valores <u>enormes</u> de $n$, temos que as funções: $f(n) = n^2$, $g(n) = n^2 - 10.000n$, $h(n) = n^2/5000$, etc. são todos "equivalentes". Ou seja, possuem a mesma "taxa de crescimento".
- Neste caso, estamos analisando as funções *assintoticamente*.

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \rightarrow \infty$.

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \to \infty$.
- Neste caso, é considerada apenas o componente mais significativo da função, ignorando-se as constantes e os componentes menos significativos.

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \to \infty$.
- Neste caso, é considerada apenas o componente mais significativo da função, ignorando-se as constantes e os componentes menos significativos.

$$f(n) = \underbrace{10n^2} + \underbrace{100n} - \underbrace{1000}$$

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \to \infty$.
- Neste caso, é considerada apenas o componente mais significativo da função, ignorando-se as constantes e os componentes menos significativos.

$$f(n) = n^2$$

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \to \infty$.
- Neste caso, é considerada apenas o componente mais significativo da função, ignorando-se as constantes e os componentes menos significativos.
- É interessante agrupar os algoritmos (ou problemas) em classes de comportamento assintótico.

# Classes de Comportamento Assintótico

- O comportamento assintótico de funções é medido quando o tamanho da entrada tende a infinito, ou seja, quando $n \to \infty$.
- Neste caso, é considerada apenas o componente mais significativo da função, ignorando-se as constantes e os componentes menos significativos.
- É interessante agrupar os algoritmos (ou problemas) em classes de comportamento assintótico.
- Quando dois algoritmos (ou problemas) estão na mesma classe de comportamento assintótico, dizemos que eles são equivalentes.

# Comportamento Assintótico de Funções

**Definição:**

Uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes $c$ e $n_0$ tais que, para $0 \leq n_0 \leq n$, temos $g(n) \leq c \cdot f(n)$.

# Comportamento Assintótico de Funções

**Definição:**

Uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes $c$ e $n_0$ tais que, para $0 \leq n_0 \leq n$, temos $g(n) \leq c \cdot f(n)$.

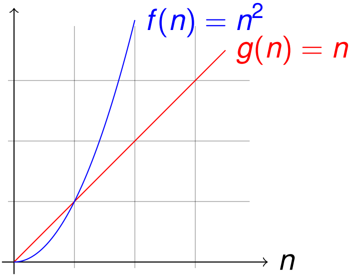
**Exemplo:** $g(n) = n$ e $f(n) = n^2$.

# Comportamento Assintótico de Funções

**Definição:**

Uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes $c$ e $n_0$ tais que, para $0 \leq n_0 \leq n$, temos $g(n) \leq c \cdot f(n)$.

**Exemplo:** $g(n) = n$ e $f(n) = n^2$.

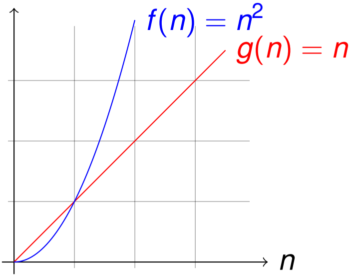


# Comportamento Assintótico de Funções

**Definição:**

Uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes $c$ e $n_0$ tais que, para $0 \leq n_0 \leq n$, temos $g(n) \leq c \cdot f(n)$.

**Exemplo:** $g(n) = n$ e $f(n) = n^2$.




$$c = 1$$
$$n_0 = 1$$

# Comportamento Assintótico de Funções

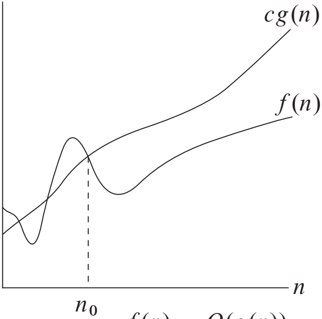
**Notação O:**

Uma função $f(n)$ é $O(g(n))$ se existem duas constantes positivas $c$ e $n_0$ tais que $f(n) \leq cg(n)$, para todo $n \geq n_0$.

# Comportamento Assintótico de Funções

## Notação O:

Uma função $f(n)$ é $O(g(n))$ se existem duas constantes positivas $c$ e $n_0$ tais que $f(n) \leq cg(n)$, para todo $n \geq n_0$.




$f(n) = O(g(n))$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$
- Complexidade linear: $f(n) = O(n)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$
- Complexidade linear: $f(n) = O(n)$
- Complexidade linearítmica: $f(n) = O(n \log n)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$
- Complexidade linear: $f(n) = O(n)$
- Complexidade linearítmica: $f(n) = O(n \log n)$
- Complexidade quadrática: $f(n) = O(n^2)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$
- Complexidade linear: $f(n) = O(n)$
- Complexidade linearítmica: $f(n) = O(n \log n)$
- Complexidade quadrática: $f(n) = O(n^2)$
- Complexidade cúbica: $f(n) = O(n^3)$

# Principais Classes

- Complexidade constante: $f(n) = O(1)$
- Complexidade logarítmica: $f(n) = O(\log n)$
- Complexidade linear: $f(n) = O(n)$
- Complexidade linearítmica: $f(n) = O(n \log n)$
- Complexidade quadrática: $f(n) = O(n^2)$
- Complexidade cúbica: $f(n) = O(n^3)$
- Complexidade exponencial: $f(n) = O(c^n)$, $O(n!)$

# Classes de Comportamento Assintótico

- No exemplo do cálculo do resto de uma divisão:

$$f(n) = 2 \Rightarrow f(n) = O(1).$$

# Classes de Comportamento Assintótico

- No exemplo do cálculo do resto de uma divisão:

$$f(n) = 2 \Rightarrow f(n) = O(1).$$

- No exemplo da remoção do primeiro elemento de um vetor:

$$f(n) = 4 \Rightarrow f(n) = O(1).$$

# Classes de Comportamento Assintótico

- No exemplo do cálculo do resto de uma divisão:

$$f(n) = 2 \Rightarrow f(n) = O(1).$$

- No exemplo da remoção do primeiro elemento de um vetor:

$$f(n) = 4 \Rightarrow f(n) = O(1).$$

- No exemplo do maior valor de um vetor:

$$f(n) = 5n + 2 \Rightarrow f(n) = O(n).$$

# Classes de Comportamento Assintótico

- No exemplo do cálculo do resto de uma divisão:

$$f(n) = 2 \Rightarrow f(n) = O(1).$$

- No exemplo da remoção do primeiro elemento de um vetor:

$$f(n) = 4 \Rightarrow f(n) = O(1).$$

- No exemplo do maior valor de um vetor:

$$f(n) = 5n + 2 \Rightarrow f(n) = O(n).$$

- No exemplo do algoritmo de ordenação:

$$f(n) = 3n^2 + 4n - 5 \Rightarrow f(n) = O(n^2).$$

# O que é recursão?

- Uma **instância de um problema** é um caso particular do problema

# O que é recursão?

- Uma **instância de um problema** é um caso particular do problema
- **Estrutura recursiva:** quando uma instância de um problema contém uma instância menor do problema.

# O que é recursão?

- Uma **instância de um problema** é um caso particular do problema
- **Estrutura recursiva:** quando uma instância de um problema contém uma instância menor do problema.
- Existem instâncias do problema que possuem **soluções triviais**.

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:
  1. Se a instância for trivial:

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:
  1. Se a instância for trivial:
     - ★ Resolva diretamente

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:

  1. Se a instância for trivial:
     - Resolva diretamente
  2. Senão:

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:
  1. Se a instância for trivial:
     - Resolva diretamente
  2. Senão:
     - reduza a instância corrente a uma instância menor

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:
  1. Se a instância for trivial:
     - Resolva diretamente
  2. Senão:
     - reduza a instância corrente a uma instância menor
     - utilize o método para resolver a instância menor

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:

  1. Se a instância for trivial:
     - ★ Resolva diretamente
  2. Senão:
     - ★ reduza a instância corrente a uma instância menor
     - ★ utilize o método para resolver a instância menor
     - ★ resolva a instância corrente utilizando o resultado obtido

# O que é recursão?

- Para resolvermos um problema com estrutura recursiva, basta executarmos os seguintes método:
  1. Se a instância for trivial:
     - Resolva diretamente
  2. Senão:
     - reduza a instância corrente a uma instância menor
     - utilize o método para resolver a instância menor
     - resolva a instância corrente utilizando o resultado obtido
- Um **algoritmo recursivo** é aquele que aplica tal método.

# Exemplo: Fatorial

- O fatorial de um número inteiro $n$ é definido da seguinte forma:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

# Exemplo: Fatorial

- O fatorial de um número inteiro $n$ é definido da seguinte forma:

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdots 2 \cdot 1}_{(n-1)!}$$

# Exemplo: Fatorial

- O fatorial de um número inteiro $n$ é definido da seguinte forma:

$$n! = n \cdot (n-1) \cdot \underbrace{(n-2) \cdots 2 \cdot 1}_{(n-2)!}$$

# Exemplo: Fatorial

- O fatorial de um número inteiro $n$ é definido da seguinte forma:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

- Outra forma de definir o fatorial de $n$ é:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n > 0. \end{cases}$$

# Exemplo: Fatorial

- Versão iterativa:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

```
int fatorial (int n){
  int fat = 1;
  if (n < 0) return 0;
  while (n > 0) {
    fat = fat * n;
    n--;
  }
  return fat;
}
```

# Exemplo: Fatorial

- Versão recursiva:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n > 0. \end{cases}$$

```
int fatoriaRec (int n) {
  if (n < 0) return 0;
  if (n == 0) return 1;
  return (n * fatorialRec(n-1));
}
```

# Exemplo: Fatorial

- Versão recursiva:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n > 0. \end{cases}$$

$$\texttt{int fatoriaRec (int n) \{}$$
$$\texttt{if (n < 0) return 0;}$$
$$\texttt{if (n == 0) return 1;}$$
$$\texttt{return (n * } \underbrace{\text{fatorialRec(n-1)}}_{\text{chamada recursiva}}\texttt{);}$$
$$\texttt{\}}$$

# Propriedades de Algoritmos Recursivos

- A ideia básica por trás de um algoritmo recursivo:

> "Para solucionar um problema, solucione um subproblema que seja uma instância menor do mesmo problema, e então use a solução dessa instância menor para solucionar o problema original."

# Propriedades de Algoritmos Recursivos

- A ideia básica por trás de um algoritmo recursivo:

  > "Para solucionar um problema, solucione um subproblema que seja uma instância menor do mesmo problema, e então use a solução dessa instância menor para solucionar o problema original."

- Os problemas menores devem em algum momento chegar ao caso base.

# Propriedades de Algoritmos Recursivos

- Podemos colocar duas regras simples:
  - Cada chamada recursiva deve ser em uma instância menor do mesmo problema.
  - As chamadas recursivas precisam em algum ponto alcançar um caso base, que é resolvido sem outra recursão.

# Exemplo: Palíndromo

## Palíndromo

Um **Palíndromo** é uma palavra que é soletrada do mesmo jeito de trás para frente.

# Exemplo: Palíndromo

**Palíndromo**

Um **Palíndromo** é uma palavra que é soletrada do mesmo jeito de trás para frente.

**Exemplos:**

| | | |
|---|---|---|
| ALA | ARARA | MIRIM |
| MAMAM | OSSO | MUSSUM |

# Exemplo: Palíndromo

- Versão iterativa:

```
int palindromo (char P[], int n){
    int i, j, meio;
    if (n < 0) return FALSE;
    meio = n/2;
    i = 0; j = n-1;
    while (i < meio) {
      if(P[i] == P[j]) {
        i++; j--;
      }
      else return FALSE;
    }
    return TRUE;
}
```

M U S S U M

$\uparrow$ $\uparrow$

$i$ $j$

# Exemplo: Palíndromo

- Versão iterativa:

```c
int palindromo (char P[], int n){
    int i, j, meio;
    if (n < 0) return FALSE;
    meio = n/2;
    i = 0; j = n-1;
    while (i < meio) {
      if(P[i] == P[j]) {
        i++; j--;
      }
      else return FALSE;
    }
    return TRUE;
}
```

M U S S U M

$\uparrow$ $i$ $\quad$ $\uparrow$ $j$

# Exemplo: Palíndromo

- Versão iterativa:

```
int palindromo (char P[], int n){
    int i, j, meio;
    if (n < 0) return FALSE;
    meio = n/2;
    i = 0; j = n-1;
    while (i < meio) {
      if(P[i] == P[j]) {
        i++; j--;
      }
      else return FALSE;
    }
    return TRUE;
}
```

$$\begin{array}{cccccc} \mathrm{M} & \mathrm{U} & \mathrm{S} & \mathrm{S} & \mathrm{U} & \mathrm{M} \\ & & \uparrow & \uparrow & & \\ & & i & j & & \end{array}$$

# Exemplo: Palíndromo

- Versão iterativa:

```
int palindromo (char P[], int n){
    int i, j, meio;
    if (n < 0) return FALSE;
    meio = n/2;
    i = 0; j = n-1;
    while (i < meio) {
      if(P[i] == P[j]) {
        i++; j--;
      }
      else return FALSE;
    }
    return TRUE;
}
```

A B A N A

$\uparrow$ $\uparrow$

$i$ $j$

# Exemplo: Palíndromo

- Versão iterativa:

```
int palindromo (char P[], int n){
    int i, j, meio;
    if (n < 0) return FALSE;
    meio = n/2;
    i = 0; j = n-1;
    while (i < meio) {
      if(P[i] == P[j]) {
        i++; j--;
      }
      else return FALSE;
    }
    return TRUE;
}
```

A B A N A

$\uparrow$ $\uparrow$

$i$ $j$

# Exemplo: Palíndromo

- Como é o algoritmo recursivo para determinar se uma palavra é palíndroma?

# Exemplo: Palíndromo

- Como é o algoritmo recursivo para determinar se uma palavra é palíndroma?
- Qual é o subproblema do problema original?

# Exemplo: Palíndromo

- Como é o algoritmo recursivo para determinar se uma palavra é palíndroma?
- Qual é o subproblema do problema original?
- Qual é o caso base?

# Exemplo: Palíndromo

- Como é o algoritmo recursivo para determinar se uma palavra é palíndroma?
- Qual é o subproblema do problema original?
- Qual é o caso base?
- Qual é o passo recursivo?

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

$$\begin{array}{cccccc} \mathbf{M} & \mathbf{U} & \mathbf{S} & \mathbf{S} & \mathbf{U} & \mathbf{M} \\ \uparrow & & & & & \uparrow \\ i & & & & & j \end{array}$$

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

$$
\begin{array}{cccccc}
\mathbf{M} & \mathbf{U} & \mathbf{S} & \mathbf{S} & \mathbf{U} & \mathbf{M} \\
 & \uparrow & & & \uparrow & \\
 & i & & & j &
\end{array}
$$

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

$$\begin{array}{cccccc} \textbf{M} & \textbf{U} & \textbf{S} & \textbf{S} & \textbf{U} & \textbf{M} \\ & & \uparrow & \uparrow & & \\ & & i & j & & \end{array}$$

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

**M U S S U M**

  - Dada uma palavra $P$ com o primeiro caractere na posição $i$ e o último caractere na posição $j$.

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

**M U S S U M**

  - Dada uma palavra $P$ com o primeiro caractere na posição $i$ e o último caractere na posição $j$.
  - Seja a palavra $P'$ obtida a partir de $P$, retirando-se os caracteres de $P$ nas posições $i$ e $j$.

# Exemplo: Palíndromo

- Qual é o subproblema do problema original?

**M U S S U M**

  - Dada uma palavra $P$ com o primeiro caractere na posição $i$ e o último caractere na posição $j$.
  - Seja a palavra $P'$ obtida a partir de $P$, retirando-se os caracteres de $P$ nas posições $i$ e $j$.
  - O subproblema é: verificar se $P'$ é palíndroma.

# Exemplo: Palíndromo

- Qual é o caso base?
  - Se a palavra possui nenhum caractere ou somente um caractere, então ela é palíndroma.
  - Para que a verificação entre os caracteres nas posições $i$ e $j$ faça sentido, temos que $i < j$.
  - Ou seja, a palavra deve possuir mais do que um caractere.

# Exemplo: Palíndromo

- Qual é o passo recursivo?
  - Dada uma palavra $P$ com o primeiro caractere na posição $i$ e o último caractere na posição $j$ e $P'$ definida anteriormente.
  - Se $P[i]$ e $P[j]$ forem diferentes, já podemos concluir que $P$ não é palíndroma.
  - Se $P[i]$ e $P[j]$ forem iguais, a palavra será palíndroma se $P'$ for palíndroma.

# Exemplo: Palíndromo

- Vamos para a implementação da função.

```
int palindromoRec(char P[], int i, int j) {




}
```

# Exemplo: Palíndromo

- Vamos para a implementação da função.

```
int palindromoRec(char P[], int i, int j) {
  if (i < j){ ⇐ Caso Base!


  }
  else return TRUE; ⇐ Caso Base!
}
```

# Exemplo: Palíndromo

- Vamos para a implementação da função.

```
int palindromoRec(char P[], int i, int j) {
  if (i < j){
    if (P[i] == P[j]) return palindromoRec(P, i+1, j-1); ⇐ Passo
Recursivo!

  }
  else return TRUE;
}
```

# Exemplo: Palíndromo

- Vamos para a implementação da função.

```
int palindromoRec(char P[], int i, int j) {
  if (i < j){
    if (P[i] == P[j]) return palindromoRec(P, i+1, j-1);
    else return FALSE;
  }
  else return TRUE;
}
```

Dúvidas?